Teach Yourself C++

Black_Devils B0ys

By : Hossein Hezami (**3v1l**)

C++ Basics

1.1 — Structure of a program

A computer program is a sequence of instructions that tell the computer what to do.

Statements and expressions

The most common type of instruction in a program is the **statement**. A statement in C++ is the smallest independent unit in the language. In human language, it is analogous to a sentence. We write sentences in order to convey an idea. In C++, we write statements in order to convey to the compiler that we want to perform a task. **Statements in C++ are terminated by a semicolon**.

There are many different kinds of statements in C++. The following are some of the most common types of simple statements:

```
1  int x;
2  x = 5;
3  cout << x;
```

`int x` is a **declaration statement**. It tells the compiler that x is a variable. All variables in a program must be declared before they are used. We will talk more about variables shortly. `x = 5` is an **assignment statement**. It assigns a value (5) to a variable (x). `cout << x;` is an output statement. It outputs the value of x (which we set to 5 in the previous statement) to the screen.

The compiler is also capable of resolving expressions. An **expression** is an mathematical entity that evaluates to a value. For example, in math, the expression 2+3 evaluates to the value 5. Expressions can involve values (such as 2), variables (such as x), operators (such as +) and functions (which return an output value based on some input value). They can be singular (such as 2, or x), or compound (such as 2+3, 2+x, x+y, or (2+x)*(y-3)).

For example, the statement `x = 2 + 3;` is a valid assignment statement. The expression 2+3 evaluates to the value of 5. This value of 5 is then assigned to x.

Functions

In C++, statements are typically grouped into units called functions. A **function** is a collection of statements that executes sequentially. Every C++ program must contain a special function called **main()**. When the C++ program is run, execution starts with the first statement inside of main(). Functions are typically written to do a very specific job. For example, a function named Max() might contain statements that figures out which of two numbers is larger. A function named CalculateGrade() might calculate a student's grade. We will talk more about functions later.

Libraries

Libraries are groups of functions that have been "packaged up" for reuse in many different programs. The core C++ language is actually very small and minimalistic — however, C++ comes with a bunch of libraries, known as the C++ standard libraries, that provide programmers with lots of extra functionality. For example, the iostream library contains functions for doing input and output. During the link stage of the compilation process, the libraries from the C++ standard library are the runtime support libraries that are linked into the program (this will be discussed further in lesson 1.4).

Taking a look at a sample program

Now that you have a brief understanding of what statements, functions, and libraries are, let's look at a simple hello world program.

Consider our hello world program:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "Hello world!" << endl;
7      return 0;
8  }
```

Line 1 is a special type of statement called a preprocessor directive. Preprocessor directives tell the compiler to perform a special task. In this case, we are telling the compiler that we would like to use the iostream library. The iostream library contains code that tells the compiler what *cout* and *endl* do. In other words, we need to include the iostream library in order to write to the screen.

Line 3 declares the main() function, which as you learned above, is mandatory. Every program must have a main() function.

Lines 4 and 8 tell the compiler which lines are part of the main function. Everything between the opening curly brace on line 4 and the closing curly brace on line 8 is considered part of the main() function.

Line 5 is our first statement (you can tell it's a statement because it ends with a semicolon). As you learned in the explanation for line 1, cout and endl live inside the iostream library. However, within iostream, they live inside a special compartment named std (short for standard). This using statement tells the compiler to look inside a compartment named std if it can't find cout or endl defined anywhere else. In other words, this statement is also necessary so the compiler can find cout and endl, which we use on line 6.

Line 6 is our output statement. Cout is a special object that represents the console/screen. The << symbol is an operator (much like + is an operator) called the output operator. Cout understands

that anything sent to it via the << operator should be printed on the screen. Endl is a special symbol that moves the cursor to the next line.

Line 7 is a new type of statement, called a **return statement**. When an executable program finishes running, it sends a value to the operating system that indicates whether it was run successfully or not. The return value of main() is used for this purpose. This particular return statement returns the value of 0 to the operating system, which means "everything went okay!". Non-zero numbers are typically used to indicate that something went wrong, and the program had to abort. We will discuss return statements in more detail when we discuss functions.

Conclusion

All of the programs we write will follow this template, or a variation on it. We will discuss each of the lines above in more detail in the upcoming sections.

Quiz

The following quiz is meant to reinforce your understanding of the material presented above.

1) What is the difference between a statement and an expression?
2) What is the difference between a function and a library?
3) What symbol do statements in C++ end with?

Quiz Answers

To see these answers, select the area below with your mouse.

1) Answer

A statement is a "complete sentence" that tells the compiler to perform a particular task. An expression is a mathematical entity that evaluates to a value. Expressions are often used inside of statements.

2) Answer

A function is a collection of statements that executes sequentially, typically designed to perform a specific job. A library is a collection of functions that is meant to be reused by many programs.

3) Answer

The semicolon (;)

1.2 — Comments

Types of comments

A **comment** is a line (or multiple lines) of text that are inserted into the source code to explain what the code is doing. In C++ there are two kinds of comments.

The `//` symbol begins a C++ single-line comment, which tells the compiler to ignore everything to the end of the line. For example:

```
1  cout << "Hello world!" << endl; // Everything from here to the right is ignored.
```

Typically, the single line comment is used to make a quick comment about a single line of code.

```
1  cout << "Hello world!" << endl; // cout and endl live in the iostream library
2  cout << "It is very nice to meet you!" << endl; // these comments make the code hard to read
3  cout << "Yeah!" << endl; // especially when lines are different lengths
```

Having comments to the right of a line can make the both the code and the comment hard to read, particularly if the line is long. Consequently, the // comment is often placed above the line it is commenting.

```
1  // cout and endl live in the iostream library
2  cout << "Hello world!" << endl;
3
4  // this is much easier to read
5  cout << "It is very nice to meet you!" << endl;
6
7  // don't you think so?
8  cout << "Yeah!" << endl;
```

The `/*` and `*/` pair of symbols denotes a C-style multi-line comment. Everything in between the symbols is ignored.

```
1  /* This is a multi-line comment.
2     This line will be ignored.
3     So will this one. */
```

Since everything between the symbols is ignored, you will sometimes see programmers “beautify” their multiline comments:

```
1  /* This is a multi-line comment.
2   * the matching asterisks to the left
3   * can make this easier to read
4   */
```

Multi-line style comments do not nest. Consequently, the following will have unexpected results:

```
1  /* This is a multiline /* comment */ this is not inside the comment */
2  //                                  ^ comment ends here
```

Rule: Never nest comments inside of other comments.

Proper use of comments

Typically, comments should be used for three things. At the library, program, or function level, comments should be used to describe *what* the library, program, or function, does. For example:

```
1  // This program calculate the student's final grade based on his test and homework scores.
1  // This function uses newton's method to approximate the root of a given equation.
1  // The following lines generate a random item based on rarity, level, and a weight factor.
```

All of these comments give the reader a good idea of what the program is trying to accomplish without having to look at the actual code. The user (possibly someone else, or you if you're trying to reuse code you've already written in the future) can tell at a glance whether the code is relevant to what he or she is trying to accomplish. This is particularly important when working as part of a team, where not everybody will be familiar with all of the code.

Second, within the library, program, or function described above, comments should be used to describe *how* the code is going to accomplish it's goal.

```
1  /* To calculate the final grade, we sum all the weighted midterm and homework scores
2      and then divide by the number of scores to assign a percentage.  This percentage is
3      used to calculate a letter grade. */
1  // To generate a random item, we're going to do the following:
2  // 1) Put all of the items of the desired rarity on a list
3  // 2) Calculate a probability for each item based on level and weight factor
4  // 3) Choose a random number
5  // 4) Figure out which item that random number corresponds to
6  // 5) Return the appropriate item
```

These comments give the user an idea of how the code is going to accomplish it's goal without going into too much detail.

At the statement level, comments should be used to describe *why* the code is doing something. A bad statement comment explains *what* the code is doing. If you ever write code that is so complex that needs a comment to explain *what* a statement is doing, you probably need to rewrite your code, not comment it.

Here are some examples of bad line comments and good statement comments.

Bad comment:

```
1  // Set sight range to 0
2  sight = 0;
```

(yes, we already can see that sight is being set to 0 by looking at the statement)

Good comment:

```
1  // The player just drank a potion of blindness and can not see anything
2  sight = 0;
```

(now we know WHY the player's sight is being set to 0)

Bad comment:

```
1  // Calculate the cost of the items
2  cost = items / 2 * storePrice;
```

(yes, we can see that this is a cost calculation, but why is items divided by 2?)

Good comment:

```
1  // We need to divide items by 2 here because they are bought in pairs
2  cost = items / 2 * storePrice;
```

(now we know!)

Programmers often have to make a tough decision between solving a problem one way, or solving it another way. Comments are a great way to remind yourself (or tell somebody else) the reason you made a one decision instead of another.

Good comments:

```
1  // We decided to use a linked list instead of an array because
2  // arrays do insertion too slowly.
```

```
1  // We're going to use newton's method to find the root of a number because
2  // there is no deterministic way to solve these equations.
```

Finally, comment should be written in a way that makes sense to someone who has no idea what the code does. It is often the case that a programmer will say "It's obvious what this does! There's no way I'll forget about this". Guess what? It's not obvious, and you will be amazed how quickly you forget. :) You (or someone else) will thank you later for writing down the what, how, and why of your code in human language. Reading individual lines of code is easy. Understanding what goal they are meant to accomplish is not.

To summarize:

- At the library, program, or function level, describe *what*
- Inside the library, program, or function, describe *how*
- At the statement level, describe *why*.

Commenting out code

Commenting out sections of code (temporarily) can be useful in several cases:

1) You're working on a new piece of code that won't compile yet, and you need to run the program. The compiler won't let you run if there are compiler errors. Commenting out the in-progress code will allow the program to compile so you can run it.

2) You want to change the way your program executes by commenting out individual lines of code. For example, you think a particular function call is crashing your program. Commenting out the function call and running your program may indicate whether that function was the culprit.

In our upcoming examples, we will remind Visual Studio Express users that they need to start their programs with `#include "stdafx.h"`. However, we will typically comment out this line of code, because it will cause compile errors on other compilers. Visual Studio Express users must uncomment the line before they compile.

1.3 — A first look at variables (and cin)

Variables

A statement such as `x = 5` seems obvious enough. As you would guess, we are assigning the value of 5 to x. But what exactly is x? x is a variable.

A **variable** in C++ is a name for a piece of memory that can be used to store information. You can think of a variable as a mailbox, or a cubbyhole, where we can put and retrieve information. All computers have memory, called RAM (random access memory), that is available for programs to use. When a variable is declared, a piece of that memory is set aside for that variable.

In this section, we are only going to consider integer variables. An **integer** is a whole number, such as 1, 2, 3, -1, -12, or 16. An integer variable is a variable that can only hold an integer value.

In order to declare a variable, we generally use a **declaration statement**. Here's an example of declaring variable x as an integer variable (one that can hold integer values):

```
1  int x;
```

When this statement is executed by the CPU, a piece of memory from RAM will be set aside. For the sake of example, let's say that the variable x is assigned memory location 140. Whenever the program sees the value x in an expression or statement, it knows that it should look in memory location 140.

One of the most common operations done with variables is assignment. To do this, we use the assignment operator, more commonly known as equals, more commonly known as the = symbol. When the CPU executes a statement such as `x = 5;`, it translates this to "put the value of 5 in memory location 140".

Later in our program, we could print that value to the screen using cout:

```
1  cout << x;  // prints the value of x (memory location 140) to the console
```

In C++, variables are also known as l-values (pronounced ell-values). An **l-value** is a value that has an address (in memory). Since all variables have addresses, all variables are l-values. They were originally named l-values because they are the only values that can be on the left side of an assignment statement. When we do an assignment, the left hand side of the assignment operator must be an l-value. Consequently, a statement like `5 = 6;` will cause a compile error, because 5 is not an l-value. The value of 5 has no memory, and thus nothing can be assigned to it. 5 means

5, and it's value can not be reassigned. When an l-value has a value assigned to it, the current value is overwritten.

The opposite of l-values are r-values. An **r-value** refers to any value that can be assigned to an l-value. r-values are always evaluated to produce a single value. Examples of r-values are single numbers (such as 5, which evaluates to 5), variables (such as x, which evaluates to whatever number was last assigned to it), or expressions (such as 2+x, which evaluates to the last value of x plus 2).

Here is an example of some assignment statements, showing how the r-values evaluate:

```
1  int y;       // declare y as an integer variable
2  y = 4;       // 4 evaluates to 4, which is then assigned to y
3  y = 2 + 5;   // 2 + 5 evaluates to 7, which is then assigned to y
4
5  int x;       // declare x as an integer variable
6  x = y;       // y evaluates to 7, which is then assigned to x.
7  x = x;       // x evaluates to 7, which is then assigned to x (useless!)
8  x = x + 1;   // x + 1 evaluates to 8, which is then assigned to x.
```

There are two important things to note. First, there is no guarantee that your variables will be assigned the same memory address each time your program is run. The first time you run your program, x may be assigned to memory location 140. The second time, it may be assigned to memory location 168. Second, when a variable is assigned to a memory location, the value in that memory location is undefined (in other words, whatever value was there last is still there).

This can lead to interesting (and by interesting, we mean dangerous) results. Consider the following short program:

```
1  // #include "stdafx.h" // Uncomment if Visual Studio user
2  #include <iostream>
3
4  int main()
5  {
6      using namespace std;     // gives us access to cout and endl
7      int x;                   // declare an integer variable named x
8
9      // print the value of x to the screen (dangerous, because x is
   uninitialized)
10     cout << x << endl;
11 }
```

In this case, the computer will assign some unused memory to x. It will then send the value residing in that memory location to cout, which will print the value. But what value will it print? The answer is "who knows!". You can try running this program in your compiler and see what value it prints. To give you an example, when we ran this program with an older version of the Visual Studio compiler, cout printed the value `-858993460`. Some newer compilers, such as Visual Studio 2005 Express will pop up a debug error message if you run this program from within the IDE.

A variable that has not been assigned a value is called an **uninitialized variable**. Uninitialized variables are very dangerous because they cause intermittent problems (due to having different values each time you run the program). This can make them very hard to debug. Most modern compilers will print warnings at compile-time if they can detect a variable that is used without being initialized. For example, compiling the above program on Visual Studio 2005 express produced the following warning:

```
c:\vc2005projects\test\test\test.cpp(11) : warning C4700: uninitialized local variable 'x' used
```

A good rule is to always assign values to variables when they are declared. C++ makes this easy by letting you assign values on the same line as the declaration of the variable:

```
1  int x = 0; // declare integer variable x and assign the value of 0 to it.
```

This ensures that your variable will always have a consistent value, making it easier to debug if something goes wrong somewhere else.

One common trick that experienced programmers use is to assign the variable an initial value that is outside the range of meaningful values for that variable. For example, if we had a variable to store the number of cats the old lady down the street has, we might do the following:

```
1  int cats = -1;
```

Having -1 cats makes no sense. So if later, we did this:

```
1  cout << cats << " cats" << endl;
```

and it printed "-1 cats", we know that the variable was never assigned a real value correctly.

Rule: Always assign values to your variables when you declare them.

We will discuss variables in more detail in an upcoming section.

cin

cin is the opposite of cout: whereas cout prints data to the console, cin reads data from the console. Now that you have a basic understanding of variables, we can use cin to get input from the user and store it in a variable.

```
1  //#include "stdafx.h" // Uncomment this line if using Visual Studio
2  #include <iostream>
3
4  int main()
5  {
6      using namespace std;
       cout << "Enter a number: "; // ask user for a number
       int x;
```

```
    cin >> x; // read number from console and store it in x
    cout << "You entered " << x << endl;
    return 0;
}
```

Try compiling this program and running it for yourself. When you run the program, it will print "Enter a number: " and then wait for you to enter one. Once you enter a number (and press enter), it will print "You entered " followed by the number you just entered.

This is an easy way to get input from the user, and we will use it in many of our examples going forward.

Quiz
What values does this program print?

```
int x = 5;
x = x - 2;
cout << x << endl; // #1

int y = x;
cout << y << endl; // #2

// x + y is an r-value in this context, so evaluate their values
cout << x + y << endl; // #3

cout << x << endl; // #4

int z;
cout << z << endl; // #5
```

Quiz Answers

To see these answers, select the area below with your mouse.

1) Answer
The program outputs 3. x – 2 evaluates to 3, which was assigned to x.

2) Answer
The program outputs 3. y was assigned the value of x, which evaluated to 3.

3) Answer
The program outputs 6. x + y evaluates to 6. There was no assignment here.

4) Answer
The program outputs 3. The value of x is still 3 because it was never reassigned.

5) Answer
The output is indeterminate . z is an uninitialized variable.

1.4 — A first look at functions

A **function** is a sequence of statements designed to do a particular job. You already know that every program must have a function named main(). However, most programs have many functions, and they all work analogously to main.

Often, your program needs to interrupt what it is doing to temporarily do something else. You do this in real life all the time. For example, you might be reading a book when you remember you need to make a phone call. You put a bookmark in your book, make the phone call, and when you are done with the phone call, you return to your book where you left off.

C++ programs work the same way. A program will be executing statements sequentially inside one function when it encounters a function call. A **function call** is an expression that tells the CPU to interrupt the current function and execute another function. The CPU "puts a bookmark" at the current point of execution, and then **calls** (executes) the function named in the function call. When the called function terminates, the CPU goes back to the point it bookmarked, and resumes execution.

Here is a sample program that shows how new functions are declared and called:

```
1  //#include <stdafx.h> // Visual Studio users need to uncomment this line
2  #include <iostream>
3
4  // Declaration of function DoPrint()
5  void DoPrint()
6  {
7      using namespace std;  // we need this in each function that uses cout
   and endl
8      cout << "In DoPrint()" << endl;
9  }
10
11 // Declaration of main()
12 int main()
13 {
14     using namespace std;  // we need this in each function that uses cout
   and endl
15     cout << "Starting main()" << endl;
16     DoPrint(); // This is a function call to DoPrint()
17     cout << "Ending main()" << endl;
18     return 0;
   }
```

This program produces the following output:

```
Starting main()
In DoPrint()
Ending main()
```

This program begins execution at the top of main(), and the first line to be executed prints `Starting main()`. The second line in main is a function call to DoPrint. At this point, execution of statements in main() is suspended, and the CPU jumps to DoPrint(). The first (and only) line in DoPrint prints `In DoPrint()`. When DoPrint() terminates, the caller (main()) resumes execution where it left off. Consequently, the next statment executed in main prints `Ending main()`.

Functions can be called multiple times:

```
1  //#include <stdafx.h> // Visual Studio users need to uncomment this line
2  #include <iostream>
3
4  // Declaration of function DoPrint()
5  void DoPrint()
6  {
7      using namespace std;
8      cout << "In DoPrint()" << endl;
9  }
10
11 // Declaration of main()
12 int main()
13 {
14     using namespace std;
15     cout << "Starting main()" << endl;
16     DoPrint(); // This is a function call to DoPrint()
17     DoPrint(); // This is a function call to DoPrint()
18     DoPrint(); // This is a function call to DoPrint()
19     cout << "Ending main()" << endl;
20     return 0;
21 }
```

This program produces the following output:

```
Starting main()
In DoPrint()
In DoPrint()
In DoPrint()
Ending main()
```

In this case, main() is interrupted 3 times, once for each call to DoPrint().

Main isn't the only function that can call other functions. In the following example, DoPrint() calls a second function, DoPrint2().

```
1  //#include <stdafx.h> // Visual Studio users need to uncomment this line
2  #include <iostream>
3
4  void DoPrint2()
5  {
6      using namespace std;
7      cout << "In DoPrint2()" << endl;
```

```
}

// Declaration of function DoPrint()
void DoPrint()
{
    using namespace std;
    cout << "Starting DoPrint()" << endl;
    DoPrint2(); // This is a function call to DoPrint2()
    DoPrint2(); // This is a function call to DoPrint2()
    cout << "Ending DoPrint()" << endl;
}

// Declaration of main()
int main()
{
    using namespace std;
    cout << "Starting main()" << endl;
    DoPrint(); // This is a function call to DoPrint()
    cout << "Ending main()" << endl;
    return 0;
}
```

This program produces the following output:

```
Starting main()
Starting DoPrint()
In DoPrint2()
In DoPrint2()
Ending DoPrint()
Ending main()
```

Return values

If you remember, when main finishes executing, it returns a value back to the operating system (the caller) by using a return statement. Functions you write can return a single value to their caller as well. We do this by changing the return type of the function in the function's declaration. A return type of **void** means the function does not return a value. A return type of int means the function returns an integer value to the caller.

```
// void means the function does not return a value to the caller
void ReturnNothing()
{
    // This function does not return a value
}

// int means the function returns an integer value to the caller
int Return5()
{
    return 5;
}
```

Let's use these functions in a program:

```
cout << Return5(); // prints 5
cout << Return5() + 2; // prints 7
cout << ReturnNothing(); // This will not compile
```

In the first statement, Return5() is executed. The function returns the value of 5 back to the caller, which passes that value to cout.

In the second statement, Return5() is executed and returns the value of 5 back to the caller. The expression 5 + 2 is then evaluated to 7. The value of 7 is passed to cout.

In the third statement, ReturnNothing() returns void. It is not valid to pass void to cout, and the compiler will give you an error when you try to compile this line.

One commonly asked question is, "Can my function return multiple values using a return statement?". The answer is no. Functions can only return a single value using a return statement. However, there are ways to work around the issue, which we will discuss when we get into the in-depth section on functions.

Returning to main

You now have the conceptual tools to understand how the main() function actually works. When the program is executed, the operating system makes a function call to main(). Execution then jumps to the top of main. The statements in main are executed sequentially. Finally, main returns a integer value (usually 0) back to the operating system. This is why main is declared as `int main()`.

Some compilers will let you get away with declaring main as `void main()`. Technically this is illegal. When these compilers see `void main()`, they interpret it as:

```
int main()
{
    // your code here
    return 0;
}
```

You should always declare main as returning an int and your main function should return 0 (or another integer if there was an error).

Parameters

In the return values subsection, you learned that a function can return a value back to the caller. **Parameters** are used to allow the caller to pass information to a function! This allows functions to be written to perform generic tasks without having to worry about the specific values used, and leaves the exact values of the variables up to the caller.

This is a case that is best learned by example. Here is an example of a very simple function that adds two numbers together and returns the result to the caller.

```
1  //#include <stdafx.h> // Visual Studio users need to uncomment this line
2  #include <iostream>
3
4  // add takes two integers as parameters, and returns the result of their
   sum
5  // add does not care what the exact values of x and y are
6  int add(int x, int y)
7  {
8      return x + y;
9  }
10
11 int main()
12 {
13     using namespace std;
14     // It is the caller of add() that decides the exact values of x and y
15     cout << add(4, 5) << endl; // x=4 and y=5 are the parameters
16     return 0;
17 }
```

When function add() is called, x is assigned the value 4, and y is assigned the value 5. The function evaluates x + y, which is the value 9, and then returns this value to the caller. This value of 9 is then sent to cout to be printed on the screen.

Output:

```
9
```

Let's take a look at a couple of other calls to functions():

```
1  //#include <stdafx.h> // Visual Studio users need to uncomment this line
2  #include <iostream>
3
4  int add(int x, int y)
5  {
6      return x + y;
7  }
8
9  int multiply(int z, int w)
10 {
11     return z * w;
12 }
13
14 int main()
15 {
16     using namespace std;
17     cout << add(4, 5) << endl; // evalutes 4 + 5
18     cout << add(3, 6) << endl; // evalues 3 + 6
```

```
16      cout << add(1, 8) << endl; // evalues 1 + 8
17
18      int a = 3;
19      int b = 5;
20      cout << add(a, b) << endl; // evaluates 3 + 5
21
22      cout << add(1, multiply(2, 3)) << endl; // evalues 1 + (2 * 3)
23      cout << add(1, add(2, 3)) << endl; // evalues 1 + (2 + 3)
24      return 0;
    }
```

This program produces the output:

```
9
9
9
8
7
6
```

The first three statements are straightforward.

The fourth is relatively easy as well:

```
1   int a = 3;
2   int b = 5;
3   cout << add(a, b) << endl; // evaluates 3 + 5
```

In this case, add() is called where x = a and y = b. Since a = 3 and b = 5, add(a, b) = add(3, 5), which resolves to 8.

Let's take a look at the first tricky statement in the bunch:

```
1   cout << add(1, multiply(2, 3)) << endl; // evalues 1 + (2 * 3)
```

When the CPU tries to call function add(), it assigns x = 1, and y = multiply(2, 3). y is not an integer, it is a function call that needs to be resolved. So before the CPU calls add(), it calls multiply() where z = 2 and w = 3. multiply(2, 3) produces the value of 6, which is assigned to add()'s parameter y. Since x = 1 and y = 6, add(1, 6) is called, which evaluates to 7. The value of 7 is passed to cout.

Or, less verbosely (where the => symbol is used to represent evaluation): add(1, multiply(2, 3)) => add(1, 6) => 7

The following statement looks tricky because one of the parameters given to add() is another call to add().

```
1   cout << add(1, add(2, 3)) << endl; // evalues 1 + (2 + 3)
```

But this case works exactly the same as the above case where one of the parameters is a call to multiply().

Before the CPU can evaluate the outer call to add(), it must evaluate the inner call to add(2, 3). add(2, 3) evaluates to 5. Now it can evaluate add(1, 5), which evaluates to the value 6. cout is passed the value 6.

Less verbosely:
add(1, add(2, 3)) => add(1, 5) => 6

Effectively using functions

One of the biggest challenges new programmers encounter (besides learning the language) is learning when and how to use functions effectively. Functions offer a great way to break your program up into manageable and reusable parts, which can then be easily connected together to perform a larger and more complex task. By breaking your program into smaller parts, the overall complexity of the program is reduced, which makes the program both easier to write and to modify.

Typically, when learning C++, you will write a lot of programs that involve 3 subtasks:

1. Reading inputs from the user
2. Calculating a value from the inputs
3. Printing the calculated value

For simple programs, reading inputs from the user can generally be done in main(). However, step #2 is a great candidate for a function. This function should take the user inputs as a parameter, and return the calculated value. The calculated value can then be printed (either directly in main(), or by another function if the calculated value is complex or has special printing requirements).

A good rule of thumb is that each function should perform one (and only one) task. New programmers often write functions that combine steps 2 and 3 together. However, because calculating a value and printing it are two different tasks, this violates the one and only one task guideline. Ideally, a function that calculates a value should return the value to the caller and let the caller decide what to do with the calculated value.

Quiz

1) What's wrong with this program fragment?

```
1  void multiply(int x, int y)
2  {
3      return x * y;
4  }
5
   int main()
```

```
6  {
7      cout << multiply(4, 5) << endl;
8      return 0;
   }
```

2) What's wrong with this program fragment?

```
1  int multiply(int x, int y)
2  {
3      int product = x * y;
4  }
5
6  int main()
7  {
8      cout << multiply(4, 5) << endl;
9      return 0;
   }
```

3) What value does the following program fragment print?

```
1  int add(int x, int y, int z)
2  {
3      return x + y + z;
4  }
5
6  int multiply(int x, int y)
7  {
8      return x * y;
9  }
10
11 int main()
12
13 {
14     cout << multiply(add(1, 2, 3), 4) << endl;
15     return 0;
   }
```

4) Write a function called doubleNumber() that takes one integer parameter and returns double it's value.

5) Write a complete program that reads an integer from the user (using cin, discussed in section 1.3), doubles it using the doubleNumber() function you wrote for question 4, and then prints the doubled value out to the console.

Quiz Answers

To see these answers, select the area below with your mouse.

1) Answer

multiply() is defined as returning void, which means it can't return a value. Since the function is trying to return a value, this function will produce a compiler error. The function should return an int.

2) Answer

multiply() calculates a value and puts the result in a variable, but never returns the value to the caller. Because there is no return statement, and the function is supposed to return an int, this will produce a compiler error.

3) Answer

multiply is called where x = add(1, 2, 3), and y = 4. First, the CPU resolves x = add(1, 2, 3), which returns 1 + 2 + 3, or x = 6. multiply(6, 4) = 24, which is the answer.

4) Answer

```
1  int doubleNumber(int x)
2  {
3      return 2 * x;
   }
```

5) Answer

```
1  #include <iostream>
2
3  int doubleNumber(int x)
4  {
5      return 2 * x;
   }
6
7  int main()
8  {
9      using namespace std;
10     int x;
11     cin >> x;
12     cout << doubleNumber(x) << endl;
13     return 0;
   }
14
15 // The following is an alternate way of doing main:
16 int main()
17 {
18     using namespace std;
19     int x;
20     cin >> x;
21     x = doubleNumber(x);
22     cout << x << endl;
       return 0;
   }
```

1.5 — A first look at operators

Revisiting expressions

In the section Introduction to programming, we had defined an expression as "A mathematical entity that evaluates to a value". However, the term *mathematical entity* is somewhat vague. More precisely, an **expression** is a combination of literals, variables, operators, and functions that evaluates to a value.

A **literal** is simply a number, such as 5, or 3.14159. When we talk about the expression "3 + 4″, both 3 and 4 are literals. Literals always evaluate to themselves.

You have already seen variables and functions. Variables evaluate to the values they hold. Functions evaluate to produce a value of the function's return type. Because functions that return void do not have return values, they are usually not part of expressions.

Literals, variables, and functions are all known as operands. **Operands** are the objects of an expression that are acted upon. Operands supply the data that the expression works with.

Operators

The last piece of the expressions puzzle is operators. **Operators** tell how to combine the operands to produce a new result. For example, in the expression "3 + 4″, the + is the plus operator. The + operator tells how to combine the operands 3 and 4 to produce a new value (7).

You are likely already quite familiar with standard arithmetic operators, including addition (+), subtraction (-), multiplication (*), and division (/). Assignment (=) is an operator as well.

Operators come in two types:
Unary operators act on one operand. An example of a unary operator is the – operator. In the expression `-5`, the – operator is only being applied to one operand (5) to produce a new value (-5).

Binary operators act on two operands (known as left and right). An example of a binary operator is the + operator. In the expression `3 + 4`, the + operator is working with a left operand (3) and a right operand (4) to produce a new value (7).

Note that some operators have more than one meaning. For example, the – operator has two contexts. It can be used in unary form to invert a number's sign (eg. -5), or it can be used in binary form to do arithmetic subtraction (eg. 4 – 3).

Conclusion

This is just the tip of the iceberg in terms of operators. We will take an in-depth look at operators in more detail in a future section.

1.6 — Whitespace and basic formatting

Whitespace is a term that refers to characters that are used for formatting purposes. In C++, this refers primarily to spaces, tabs, and (sometimes) newlines. The C++ compiler generally ignores whitespace, with a few minor exceptions.

Consequently, the following statements all do the exact same thing:

```
1  cout << "Hello world!";
2
3  cout                <<             "Hello world!";
4
5          cout <<        "Hello world!";
6
7  cout
8      << "Hello world!";
```

Even the last statement with the newline in it compiles just fine.

The following functions all do the same thing:

```
1  int add(int x, int y) { return x + y; }
2
3  int add(int x, int y) {
4      return x + y; }
5
6  int add(int x, int y)
7  {     return x + y; }
8
9  int add(int x, int y)
10 {
11     return x + y;
12 }
```

One exception where the C++ compiler *does* pays attention to whitespace is inside quoted text, such as `"Hello world!"`.

```
"Hello world!"
```

is different than

```
"Hello     world!"
```

and each prints out exactly as you'd expect. Newlines are not allowed in quoted text:

```
1  cout << "Hello
2       world!" << endl; // Not allowed!
```

Another exception where the C++ compiler pays attention to whitespace is with // comments. Single-line comments only last to the end of the line. Thus doing something like this will get you in trouble:

```
1  cout << "Hello world!" << endl; // Here is a single-line comment
2  this is not part of the comment
```

Basic formatting

Unlike some other languages, C++ does not enforce any kind of formatting restrictions on the programmer (remember, trust the programmer!). Many different methods of formatting C++ programs have been developed throughout the years, and you will find disagreement on which ones are best. Our basic rule of thumb is that the best styles are the ones that produce the most readable code, and provide the most consistency.

Here are our recommendations for basic formatting:

1) Your tab should be set to 4 spaces.

2) The braces that tell where a function begins and ends should be aligned with the function name, and be on their own lines:

```
1  int main()
2  {
3  }
```

3) Each statement within braces should start one tab in from the opening brace of the function it belongs to. For example:

```
1  int main()
2  {
3      cout << "Hello world!" << endl;
4      cout << "Nice to meet you." << endl;
5  }
```

4) Lines should not be too long. Typically, 72, 78, or 80 characters is the maximum length a line should be. If a line is going to be longer, it should be broken (at a reasonable spot) into multiple lines by indenting with a double tab.

```
1  int main()
2  {
```

```
3     cout << "This is a really, really, really, really, really, really,
4 really, " <<
5             "really long line" << endl;
6     cout << "This one is short" << endl;
  }
```

5) If a long line that is broken into pieces is broken with an operator (eg. << or +), the operator should be placed at the end of the line, not the start of the next line:

```
1 cout << "This is a really, really, really, really, really, really, really,
2 " <<
        "really long line" << endl;
```

Not

```
1 cout << "This is a really, really, really, really, really, really, really,
2 "
        << "really long line" << endl;
```

This makes it more obvious from looking at the first line that the next line is going to be a continuation.

6) Use whitespace to make your code easier to read.

Harder to read:

```
1 nCost = 57;
2 nPricePerItem = 24;
3 nValue = 5;
4 nNumberOfItems = 17;
```

Easier to read:

```
1 nCost          = 57;
2 nPricePerItem  = 24;
3 nValue         = 5;
4 nNumberOfItems = 17;
```

Harder to read:

```
1 cout << "Hello world!" << endl; // cout and endl live in the iostream
2 library
3 cout << "It is very nice to meet you!" << endl; // these comments make the
  code hard to read
  cout << "Yeah!" << endl; // especially when lines are different lengths
```

Easier to read:

```
1 cout << "Hello world!" << endl;                    // cout and endl live in
```

```
2 | the iostream library
3 | cout << "It is very nice to meet you!" << endl;  // these comments are
  | easier to read
  | cout << "Yeah!" << endl;                        // especially when all
  | lined up
```

Harder to read:

```
1 | // cout and endl live in the iostream library
2 | cout << "Hello world!" << endl;
3 | // these comments make the code hard to read
4 | cout << "It is very nice to meet you!" << endl;
5 | // especially when all bunched together
6 | cout << "Yeah!" << endl;
```

Easier to read:

```
1 | // cout and endl live in the iostream library
2 | cout << "Hello world!" << endl;
3 |
4 | // these comments are easier to read
5 | cout << "It is very nice to meet you!" << endl;
6 |
7 | // when separated by whitespace
8 | cout << "Yeah!" << endl;
```

We will follow these conventions throughout this tutorial, and they will become second nature to you. As we introduce new topics to you, we will introduce new style recommendations to go with those features.

Ultimately, C++ gives you the power to choose whichever style you are most comfortable with, or think is best. However, we highly recommend you utilize the same style that we use for our examples. It has been battle tested by thousands of programmers over billions of lines of code, and is optimized for success.

1.7 — Forward declarations

Take a look at this seemingly innocent sample program called add.cpp:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
7      return 0;
8  }
9
10 int add(int x, int y)
11 {
12     return x + y;
13 }
```

You would expect this program to produce the result:

```
The sum of 3 and 4 is: 7
```

But in fact, it doesn't compile at all! Visual Studio 2005 Express produces the following compile errors:

```
add.cpp(10) : error C3861: 'add': identifier not found
add.cpp(15) : error C2365: 'add' : redefinition; previous definition was
'formerly unknown identifier'
```

The reason this program doesn't compile is because the compiler reads files sequentially. When it reaches the function call to add() inside of main(), it doesn't know what add is, because we haven't defined add() until later! That produces the error on line 10. Then when it gets to the actual declaration of add(), it complains about add being redefined (which seems slightly misleading, given that it wasn't ever defined in the first place). Often times, a single error in your code will end up producing multiple warnings.

Rule: When addressing compile errors in your programs, always resolve the first error produced first.

In this case, that means we need to address the fact that the compiler doesn't know what add is. There are three ways to fix this problem.

The first way is to reorder our function calls so add is defined before main:

```
1  #include <iostream>
2
3  int add(int x, int y)
4  {
5      return x + y;
6  }
7
8  int main()
9  {
10     using namespace std;
11     cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
12     return 0;
13 }
```

That way, by the time main() calls add(), it will already know what add is. Because this is such a simple program, this change is relatively easy to do. However, in a large program, it would be extremely tedious trying to decipher which functions called which other functions so they could be declared in the correct order.

Furthermore, this option is not always available. Let's say we're writing a program that has two functions A and B. If function A calls function B, and function B calls function A, then there's no way to order the functions in a way that they will both be happy. If you define A first, the compiler will complain it doesn't know what B is. If you define B first, the compiler will complain that it doesn't know what A is.

Consequently, a better solution is to use a forward declaration. In a **forward declaration**, we declare (but do not define) our function in advance of where we use it, typically at the top of the file. This way, the compiler will understand what our function looks like when it encounters a call to it later. We do this by writing a declaration statement known as a function prototype. A **function prototype** is a declaration of a function that includes the function's name, parameters, and return type, but does not implement the function.

Here's our original program with a forward declaration:

```
1  #include <iostream>
2
3  int add(int x, int y); // forward declaration of add() using a function
   prototype
4
5  int main()
6  {
7      using namespace std;
8      cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
9      return 0;
10 }
11
12 int add(int x, int y)
   {
```

```
13      return x + y;
14  }
15
```

Now when the compiler reaches add() in main, it will know what add looks like (a function that takes two integer parameters and returns an integer), and it won't complain.

It is worth noting that function prototypes do not need to specify the names of the parameters. In the above code, you could also forward declare your function like this:

```
1int add(int, int);
```

However, we prefer the method where the parameters are named because it's more descriptive to a human reader.

One question many new programmers have is: what happens if we forward declare a function but do not define it?

The answer is: it depends. If a forward declaration is made, but the function is never called, the program will compile and run fine. However, if a forward declaration is made, the function is called, but the program never defines the function, the program will compile okay, but the linker will complain that it can't resolve the function call. Consider the following program:

```
1   #include "stdafx.h"
2   #include <iostream>
3
4   int add(int x, int y);
5
6   int main()
7   {
8       using namespace std;
9       cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
10      return 0;
11  }
```

In this program, we forward declare add(), and we call add(), but we never define add(). When we try and compile this program, Visual Studio 2005 Express produces the following message:

```
Compiling...
add.cpp
Linking...
add.obj : error LNK2001: unresolved external symbol "int __cdecl
add(int,int)" (?add@@YAHHH@Z)
add.exe : fatal error LNK1120: 1 unresolved externals
```

As you can see, the program compiled okay, but it failed at the link stage because int add(int, int) was never defined.

The third solution is to use a header file, which we will discuss shortly.

Quiz

1) What's the difference between a function prototype and a forward declaration?

2) Write the function prototype for this function:

```
1  int DoMath(int first, int second, int third, int fourth)
2  {
3       return first + second * third / fourth;
4  }
```

For each of the following programs, state whether they fail to compile, fail to link, or compile and link. If you are not sure, try compiling them!

3)

```
1  #include <iostream>
2  int add(int x, int y);
3
4  int main()
5  {
6      using namespace std;
7      cout << "3 + 4 + 5 = " << add(3, 4, 5) << endl;
8      return 0;
9  }
10
11 int add(int x, int y)
12 {
13     return x + y;
14 }
```

4)

```
1  #include <iostream>
2  int add(int x, int y);
3
4  int main()
5  {
6      using namespace std;
7      cout << "3 + 4 + 5 = " << add(3, 4, 5) << endl;
8      return 0;
9  }
10
11 int add(int x, int y, int z)
12 {
13     return x + y + z;
```

```
}
```

5)

```
#include <iostream>
int add(int x, int y);

int main()
{
    using namespace std;
    cout << "3 + 4 + 5 = " << add(3, 4) << endl;
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

6)

```
#include <iostream>
int add(int x, int y, int z);

int main()
{
    using namespace std;
    cout << "3 + 4 + 5 = " << add(3, 4, 5) << endl;
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Quiz **Answers**

1) Answer

A function prototype is declaration statement that tells the compiler what a function's return type is, what the name of the function is, and what the types of it's parameters are. A function prototype can be used to forward declare a function. A forward declaration is when something (such as a function or class) is declared in advance of where it is implemented.

2) Answer

```
// Either of these is correct.
// Do not forget the semicolon on the end, since these are statements.
int DoMath(int first, int second, int third, int fourth);
int DoMath(int, int, int, int);
```

3) Answer

Doesn't compile. The compiler will complain that the add() called in main() does not have the same number of parameters as the one that was forward declared.

4) Answer

Doesn't compile. The compiler will complain that the add() called in main() does not have the same number of parameters as the one that was forward declared.

5) Answer

Doesn't link. The compiler will match the forward declared prototype of add to the function call to add() in main(). However, no add() function that takes two parameters was ever implemented (we only implemented one that took 3 parameters), so the linker will complain.

6) Answer

Compiles and links. The function call to add() matches the prototype that was forward declared, the the implemented function also matches.

1.8 — Programs with multiple files

As programs get larger, it is not uncommon to split them into multiple files for organizational purposes. One advantage of working with an IDE is they make working with multiple files much easier. You already know how to create and compile single-file projects. Adding new files to existing projects is very easy.

In Visual Studio 2005 Express, right click on "Source Files" in the Solution Explorer window on the left, and choose Add -> New Item. Give the new file a name, and it will be added to your project.

In Code::Blocks, go to the file menu and choose "new file". Give the new file a name, and Code::Blocks will ask you if you want to add it to the active project. Click "Yes". Note that you will also have to click the "Release" and "Debug" checkboxes, to make sure it gets added to both versions.

Compile your project just the same as before. Couldn't be much easier!

Now, consider the following multiple-file program:

add.cpp:

```
1  int add(int x, int y)
2  {
3      return x + y;
4  }
```

main.cpp:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
7      return 0;
8  }
```

Try compiling this program for yourself. You will note that it doesn't compile, and it gives the same compiler error as the program in the previous lesson where the functions were declared in the wrong order:

```
add.cpp(10) : error C3861: 'add': identifier not found
```

```
add.cpp(15) : error C2365: 'add' : redefinition; previous definition was 'formerly unknown identifier'
```

When the compiler is compiling a code file, it does not know about the existence of functions that live in any other files. This is done so that files may have functions or variables that have the same names as those in other files without causing a conflict.

However, in this case, we want main.cpp to know about (and use) the add() function that lives in add.cpp. To give main.cpp access to the add function, we can use a forward declaration:

main.cpp with forward declaration:

```
1  #include <iostream>
2
3  int add(int x, int y); // forward declaration using function prototype
4
5  int main()
6  {
7      using namespace std;
8      cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
9      return 0;
10 }
```

Now, when the compiler is compiling main.cpp, it will know what add is. Using this method, we can give files access to functions that live in another file. However, as programs grow larger and larger, it becomes tedious to have to forward declare every function you use that lives in a different file. To solve that problem, the concept of header files was introduced. We discuss header files in the lesson on header files.

Try compiling add.cpp and the main.cpp with the forward declaration for yourself. We will begin working with multiple files a lot once we get into object-oriented programming, so now's as good a time as any to make sure you understand how to add and compile multiple file projects.

Code files (with a .cpp extension) are not the only files commonly seen in programs. The other type of file is called a **header file**, sometimes known as an **include file**. Header files almost always have a .h extension. The purpose of a header file is to hold declarations for other files to use.

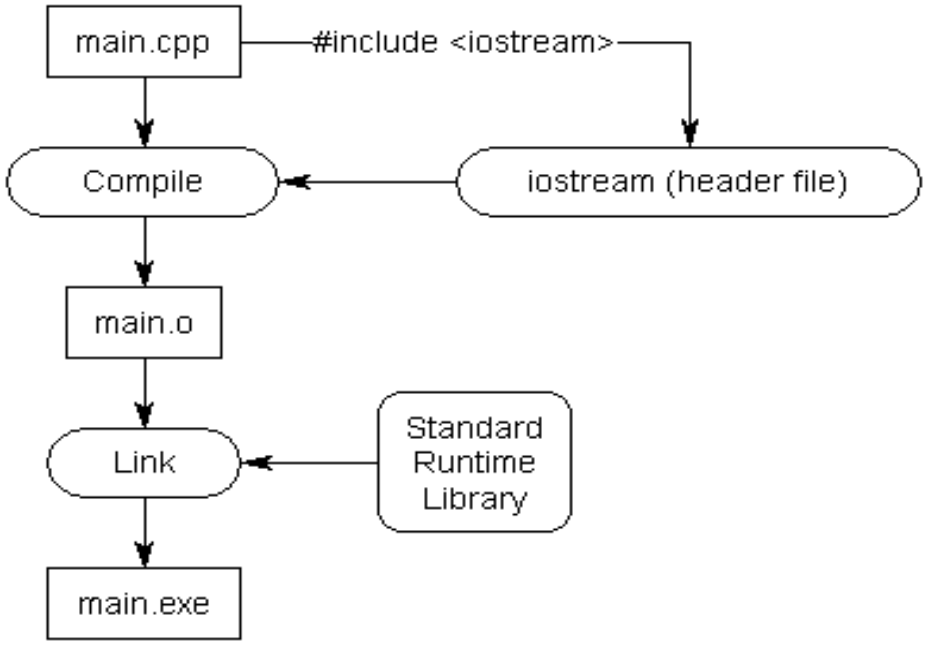
Using standard library header files

Consider the following program:

```
1  #include <iostream>
2  int main()
3  {
4      using namespace std;
5      cout << "Hello, world!" << endl;
6      return 0;
7  }
```

This program prints "Hello, world!" to the console using cout. However, our program never defines cout, so how does the compiler know what cout is? The answer is that cout has been declared in a header file called "iostream". When we use the line `#include <iostream>`, we are telling the compiler to locate and then read all the declarations from a header file named "iostream".

Keep in mind that header files typically only contain declarations. They do not define how something is implemented, and you already know that your program won't link if it can't find the implementation of something you use. So if cout is only *defined* in the "iostream" header file, where is it actually implemented? It is implemented in the runtime support library, which is automatically linked into your program during the link phase.



A **library** is a package of code that is meant to be reused in many programs. Typically, a library includes a header file that contains declarations for everything the library wishes to expose (make public) to users, and a precompiled object that contains all of the implementation code compiled into machine language. These libraries typically have a .lib or .dll extension on Windows, and a .a or .so extension on Unix. Why are libraries precompiled? First, since libraries rarely change, they do not need to be recompiled often, if ever. It would be a waste of time to compile them every time you wrote a program that used them. Second, because precompiled objects are in machine language, it prevents people from accessing or changing the source code, which is important to businesses or people who don't want to make their source code available for intellectual property reasons.

Writing your own header files

Now let's go back to the example we were discussing in the previous lesson. When we left off, we had two files, add.cpp and main.cpp, that looked like this:

add.cpp:

```
1  int add(int x, int y)
2  {
3      return x + y;
4  }
```

main.cpp:

```
1  #include <iostream>
2
3  int add(int x, int y); // forward declaration using function prototype
4
5  int main()
6  {
7      using namespace std;
8      cout << "The sum of 3 and 4 is " << add(3, 4) << endl;
9      return 0;
10 }
```

We'd used a forward declaration so that the compiler would know what add was when compiling main.cpp. As previously mentioned, writing forward declarations for every function you want to use that lives in another file can get tedious quickly.

Header files can relieve us of this burden. A header file only has to be written once, and it can be included in as many files as needed. This also helps with maintenance by minimizing the number of changes that need to be made if a function prototype ever changes (eg. by adding a new parameter).

Writing our own header files is surprisingly easy. Header files consist of two parts. The first part is called a **header guard**, which is discussed in the next lesson (on the preprocessor). The second part is the actual content of the .h file, which should be the declarations for all of the functions we want other files to be able to see. Our header files should all have a .h extension, so we'll call our new header file add.h:

add.h:

```
#ifndef ADD_H
#define ADD_H

int add(int x, int y); // function prototype for add.h

#endif

```

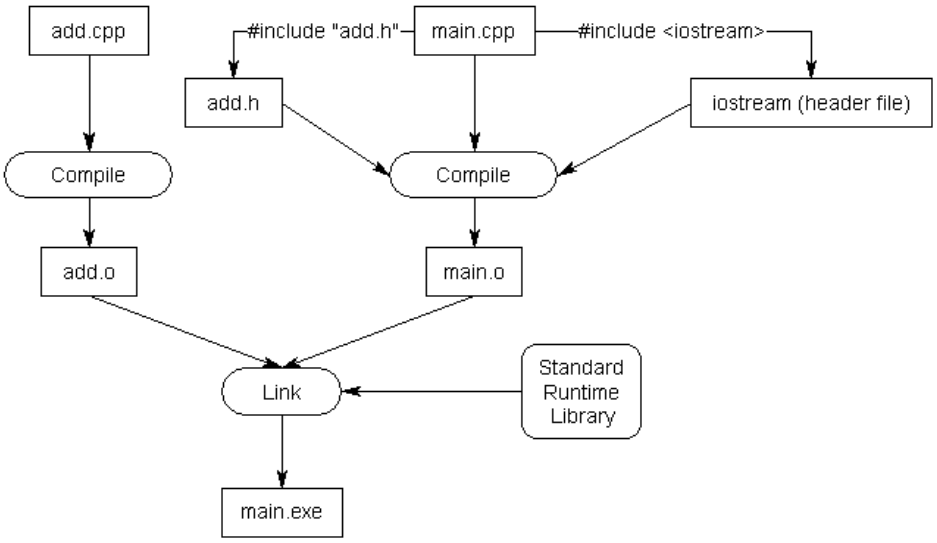
In order to use this header file in main.cpp, we have to include it. Here is the new main.cpp:

main.cpp that includes add.h:

```
#include <iostream>
#include "add.h" // this brings in the declaration for add()

int main()
{
    using namespace std;
    cout << "The sum of 3 and 4 is " << add(3, 4) << endl;
    return 0;
}

```

When the compiler compiles the `#include "add.h"` line, it copies the contents of add.h into the current file. Because our add.h contains a function prototype for add(), this prototype is now being used as a forward declaration of add()!

Consequently, our program will compile and link correctly.



You're probably curious why we use angled brackets for iostream, and double quotes for add.h. The answer is that angled brackets are used to tell the compiler that we are including a header file that was included with the compiler. The double-quotes tell the compiler that this is a header file we are supplying, which causes it to look for that header file in the current directory containing our source code first.

Rule: Use angled brackets to include header files that come with the compiler. Use double quotes to include any other header files.

Another commonly asked question is "why doesn't iostream have a .h extension?". The answer is, because iostream.h is a different header file than iostream is! To explain requires a very short history lesson.

When C++ was first created, all of the files in the standard runtime library ended in .h. Life was consistent, and it was good. The original version of cout and cin lived in iostream.h. When the language was standardized by the ANSI committee, they decided to move all of the functions in the runtime library into the std namespace (which is generally a good idea). However, this presented a problem: if they moved all the functions into the std namespace, none of the old programs would work any more!

To try to get around this issue and provide backwards compatibility for older programs, a new set of header files was introduced that use the same names but lack the .h extension. These new header files have all their functionality inside the std namespace. This way, older programs that include `#include <iosteam.h>` do not need to be rewritten, and newer programs can `#include <iostream>`.

Make sure when you include a header file from the standard library that you use the non .h version if it exists. Otherwise you will be using a deprecated version of the header that is no longer supported.

As a side note, many headers in the standard library do not have a non .h version, only a .h version. For these files, it is fine to include the .h version. Many of these libraries are backwards compatible with standard C programming, and C does not support namespaces. Consequently, the functionality of these libraries will not be accessed through the std namespace. Also, when you write your own header files, they will all have a .h extension, since you will not be putting your code in the std namespace.

Rule: use the non .h version of a library if it exists, and access the functionality through the std namespace. If the non .h version does not exist, or you are creating your own headers, use the .h version

Header file best practices

Here are a few best practices for creating your own header files.

- Always include header guards.

- Do not declare variables in header files unless they are constants. Header files should generally only be used for declarations.
- Do not define functions in header files unless they are trivial. Doing so makes your header files harder for humans to read.
- Each header file should have a specific job, and be as independent as possible. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h. That way if you only care about A later, you can just include A.h and not get any of the stuff related to B.
- Try to include as few other header files as possible in your header files.

1.10 — A first look at the preprocessor

The preprocessor is perhaps best thought of as a separate program that runs before the compiler when you compile your program. It's purpose is to process **directives**. Directives are specific instructions that start with a # symbol and end with a newline (NOT a semicolon). There are several different types of directives, which we will cover below. The preprocessor is not smart — it does not understand C++ syntax; rather, it manipulates text before the compiler gets to it.

Includes

You've already seen the #include directive in action. #Include tells the preprocessor to insert the contents of the included file into the current file at the point of the #include directive. This is useful when you have information that needs to be included in multiple places (as forward declarations often are).

The #include command has two forms: `#include <filename>` tells the compiler to look for the file in a special place defined by the operating system where header files for the runtime library are held. `#include "filename"` tells the compiler to look for the file in directory containing the source file doing the #include. If that fails, it will act identically to the angled brackets case.

Macro defines

Macro defines take the form:

```
#define identifier replacement
```

Whenever the preprocessor encounters this directive, any further occurrence of 'identifier' is replaced by 'replacement'. The identifier is traditionally typed in all capital letters, using underscores to represent spaces.

For example, consider the following snippet:

```
1  #define MY_NAME  "Alex"
2
3  cout << "Hello, " << MY_NAME << endl;
```

The preprocessor converts this into the following:

```
1  #define MY_NAME  "Alex"
2
3  cout << "Hello, " << "Alex" << endl;
```

Which, when run, prints the output `Hello, Alex`.

#defines used in this manner have two important properties. First, they allow you to give a descriptive name to something, such as a number.

For example, consider the following snippet:

```
1  int nYen = nDollars * 122;
```

A number such as the 122 in the program above is called a magic number. A **magic number** is a hard-coded number in the middle of the code that does not have any context — what does 122 mean? Is it a conversion rate? Is it something else? It's not really clear. In more complex programs, it is often impossible to tell what a hard-coded number represents.

This snippet is clearer:

```
1  #define YEN_PER_DOLLAR  122
2  int nYen = nDollars * YEN_PER_DOLLAR;
```

Second, #defined numbers can make programs easier to change. Assume the exchange rate changed from 122 to 123, and our program needed to reflect that. Consider the following program:

```
1  int nYen1 = nDollars1 * 122;
2  int nYen2 = nDollars2 * 122;
3  int nYen3 = nDollars3 * 122;
4  int nYen4 = nDollars4 * 122;
5  SetWidthTo(122);
```

To update our program to use the new exchange rate, we'd have to update the first four statements from 122 to 123. But what about the 5th statement? Does that 122 have the same meaning as the other 122s? If so, it should be updated. If not, it should be left alone, or we might break our program somewhere else.

Now consider the same program with #defined values:

```
1  #define YEN_PER_DOLLAR  122
2  #define COLUMNS_PER_PAGE  122
3
4  int nYen1 = nDollars1 * YEN_PER_DOLLAR;
5  int nYen2 = nDollars2 * YEN_PER_DOLLAR;
6  int nYen3 = nDollars3 * YEN_PER_DOLLAR;
7  int nYen4 = nDollars4 * YEN_PER_DOLLAR;
8  SetWidthTo(COLUMNS_PER_PAGE);
```

To change the exchange rate, all we have to do is make one change:

```
1  #define YEN_PER_DOLLAR  123
2  #define COLUMNS_PER_PAGE  122
3
4  int nYen1 = nDollars1 * YEN_PER_DOLLAR;
5  int nYen2 = nDollars2 * YEN_PER_DOLLAR;
6  int nYen3 = nDollars3 * YEN_PER_DOLLAR;
7  int nYen4 = nDollars4 * YEN_PER_DOLLAR;
   SetWidthTo(COLUMNS_PER_PAGE);
```

Now we've updated our yen conversions, and don't have to worry about inadvertently changing the number of columns per page.

While #define values are a huge improvement over magic numbers, they have some issues of their own, which we will talk more about later when we get into variables and scoping issues.

You can undefine a previously defined value by using the #undef preprocessor directive.

Consider the following snippet:

```
1  #define MY_NAME "Alex"
2  cout << "My name is " << MY_NAME << endl;
3  #undef MY_NAME
3  cout << "My name is " << MY_NAME << endl;
4
```

The last line of the program causes a compile error because MY_NAME has been undefined.

Conditional compilation

The conditional compilation preprocessor directives allow you to specify under what conditions something will or won't compile. The only conditional compilation directives we are going to cover in this section are #ifdef, #ifndef, and #endif.

The #ifdef preprocessor directive allow the preprocessor to check whether a value has been previously #defined. If so, the code between the #ifdef and corresponding #endif is compiled. If not, the code is ignored.

Consider the following snippet of code:

```
1  #define PRINT_JOE
2
3  #ifdef PRINT_JOE
4  cout << "Joe" << endl;
5  #endif
6
7  #ifdef PRINT_BOB
8  cout << "Bob" << endl;
9  #endif
```

Because PRINT_JOE has been #defined, the line `cout << "Joe" << endl;` will be compiled. Because PRINT_BOB has not been #defined, the line `cout << "Bob" << endl;` will not be compiled.

#ifndef is the opposite of #ifdef, in that it allows you to check whether a name has NOT been defined yet.

```
1 #ifndef PRINT_BOB
2 cout << "Bob" << endl;
3 #endif
```

This program prints "Bob", because PRINT_BOB was never #defined.

Header guards

Because header files can include other header files, it is possible to end up in the situation where a header file gets included multiple times. For example, consider the following program:

add.h:

```
1 #include "mymath.h"
2 int add(int x, int y);
```

subtract.h:

```
1 #include "mymath.h"
2 int subtract(int x, int y);
```

main.cpp:

```
1 #include "add.h"
2 #include "subtract.h"
```

When we include add.h, it brings in both mymath.h and the prototype for add(). When we include subtract.h, it brings in both mymath.h (again) and the prototype for subtract(). Consequently, all the contents of mymath.h will have been included twice, which will cause the compiler to complain.

To prevent this from happening, we use header guards, which are conditional compilation directives that take the following form:

```
1 #ifndef SOME_UNIQUE_NAME_HERE
2 #define SOME_UNIQUE_NAME_HERE
3
4 // your declarations here
5
  #endif
```

When this header is included, the first thing it does is check whether SOME_UNIQUE_NAME_HERE has been previous defined. If this is the first time we've included the header, SOME_UNIQUE_NAME_HERE will not have been defined. Consequently, it #defines SOME_UNIQUE_NAME_HERE and includes the contents of the file. If this is not the first time we've included the header, SOME_UNIQUE_NAME_HERE will already have been defined from the first time the contents of the header were included. Consequently, the entire header will be ignored.

All of your header files should have header guards on them. SOME_UNIQUE_NAME_HERE can be any name you want, but typically the name of the header file with a _H appended to it is used. For example, add.h would have the header guard:

```
1  #ifndef ADD_H
2  #define ADD_H
3
4  // your declarations here
5
6  #endif
```

Even the standard library includes use header guards. If you were to take a look at the iostream header file from Visual Studio 2005 Express, you would see:

```
1  #ifndef _IOSTREAM_
2  #define _IOSTREAM_
3
4  // content here
5
6  #endif
```

1.10a — How to design your first programs

Now that you've learned some basics about programs, let's look more closely at how to design a program. When you sit down to write a program, generally you have some sort of problem that you'd like to solve, or situation that you'd like to simulate. New programmers often have trouble figuring out how to convert that idea into actual code. But it turns out, you have many of the problem solving skills you need already, acquired from every day life.

The most important thing to remember (and hardest thing to do) is to design your program *before you start coding*. In many regards, programming is like architecture. What would happen if you tried to build a house without following an architectural plan? Odds are, unless you were very talented, you'd end up with a house that had a lot of problems: leaky roofs, walls that weren't straight, etc… Similarly, if you try to program before you have a good gameplan moving forward, you'll likely find that your code has a lot of problems, and you'll have to spend a lot of time fixing problems that could have been avoided altogether with a little design.

A little up-front planning will save you both time and frustration in the long run.

Step 1: Define the problem

The first thing you need to figure out is what problem your program is attempting to solve. Ideally, you should be able to state this in a sentence or two. For example:

- I want to write a phone book application to help me keep track of my friend's phone numbers.
- I want to write a random dungeon generator that will produce interesting looking caverns.
- I want to write a program that will take information about stocks and attempt to predict which ones I should buy.

Although this step seems obvious, it's also highly important. The worst thing you can do is write a program that doesn't actually do what you (or your boss) wanted!

Step 2: Define your targets

When you are an experienced programmer, there are many other steps that typically would take place at this point, including:

- Understanding who your target user is
- Defining what target architecture and/or OS your program will run on
- Determining what set of tools you will be using
- Determining whether you will write your program alone or as part of a team
- Collecting requirements (a documented list of what the program should do)

However, as a new programmer, the answers to these questions are typically simple: You are writing a program for your own use, alone, on your own system, using an IDE you purchased or downloaded. This makes things easy, so we won't spend any time on this step.

Step 3: Make a heirarchy of tasks

In real life, we often need to perform tasks that are very complex. Trying to figure out how to do these tasks can be very challenging. In such cases, we often make use of the **top down** method of problem solving. That is, instead of solving a single complex task, we break that task into multiple subtasks, each of which is individually easier to solve. If those subtasks are still too difficult to solve, they can be broken down further. By continuously splitting complex tasks into simpler ones, you can eventually get to a point where each individual task is manageable, if not trivial.

Let's take a look at an example of this. Let's say we want to write a report on carrots. Our task hierarchy currently looks like this:

- Write report on carrots

Writing a report on carrots is a pretty big task to do in one sitting, so let's break it into subtasks:

- Write report on carrots
    - Do research on carrots
    - Write outline
    - Fill in outline with details about carrots

That's a more managable, as we now have three tasks that we can focus on individually. However, in this case, "Do research on carrots is somewhat vague", so we can break it down further:

- Write report on carrots
    - Do research on carrots
        - Go to library and get book on carrots
        - Look for information about carrots on internet
    - Write outline
        - Information about growing
        - Information about processing
        - Information about nutrition
    - Fill in outline with details about carrots

Now we have a hierarchy of tasks, none of them particularly hard. By completing each of these relatively manageable sub-items, we can complete the more difficult overall task of writing a report on carrots.

The other way to create a hierarchy of tasks is to do so from the **bottom up**. In this method, we'll start from a list of easy tasks, and construct the hierarchy by grouping them.

As an example, many people have to go to work or school on weekdays, so let's say we want to solve the problem of "get from bed to work". If you were asked what tasks you did in the morning to get from bed to work, you might come up with the following list:

- Pick out clothes
- Get dressed
- Eat breakfast
- Drive to work
- Brush your teeth
- Get out of bed
- Prepare breakfast
- Get in your car
- Take a shower

Using the bottom up method, we can organize these into a hierarchy of items by looking for ways to group items with similarities together:

- Get from bed to work
    - Bedroom things
        - Get out of bed
        - Pick out clothes
    - Bathroom things
        - Take a shower
        - Brush your teeth
    - Breakfast things
        - Prepare breakfast
        - Eat breakfast
    - Transportation things
        - Get in your car
        - Drive to work

As it turns out, these task hierarchies are extremely useful in programming, because once you have a task hierarchy, you have essentially defined the structure of your overall program. The top level task (in this case, "Write a report on carrots" or "Get from bed to work") becomes main() (because it is the main item you are trying to solve). The subitems become functions in the program.

If it turns out that one of the items (functions) is too difficult to implement, simply split that item into multiple subitems, and have that function call multiple subfunctions that implement those new tasks. Eventually you should reach a point where each function in your program is trivial to implement.

Step 4: Figure out the sequence of events

Now that your program has a structure, it's time to determine how to link all the tasks together. The first step is to determine the sequence of events that will be performed. For example, when you get up in the morning, what order do you do the above tasks? It might look like this:

- Get out of bed
- Pick out clothes
- Take a shower
- Get dressed
- Prepare breakfast
- Eat breakfast
- Brush your teeth
- Get in your car
- Drive to work

If we were writing a calculator, we might do things in this order:

- Get first number from user
- Get mathematical operation from user
- Get second number from user
- Calculate result
- Print result

This list essentially defines what will go into your main() function:

```
1  int main()
2  {
3      GetOutOfBed();
4      PickOutClothes();
5      TakeAShower();
6      GetDressed();
7      PrepareBreakfast();
8      EatBreakfast();
9      BrushTeeth();
10     GetInCar();
11     DriveToWork();
12 }
```

Or in the case of the calculator:

```
1  int main()
2  {
3      // Get first number from user
4      GetUserInput();
5
6      // Get mathematical operation from user
7      GetMathematicalOperation();
8
9      // Get second number from user
10     GetUserInput();
11
```

```
10      // Calculate result
11      CalculateResult();
12
13      // Print result
14      PrintResult();
    }
```

Step 5: Figure out the data inputs and outputs for each task

Once you have a hierarchy and a sequence of events, the next thing to do is figure out what input data each task needs to operate, and what data it produces (if any). If you already have the input data from a previous step, that input data will become a parameter. If you are calculating output for use by some other function, that output will generally become a return value.

When we are done, we should have prototypes for each function. In case you've forgotten, a **function prototype** is a declaration of a function that includes the function's name, parameters, and return type, but does not implement the function.

Let's do a couple examples. GetUserInput() is pretty straightforward. We're going to get a number from the user and return it back to the caller. Thus, the function prototype would look like this:

```
1   int GetUserInput()
```

In the calculator example, the CalculateResult() function will need to take 3 pieces of input: Two numbers and a mathematical operator. We should already have all three of these by the time we get to the point where this function is called, so these three pieces of data will be function parameters. The CalculateResult() function will calculate the result value, but it does not display the result itself. Consequently, we need to return that result as a return value so that other functions can use it.

Given that, we could write the function prototype like this:

```
1   int CalculateResult(int nInput1, char chOperator, int nInput2);
```

Step 6: Write the task details

In this step, for each task, you will write it's actual implementation. If you have broken the tasks down into small enough pieces, each task should be fairly simple and straightforward. If a given task still seems overly-complex, perhaps it needs to be broken down into subtasks that can be more easily implemented.

For example:

```
1   char GetMathematicalOperation()
2   {
3       cout << "Please enter an operator (+,-,*,or /): ";
```

```
4
5       char chOperation;
6       cin >> chOperation;
7
8       // What if the user enters an invalid character?
9       // We'll ignore this possibility for now
        return chOperation;
    }
```

Step 7: Connect the data inputs and outputs

Finally, the last step is to connect up the inputs and outputs of each task in whatever way is appropriate. For example, you might send the output of CalculateResult() into an input of PrintResult(), so it can print the calculated answer. This will often involve the use of intermediary variables to temporary store the result so it can be passed between functions. For example:

```
1   // nResult is a temporary value used to transfer the output of
2   CalculateResult()
3   // into an input of PrintResult()
4   int nResult = CalculateResult(nInput1, chOperator, nInput2);
    PrintResult(nResult);
```

This tends to be much more readable than the alternative condensed version that doesn't use a temporary variable:

```
1   PrintResult( CalculateResult(nInput1, chOperator, nInput2) );
```

This is often the hardest step for new programmers to get the hang of.

Note: To fully complete this example, you'll need to utilize if statements, which we won't cover for a while, but you're welcome to take a sneak-peak at now.

A fully completed version of the above calculator sample follows (hidden in case you want to take a stab at it yourself first):

Words of advice when writing programs

Keep your program simple to start. Often new programmers have a grand vision for all the things they want their program to do. "I want to write a role-playing game with graphics and sound and random monsters and dungeons, with a town you can visit to sell the items that you find in the dungeon" If you try to write something too complex to start, you will become overwhelmed and discouraged at your lack of progress. Instead, make your first goal as simple as possible, something that is definitely within your reach. For example, "I want to be able to display a 2d representation of the world on the screen".

Add features over time. Once you have your simple program working and working well, then you can add features to it. For example, once you can display your 2d world, add a character who can walk around. Once you can walk around, add walls that can impede your progress. Once you have walls, build a simple town out of them. Once you have a town, add merchants. By adding each feature incrementally your program will get progressively more complex without overwhelming you in the process.

Focus on one area at a time. Don't try to code everything at once, and don't divide your attention across multiple tasks. Focus on one task at a time, and see it through to completion as much as is possible. It is much better to have one fully working task and five that haven't been started yet than six partially-working tasks. If you split your attention, you are more likely to make mistakes and forget important details.

Test each piece of code as you go. New programmers will often write the entire program in one pass. Then when they compile it for the first time, the compiler reports hundreds of errors. This can not only be intimidating, if your code doesn't work, it may be hard to figure out why. Instead, write a piece of code, and then compile and test it immediately. If it doesn't work, you'll know exactly where the problem is, and it will be easy to fix. Once you are sure that the code works, move to the next piece and repeat. It may take longer to finish writing your code, but when you are done the whole thing should work, and you won't have to spend twice as long trying to figure out why it doesn't.

Most new programmers will shortcut many of these steps and suggestions (because it seems like a lot of work and/or it's not as much fun as writing the code). However, for any non-trivial project, following these steps will definitely save you a lot of time in the long run. A little planning up front saves a lot of debugging at the end.

The good news is that once you become comfortable with all of these concepts, they will start coming naturally to you without even thinking about it. Eventually you will get to the point where you can write entire functions without any pre-planning at all.

1) Write a program that reads two numbers from the user, adds them together, and then outputs the answer. The program should use two functions: A function named ReadNumber() should be used to get an integer from the user, and a function named "WriteAnswer" should be used to output the answer. You do not need to write a function to do the adding.

2) Modify the program you wrote in #1 so that the function to read a number from the user and the function to output the answer are in a separate file called "io.cpp". Use a forward declaration to access them from your main() function, which should live in "main.cpp".

3) Modify the program you wrote in #2 so that it uses a header file to access the functions instead of forward declarations. Make sure your header file uses header guards.

Quiz solutions

1) Answer

```
1  #include <iostream>
2  int ReadNumber()
3  {
4      using namespace std;
5      cout << "Enter a number: ";
6      int x;
7      cin >> x;
8      return x;
9  }
10
11 void WriteAnswer(int x)
12 {
13     using namespace std;
14     cout << "The answer is " << x << endl;
15 }
16
17 int main()
18 {
19     int x = ReadNumber();
20     int y = ReadNumber();
21     WriteAnswer(x+y);
22     return 0;
23 }
```

2) Answer

io.cpp:

```
1  #include <iostream>
2  int ReadNumber()
3  {
4      using namespace std;
5      cout << "Enter a number: ";
6      int x;
7      cin >> x;
```

```
7       return x;
8   }
9
10  void WriteAnswer(int x)
11  {
12      using namespace std;
        cout << "The answer is " << x << endl;
    }
```

main.cpp:

```
1   int ReadNumber();
2   void WriteAnswer(int x);
3
4   int main()
5   {
6       int x = ReadNumber();
7       int y = ReadNumber();
8       WriteAnswer(x+y);
9       return 0;
10  }
```

3) Answer

io.cpp:

```
1   #include <iostream>
2   int ReadNumber()
3   {
4       using namespace std;
5       cout << "Enter a number: ";
6       int x;
7       cin >> x;
8       return x;
9   }
10
11  void WriteAnswer(int x)
12  {
13      using namespace std;
        cout << "The answer is " << x << endl;
    }
```

io.h:

```
1   #ifndef IO_H
2   #define IO_H
3
4   int ReadNumber();
5   void WriteAnswer(int x);
```

```
6  #endif
```

main.cpp:

```
1  #include "io.h"
2
3  int main()
4  {
5      int x = ReadNumber();
6      int y = ReadNumber();
7      WriteAnswer(x+y);
       return 0;
   }
```

Variables Part I

2.1 — Basic addressing and variable declaration

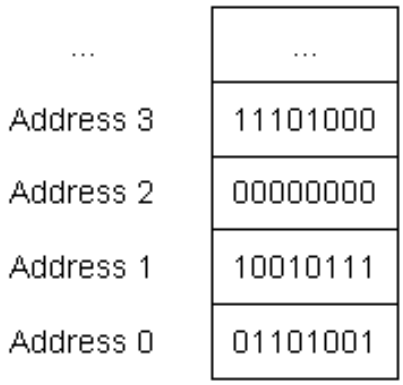
Addressing memory

This lesson builds directly on the material in the section "A first look at variables".

In the previous lesson on variables, we talked about the fact that variables are names for a piece of memory that can be used to store information. To recap briefly, computers have random access memory (RAM) that is available for program to use. When a variable is declared, a piece of that memory is set aside for that variable.

The smallest unit of memory is a binary digit (bit), which can hold a value of 0 or 1. You can think of a bit as being like a traditional light switch — either the light is off (0), or it is on (1). There is no in-between. If you were to look at a sequential piece of memory, all you would see is …011010100101010… or some combination thereof. Memory is organized into individual sections called **addresses**. Perhaps surprisingly, in modern computers, each bit does not get it's own address. The smallest addressable unit of memory is a group of 8 bits known as a **byte**.

The following picture shows some sequential memory addresses, along with the corresponding byte of data:



Because all data on a computer is just a sequence of bits, we use a **data type** to tell us how to interpret the contents of memory in some meaningful way. You have already seen one example of a data type: the integer. When we declare a variable as an integer, we are telling the computer "the piece of memory that this variable addresses is going to be interpreted as a whole number".

When you assign a value to a data type, the computer takes care of the details of encoding your value into the appropriate sequence of bits for that data type. When you ask for your value back, the program "reconstitutes" your number from the sequence of bits in memory.

There are many other data types in C++ besides the integer, most of which we will cover shortly. As shorthand, we typically refer to a variable's "data type" as it's "type".

Declaring a variable

In the "basic C++" section, you already learned how to declare an integer variable:

```
1  int nVarName; // int is the type, nVarName is the name of the variable
```

To declare variables of other data types, the idea is exactly the same:

```
1  type varName; // type is the type (eg. int), varName is the name of the
   variable
```

In the following example, we declare 5 different variables of 5 different types.

```
1  bool bValue;
2  char chValue;
3  int nValue;
4  float fValue;
   double dValue;
```

It's that simple. (Well, almost — there are a few things you can't name your variables, which we'll talk about in the next section)

You can also assign values to your variables upon declaration. When we assign values to a variable using the assignment operator (equals sign), it's called an **explicit assignment**:

```
1  int nValue = 5; // explicit assignment
```

You can also assign values to variables using an **implicit assignment**:

```
1  int nValue(5); // implicit assignment
```

Even though implicit assignments look a lot like function calls, the compiler keeps track of which names are variables and which are functions so that they can be resolved properly.

Declaring multiple variables

It is possible to declare multiple variables *of the same type* in one statement by separating the names with a comma. The following 2 snippets of code are effectively the same:

```
1  int nValue1, nValue2;
```

```
1  int nValue1;
2  int nValue2;
```

You can also assign them values on the declaration line:

```
1  int nValue1 = 5, nValue2 = 6;
2  int nValue3(7), nValue4(8);
```

Which is effectively the same as:

```
1  int nValue1 = 5;
2  int nValue2 = 6;
3  int nValue3 = 7;
   int nValue4 = 8;
```

There are three mistakes that new programmers tend to make when declaring multiple variables in the same statement.

The first mistake is declaring each variable as int (or whatever type it is) in sequence. This is not a bad mistake because the compiler will complain and ask you to fix it.

```
1  int nValue1, int nValue2; // wrong (compiler error)
2
3  int nValue1, nValue2; // correct
```

The second error is to try to declare two variables of different types on the same line, which is not allowed. Variables of different types must be declared in separate statements. This is also not a bad mistake because the compiler will complain and ask you to fix it.

```
1  int nValue1, double dValue2; // wrong (compiler error)
2
3  int nValue1; double dValue2; // correct (but not recommended)
4
5  // correct and recommended (easier to read)
   int nValue1;
6  double dValue2;
```

The last mistake is the dangerous case. In this case, the programmer mistakenly tries to initialize both variables by using one assignment statement:

```
1  int nValue1, nValue2 = 5; // wrong (nValue1 is uninitialized!)
2
3  int nValue1 = 5, nValue2 = 5; // correct
```

In the top statement, the nValue1 variable will be left uninitialized, and the compiler will NOT complain. This is a great way to have your program intermittently crash and produce sporadic results.
The best way to remember that this is wrong is consider the case of implicit initialization:

```
1  int nValue1, nValue2(5);
```

This makes it seem a little more clear that the value 5 is only being assigned to nValue2. The explicit assignment case is no different.

Where to declare variables

Older C compilers forced users to declare all of the variables in a function at the top of the function:

```
1  int main()
2  {
3      // all variable up top
4      int x;
5      int y;
6
7      // then code
8      using namespace std;
9      cout << "Enter a number: ";
10     cin >> x;
11
12     cout << "Enter another number: ";
13     cin >> y;
14
15     cout << "The sum is: " << x + y << endl;
16     return 0;
17 }
```

This style is now obsolete. C++ compilers do not require all variables to be declared at the top of a function. The proper C++ style is to declare variables when and where they are needed:

```
1  int main()
2  {
3      // then code
4      using namespace std;
5
6      cout << "Enter a number: ";
7      int x; // we need x starting here.
8      cin >> x;
9
10     cout << "Enter another number: ";
11     int y; // we don't need y until now
12     cin >> y;
13
14     cout << "The sum is: " << x + y << endl;
15     return 0;
16 }
```

This has quite a few advantages. First, variables that are declared only when needed are given context by the statements around them. If x were declared at the top of the function, we would have no idea what it was used for until we scanned the function and found where it was used. Declaring x amongst a bunch of input/output statements helps make it obvious that this variable is being used for input and/or output.

Second, declaring a variable only where it is needed tells us that this variable does not affect anything above it, making our program easier to understand and requiring less scrolling. Finally, it reduces the likelihood of inadvertently leaving a variable uninitialized, because we can declare and then immediately initialize it with the value we want it to have.

Rule: Declare variables where they are needed.

2.2 — Keywords and naming identifiers

Keywords

C++ reserves a set of 63 words for it's own use. These words are called **keywords**, and each of these keywords has a special meaning with in the C++ language. The 15 keywords that are starred (*) were added to the language after it's initial release, consequently, some older reference books or material may omit these.

Here is a list of all the C++ keywords:

asm auto bool * break case catch char class	const const_cast * continue default delete do double dynamic_cast * 	else enum explicit * export * extern false * float for	friend goto if inline int long mutable * namespace * 	new operator private protected public register reinterpret_cast * return	short signed sizeof static static_cast * struct switch template	this throw true * try typedef typeid * typename * union	unsigned using * virtual void volatile wchar_t * while

You have already run across some of these keywords, including *int*, *void*, *return*, *using*, and *namespace*. Along with a set of operators, these keywords define the entire language of C++ (preprocessor commands excluded). Because these keywords have special meaning, your IDEs will change the text color of these words (usually to blue) to make them more visible.

By the time you are done with this tutorial, you will understand what almost all of these words do!

Identifiers, and naming them

The name of a variable, function, class, or other entity in C++ is called an **identifier**. C++ gives you a lot of flexibility to name identifiers as you wish. However, there are a few rules that must be followed when naming identifiers:

- The identifier can not be a keyword. Keywords are reserved.
- The identifier can only be composed of letters, numbers, and the underscore character. That means the name can not contains symbols (except the underscore) nor whitespace.
- The identifier must begin with a letter or an underscore. It can not start with a number.
- C++ distinguishes between lower and upper case letters. `nvalue` is different than `nValue` is different than `NVALUE`.

Not too difficult, eh?

Now that you know how you *can* name a variable, let's talk about how you *should* name a variable.

First, it is a convention that variable (and function) names begin with a lower case letter. If the variable or function name is one word, the whole thing should be written in lower case letters.

```
1  int value; // correct
2
3  int Value; // incorrect (should start with lower case letter)
4  int VALUE; // incorrect (should start with lower case letter)
5  int VaLuE; // incorrect (see your psychiatrist) ;)
```

If the identifier is a multiword name, there are two common conventions: separated by underscores, or intercapped.

```
1  int my_variable_name; // correct (separated by underscores)
2  int myVariableName; // correct (intercapped)
3
4  int my variable name; // incorrect (spaces not allowed)
5  int MyVariableName; // incorrect (should start with lower case letter)
```

In this tutorial, we will use the intercapped approach because it's easier to read (it's easy to mistake an underscore for a space in dense blocks of code).

Second, and this is perhaps the most important rule of all, give your identifiers names that actually describe what they are. It is typical for inexperienced programmers to make variable names as short as possible, either to save on typing or because they figure the meaning is obvious. This is almost always a mistake. Ideally, variables should be named in a way that would help someone who has no idea what your code does be able to figure it out as quickly as possible. In 3 months, when you look at your program again, you'll have forgotten how it works, and you'll thank yourself for picking variable names that make sense. The more complex the code the variable is being used in, the better name it needs to have.

int ccount	Bad	Nobody knows what a ccount is
int customerCount	Good	Clear what we're counting
int i	Bad	What does I stand for?*
int index	Good	This variable is indexing something
int _count	Bad	Do not start variable names with underscore
int count	Either	Okay only if obvious what we're counting
int data	Bad	What kind of data?

int value1, value2	Either	Can be hard to differentiate between the two
int numberOfApples	Good	Descriptive
int totalScore	Good	Descriptive
int monstersKilled	Good	Descriptive
int x, y	Either	Okay only in trivial mathematical functions

* Note: it is okay to use trivial variable names for variables that have a trivial use, such as loop variables. We will address this topic in the section on flow control.

Third, a clarifying comment can go a long way. For example, say we've declared a variable named *numberOfChars* that is supposed to store the number of characters in a piece of text. Does the text "Hello World!" have 10, 11, or 12 characters? It depends on whether we're including whitespace or punctuation. Rather than naming the variable *numberOfCharsIncludingWhitespaceAndPunctuation*, which is rather lengthy, a well placed comment on the declaration line should help the user figure it out:

```
1 // holds number of chars in a piece of text -- including whitespace and
2 punctuation!
  int numberOfChars;
```

Hungarian notation

One method that is sometimes used in naming variables is called Hungarian notation. In Hungarian notation, variables always begin with a prefix that indicates the variable's type.

```
1 int nValue; // the n before Value represents that this is an integer
2 bool bValue; // b means boolean
3 char chValue; // ch means char
4 double dValue; // d means double
  float fValue; // f means float
```

We will talk more about Hungarian notation later in this section once we've covered the different numerical data types.

Quiz

Pick which variables are improperly declared and named according to standard conventions (ie. how you *should* name a variable).

1) int sum;
2) int _apples;
3) int cats=5, dogs=5;
4) int VALUE;
5) int clouds=5, int trees;

6) int my variable name;
7) int length=3; int width=4;
8) int TotalCustomers;
9) int void = 5;
10) int nAngle;
11) int 3some;
12) int meters_of_pipe;
13) int length, width=5;

Answer

2) Variable names should not start with an underscore.
4) Variable names should start with a lower case letter.
5) Compiler will complain about the “int” before trees.
6) Variable names can not contain spaces.
8) Variable names should start with a lower case letter.
9) Void is a keyword.
11) Variable names can not start with a number.
13) Length is uninitialized.

2.3 — Variable sizes and the sizeof operator

As you learned in the lesson on basic addressing, memory on modern machines is typically organized into byte-sized pieces, with each piece having a unique address. Up to this point, it has been useful to think of memory as a bunch of cubbyholes or mailboxes where we can put and retrieve information, and variables as names for accessing those cubbyholes or mailboxes.

However, this analogy is not quite correct in one regard — most variables actually take up more than 1 byte of memory. Consequently, a single variable may use 2, 4, or even 8 consecutive memory addresses. The amount of memory that a variable uses is based on it's data type. Fortunately, because we typically access memory through variable names and not memory addresses, the compiler is largely able to hide the details of working with different sized variables from us.
There are several reasons it is useful to know how much memory a variable takes up.

First, the more memory a variable takes up, the more information it can hold. Because each bit can only hold a 0 or a 1, we say that bit can store 2 values. 2 bits can store 4 different values:

bit 0	bit 1
0	0
0	1
1	0
1	1

3 bits can store 8 values. n bits can store 2^n values. Because a byte is 8 bits, a byte can store 2^8 (256) values.

The size of the variable puts a limit on the amount of information it can store — variables that are bigger can hold larger numbers. We will address this issue further when we get into the different types of variables.

Second, computers have a finite amount of free memory. Every time we declare a variable, a small portion of that free memory is used as long as the variable is in existence. Because modern computers have a lot of memory, this often isn't a problem, especially if only declaring a few variables. However, for programs that need a large amount of variables (eg. 100,000), the difference between using 1 byte and 8 byte variables can be significant.

The obvious next question is "how much memory do variables of different data types take?". The size of a given data type is dependent on the compiler and/or the computer architecture. On most 32-bit machines (as of this writing), a char is 1 byte, a bool is 1 byte, a short is 2 bytes, an int is 4 bytes, a long is 4 bytes, a float is 4 bytes, and a double is 8 bytes.

In order to determine the size of data types on a particular machine, C++ provides an operator named sizeof. The **sizeof operator** is a unary operator that takes either a type or a variable, and returns its size in bytes. You can compile and run the following program to find out how large your data types are:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "bool:\t\t" << sizeof(bool) << " bytes" << endl;
7      cout << "char:\t\t" << sizeof(char) << " bytes" << endl;
8      cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes" << endl;
9      cout << "short:\t\t" << sizeof(short) << " bytes" << endl;
10     cout << "int:\t\t" << sizeof(int) << " bytes" << endl;
11     cout << "long:\t\t" << sizeof(long) << " bytes" << endl;
12     cout << "float:\t\t" << sizeof(float) << " bytes" << endl;
13     cout << "double:\t\t" << sizeof(double) << " bytes" << endl;
       cout << "long double:\t" << sizeof(long double) << " bytes" << endl;
       return 0;
   }
```

Here is the output from the author's Pentium 4 machine, using Visual Studio 2005 Express:

```
bool:           1 bytes
char:           1 bytes
wchar_t:        2 bytes
short:          2 bytes
int:            4 bytes
long:           4 bytes
float:          4 bytes
double:         8 bytes
long double:    8 bytes
```

Your results may vary if you are using a different type of machine, or a different compiler.

If you're wondering what \t is in the above program, it's a special symbol that inserts a tab. We will cover \t and other special symbols when we talk about the char data type.

Interestingly, the sizeof operator is one of only three operators in C++ that is a word instead of a symbol. The other two are *new* and *delete*.

You can also use the sizeof operator on a variable name:

```
1  int x;
2  cout << "x is " << sizeof(x) << " bytes"<<endl;
```

```
x is 4 bytes
```

Now you know enough about variables that we can start discussing the different data types!

2.4 — Integers

An **integer** type variable is a variable that can only hold whole numbers (eg. -2, -1, 0, 1, 2). C++ actually has *four* different integer variables available for use: **char**, **short**, **int**, and **long**. The only difference between these different integer types is that they have varying sizes — the larger integers can hold bigger numbers. You can use the sizeof operator to determine how large each type is on your machine.

In the following tutorials, we will typically assume:

- a char is 1 byte
- a short is 2 bytes
- an int is either 2 or 4 bytes
- a long is 4 bytes

Declaring some integers:

```
1  char chChar;
2  short int nShort; // "short int" is technically correct
3  short nShort2; // "short" is preferred shorthand
4  int nInteger;
5  long int nLong; // "long int" is technically correct
   long nLong2; // "long" is preferred shorthand
```

While *short int* and *long int* are technically correct, we prefer to use the shorthand versions *short* and *long* instead. Adding the prefix *int* makes the type harder to distinguish from variables of type int. This can lead to mistakes (such as overflow) if the short or long modifier is inadvertently missed.

Because the size of char, short, int, and long can vary depending on the compiler and/or computer architecture, it can be instructive to refer to integers by their size rather than name. We often refer to integers by the number of bits or bytes a variable of that type is allocated.

As you learned in the last section, a variable with n bits can store 2^n different values. We call the set of values that a data type can hold it's **range**. Integers can have two different ranges, depending on whether they are signed or unsigned.

Signed and unsigned variables

A **signed** integer is a variable that can hold both negative and positive numbers. To declare a variable as signed, you can use the *signed* keyword:

```
1  signed char chChar;
2  signed short nShort;
   signed int nInt;
```

```
3   signed long nLong;
```

A 1-byte signed variable has a range of -128 to 127. Any value between -128 and 127 (inclusive) can be put in a 1-byte signed variable safely.

Sometimes, we know in advance that we are not going to need negative numbers. This is common when using a variable to store the quantity or size of something (such as your height — it doesn't make sense to have a negative height!). An **unsigned** integer is one that can only hold positive values. To declare a variable as unsigned, use the *unsigned* keyword:

```
1   unsigned char chChar;
2   unsigned short nShort;
3   unsigned int nInt;
    unsigned long nLong;
```

A 1-byte unsigned variable has a range of 0 to 255.

Note that declaring a variable as unsigned means that it can not store negative numbers, but it can store positive numbers that are twice as large!

So what happens if we do not declare a variable as signed or unsigned? All integer variables except char are signed by default. Char can be either signed or unsigned by default (but is usually signed).

```
1   short nShort; // signed by default
2   int nInt; // signed by default
3   long nLong; // signed by default
4   char chChar; // can be signed or unsigned by default, but probably signed.
```

New programmers sometimes get signed and unsigned mixed up. The following is a simple way to remember the difference: in order to differentiate positive and negative numbers, we typically use a negative sign. If a sign is not provided, we assume a number is positive. Consequently, an integer with a sign (a signed integer) can tell the difference between positive and negative. An integer without a sign (an unsigned integer) assumes all values are positive.

Now that you understand the difference between signed and unsigned, let's take a look at the ranges for different sized signed and unsigned variables:

Size/Type	**Range**
1 byte signed	-128 to 127
1 byte unsigned	0 to 255
2 byte signed	-32,768 to 32,767
2 byte unsigned	0 to 65,535

4 byte signed	-2,147,483,648 to 2,147,483,647
4 byte unsigned	0 to 4,294,967,296
8 byte signed	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807
8 byte unsigned	0 to 18,446,744,073,709,551,615

For the math inclined, an n-bit signed variable has a range of -(2^(n-1)) to (2^(n-1))-1. An n-bit unsigned variable has a range of 0 to (2^n)-1. For the non-math inclined… use the table. :)

What happens if we try to put a number outside of the data type's range into our variable? We get…

Overflow

In binary, we count from 0 to 15 like this: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111. As you can see, the larger numbers require more bits to represent. Because our variables have a fixed number of bits, this puts a limit on the largest number they can hold.

Consider a hypothetical variable that can only hold 4 bits. Any of the binary numbers enumerated above would fit comfortably inside this variable because none of them are larger than 4 bits. But what happens if we try to assign a value that takes 5 bits to our variable? We get overflow: our variable will only store 4 of the 5 bits, and the excess bits are lost.

Overflow occurs when bits are lost because a variable does not have enough memory to store them.

We can see this in action with the following program:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      unsigned short x = 65535; // largest 2-byte unsigned value possible
7      cout << "x was: " << x << endl;
8      x = x + 1; // We desire 65536, but we get overflow!
9      cout << "x is now: " << x << endl;
10 }
```

What do you think the result of this program will be?

```
x was: 65535
x is now: 0
```

What happened? Informally, we overflowed the variable by trying to put a number that was too big into it, and the result is that our value "wrapped around" back to the beginning of the range. For non-integer data types, overflowed variables do not always wrap around the range, so do not rely on this happening!

The following paragraph explains exactly why we ended up getting a value of 0 after overflow. It is optional reading. If all this binary stuff is confusing, you can skip it.

The number 65,535 is represented by the bit pattern `1111 1111 1111 1111` in binary. 65,535 is the largest number an unsigned 2 byte (16-bit) integer can hold, as it uses all 16 bits. When we add 1 to the value, the new value should be 65,536. However, the bit pattern of 65,536 is represented in binary as `1 0000 0000 0000 0000`, which is 17 bits! Consequently, the highest bit (which is the 1) is lost, and the low 16 bits are all that is left. The bit pattern `0000 0000 0000 0000` corresponds to the number 0, which is our result!

Similarly, we can overflow the bottom end of our range as well.

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      unsigned short x = 0; // smallest 2-byte unsigned value possible
7      cout << "x was: " << x << endl;
8      x = x - 1; // We expect -1, we get overflow!
9      cout << "x is now: " << x << endl;
   }
```

```
x was: 0
x is now: 65535
```

In the case of a signed integer, the result is identical.

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      signed short x = 32767; // largest 2-byte signed value possible
7      cout << "x was: " << x << endl;
8      x = x + 1; // We desire 32768, but we get overflow!
9      cout << "x is now: " << x << endl;
   }
```

```
x was: 32767
x is now: -32768
```

Overflow results in information being lost, which is almost never desirable. If there is ANY doubt that a variable might need to store a value that falls outside it's range, use a larger variable!

Integer division

Integer division can also cause issues because dividing 2 integers can produce a fractional result, and integers can not store fractions. Consider the statement `int x = 5 / 3;`. Under normal mathematical rules, x would be assigned the value of 5/3, which is 1.6666. However, in integer division, the fraction is dropped, so x is assigned the value of 1. Integer division always drops the fraction — it does not round.

Fixed width integers

Some compilers provide access to fixed width integers (integers whose size is not dependent on the platform). Since these are not officially part of the C++ standard, information on these has been relegated to appendix A.6 — Fixed width integers. Nevertheless, I recommend you check them out.

2.5 — Floating point numbers

Integers are great for counting whole numbers, but sometimes we need to store very large numbers, or numbers with a fractional component. A **floating point** type variable is a variable that can hold a real number, such as 4.0, 2.5, 3.33, or 0.1226. There are three different floating point data types: **float**, **double**, and **long double**. A float is usually 4 bytes and a double 8 bytes, but these are not strict requirements, so sizes may vary. Long doubles were added to the language after it's release for architectures that support even larger floating point numbers. But typically, they are also 8 bytes, equivalent to a double. Floating point data types are always signed (can hold positive and negative values).

Here are some declarations of floating point numbers:

```
1  float fValue;
2  double dValue;
   long double dValue2;
```

The *floating* part of the name *floating point* refers to the fact that a floating point number can have a variable number of decimal places. For example, 2.5 has 1 decimal place, whereas 0.1226 has 4 decimal places.

When we assign numbers to floating point numbers, it is convention to use at least one decimal place. This helps distinguish floating point values from integer values.

```
1  int nValue = 5; // 5 means integer
2  float fValue = 5.0; // 5.0 means floating point
```

How floating point variables store information is beyond the scope of this tutorial, but it is very similar to how numbers are written in scientific notation. **Scientific notation** is a useful shorthand for writing lengthy numbers in a concise manner. In scientific notation, a number has two parts: the significand, and a power of 10 called an exponent. The letter 'e' or 'E' is used to separate the two parts. Thus, a number such as 5e2 is equivalent to 5 * 10^2, or 500. The number 5e-2 is equivalent to 5 * 10^-2, or 0.05.

In fact, we can use scientific notation to assign values to floating point variables.

```
1  double dValue1 = 500.0;
2  double dValue2 = 5e2; // another way to assign 500
3
   double dValue3 = 0.05;
4  double dValue4 = 5e-2; // another way to assign 0.05
```

Furthermore, if we output a number that is large enough, or has enough decimal places, it will be printed in scientific notation:

```
1  #include <iostream>
2  int main()
3  {
4      using namespace std;
5
6      double dValue = 1000000.0;
7      cout << dValue << endl;
8      dValue = 0.00001;
9      cout << dValue << endl;
10     return 0;
11 }
```

Outputs:

```
1e+006
1e-005
```

Precision

Consider the fraction 1/3. The decimal representation of this number is 0.33333333333333… with 3′s going out to infinity. An infinite length number would require infinite memory, and we typically only have 4 or 8 bytes. Floating point numbers can only store a certain number of digits, and the rest are lost. The **precision** of a floating point number is how many digits it can represent without information loss.

When outputting floating point numbers, cout has a default precision of 6 — that is, it assumes all variables are only significant to 6 digits, and hence it will truncate anything after that.

The following program shows cout truncating to 6 digits:

```
1  #include <iostream>
2  int main()
3  {
4      using namespace std;
5      float fValue;
6      fValue = 1.222222222222222f;
7      cout << fValue << endl;
8      fValue = 111.22222222222222f;
9      cout << fValue << endl;
10     fValue = 111111.222222222222f;
11     cout << fValue << endl;
12 }
```

This program outputs:

```
1.22222
111.222
111111
```

Note that each of these is only 6 digits.

However, we can override the default precision that cout shows by using the setprecision() function that is defined in a header file called iomanip.

```
#include <iostream>
#include <iomanip> // for setprecision()
int main()
{
    using namespace std;

    cout << setprecision(16); // show 16 digits
    float fValue = 3.33333333333333333333333333333333333333f;
    cout << fValue << endl;
    double dValue = 3.33333333333333333333333333333333333333;
    cout << dValue << endl;
```

Outputs:

```
3.333333253860474
3.333333333333334
```

Because we set the precision to 16 digits, each of the above numbers has 16 digits. But, as you can see, the numbers certainly aren't precise to 16 digits!

Variables of type float typically have a precision of about 7 significant digits (which is why everything after that many digits in our answer above is junk). Variables of type double typically have a precision of about 16 significant digits. Variables of type double are named so because they offer approximately double the precision of a float.

Now let's consider a really big number:

```
#include <iostream>

int main()
{
    using namespace std;
    float fValue = 123456789.0f;
    cout << fValue << endl;
    return 0;
}
```

Output:

```
1.23457e+008
```

1.23457e+008 is 1.23457 * 10^8, which is 123457000. Note that we have lost precision here too!

Consequently, one has to be careful when using floating point numbers that require more precision than the variables can hold.

Rounding errors

One of the reasons floating point numbers can be tricky is due to non-obvious differences between binary and decimal (base 10) numbers. In normal decimal numbers, the fraction 1/3rd is the infinite decimal sequence: 0.333333333… Similarly, consider the fraction 1/10. In decimal, this is easy represented as 0.1, and we are used to thinking of 0.1 as an easily representable number. However, in binary, 0.1 is represented by the infinite sequence: 0.00011001100110011…

You can see the effects of this in the following program:

```
1  #include <iomanip>
2  int main()
3  {
4      using namespace std;
5      cout << setprecision(17);
6      double dValue = 0.1;
       cout << dValue << endl;
   }
```

This outputs:

```
0.10000000000000001
```

Not quite 0.1! This is because the double had to truncate the approximation due to it's limited memory, which resulted in a number that is not exactly 0.1. This is called a **rounding error**.

Rounding errors can play havoc with math-intense programs, as mathematical operations can compound the error. In the following program, we use 9 addition operations.

```
1  #include <iostream>
2  #include <iomanip>
3  int main()
4  {
5      using namespace std;
6      cout << setprecision(17);
7      double dValue;
       dValue = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;
8      cout << dValue << endl;
   }
```

This program should output 1, but it actually outputs:

```
0.99999999999999989
```

Note that the error is no longer in the last column like in the previous example! It has propagated to the second to last column. As you continue to do mathematical operations, this error can propagate further, causing the actual number to drift farther and farther from the number the user would expect.

Comparison of floating point numbers

One of the things that programmers like to do with numbers and variables is see whether two numbers or variables are equal to each other. C++ provides an operator called the equality operator (==) precisely for this purpose. For example, we could write a code snippet like this:

```
1  int x = 5; // integers have no precision issues
2  if (x==5)
3      cout << "x is 5" << endl;
4  else
       cout << "x is not 5" << endl;
```

This program would print "x is 5".

However, when using floating point numbers, you can get some unexpected results if the two numbers being compared are very close. Consider:

```
1  float fValue1 = 1.345f;
2  float fValue2 = 1.123f;
3  float fTotal = fValue1 + fValue2; // should be 2.468
4
5  if (fTotal == 2.468)
6      cout << "fTotal is 2.468";
7  else
       cout << "fTotal is not 2.468";
```

This program prints:

```
fTotal is not 2.468
```

This result is due to rounding error. fTotal is actually being stored as 2.4679999, which is not 2.468!

For the same reason, the comparison operators >, >=, <, and <= may produce the wrong result when comparing two floating point numbers that are very close.

Conclusion

To summarize, the two things you should remember about floating point numbers:
1) Floating point numbers offer limited precision. Floats typically offer about 7 significant digits worth of precision, and doubles offer about 16 significant digits. Trying to use more significant digits will result in a loss of precision. (Note: placeholder zeros do not count as significant digits, so a number like 22,000,000,000, or 0.00000033 only counts for 2 digits).

2) Floating point numbers often have small rounding errors. Many times these go unnoticed because they are so small, and because the numbers are truncated for output before the error propagates into the part that is not truncated. Regardless, comparisons on floating point numbers may not give the expected results when two numbers are close.

The section on relational operators has more detail on comparing floating point numbers.

2.6 — Boolean Values

The next data type we're going to look at is the boolean data type. Boolean variables only have two possible values: true (1) and false (0).

To declare a boolean variable, we use the keyword *bool*.

```
1 bool bValue;
```

When assigning values to boolean variables, we use the keywords *true* and *false*.

```
1 bool bValue1 = true; // explicit assignment
2 bool bValue2(false); // implicit assignment
```

Just as the unary minus operator (-) can be used to make an integer negative, the logical NOT operator (!) can be used to flip a boolean value from true to false, or false to true:

```
1 bool bValue1 = !true; // bValue1 will have the value false
2 bool bValue2(!false); // bValue2 will have the value true
```

When boolean values are evaluated, they actually don't evaluate to true or false. They evaluate to the numbers 0 (false) or 1 (true). Consequently, when we print their values with cout, it prints 0 for false, and 1 for true:

```
1 bool bValue = true;
2 cout << bValue << endl;
3 cout << !bValue << endl;
4
5 bool bValue2 = false;
6 cout << bValue2 << endl;
7 cout << !bValue2 << endl;
```

Outputs:

```
1
0
0
1
```

One of the most common uses for boolean variables is inside *if* statements:

```
1 bool bValue = true;
2 if (bValue)
3     cout << "bValue was true" << endl;
4 else
5     cout << "bValue was false" << endl;
```

Output:

```
bValue was true
```

Don't forget that you can use the logical not operator to reverse a boolean value:

```
1  bool bValue = true;
2  if (!bValue)
3      cout << "The if statement was true" << endl;
4  else
       cout << "The if statement was false" << endl;
```

Output:

```
The if statement was false
```

Boolean values are also useful as the return values for functions that check whether something is true or not. Such functions are typically named starting with the word *Is*:

```
1  #include <iostream>
2
3  using namespace std;
4
5  // returns true if x and y are equal
6  bool IsEqual(int x, int y)
7  {
8      return (x == y); // use equality operator to test if equal
9  }
10
11 int main()
12 {
13     cout << "Enter a value: ";
14     int x;
15     cin >> x;
16
17     cout << "Enter another value: ";
18     int y;
19     cin >> y;
20
21     bool bEqual = IsEqual(x, y);
22     if (bEqual)
23         cout << x << " and " << y << " are equal"<<endl;
       else
           cout << x << " and " << y << " are not equal"<<endl;
       return 0;
   }
```

In this case, because we only use bEqual in one place, there's really no need to assign it to a variable. We could do this instead:

```
1  if (IsEqual(x, y))
       cout << x << " and " << y << " are equal"<<endl;
```

```
2
3   else
        cout << x << " and " << y << " are not equal"<<endl;
```

IsEqual() evaluates to true or false, and the if statement then branches based on this value.

Boolean variables are quite refreshing in their simplicity!

One additional note: when converting integers to booleans, the integer zero resolves to boolean false, whereas non-zero integers all resolve to true.

2.7 — Chars

Even though the *char* data type is an integer (and thus follows all of the normal integer rules), we typically work with chars in a different way than normal integers. Characters can hold either a small number, or a letter from the ASCII character set. **ASCII** stands for American Standard Code for Information Interchange, and it defines a mapping between the keys on an American keyboard and a number between 1 and 127 (called a code). For instance, the character 'a' is mapped to code 97. 'b' is code 98. Characters are always placed between single quotes.

The following two assignments do the same thing:

```
1  char chValue = 'a';
2  char chValue2 = 97;
```

cout outputs char type variables as characters instead of numbers.

The following snippet outputs 'a' rather than 97:

```
1  char chChar = 97; // assign char with ASCII code 97
2  cout << chChar; // will output 'a'
```

If we want to print a char as a number instead of a character, we have to tell cout to print the char as if it were an integer. We do this by using a *cast* to have the compiler convert the char into an int before it is sent to cout:

```
1  char chChar = 97;
2  cout << (int)chChar; // will output 97, not 'a'
```

The *(int)* cast tells the compiler to convert chChar into an int, and cout prints ints as their actual values. We will talk more about casting in a few lessons.

The following program asks the user to input a character, then prints out both the character and it's ASCII code:

```
1  #include "iostream";
2
3  int main()
4  {
5      using namespace std;
6      char chChar;
7      cout << "Input a keyboard character: ";
8      cin >> chChar;
9      cout << chChar << " has ASCII code " << (int)chChar << endl;
10 }
```

```
10
```

Note that even though cin will let you enter multiple characters, chChar will only hold 1 character. Consequently, only the first character is used.

One word of caution: be careful not to mix up character (keyboard) numbers with actual numbers. The following two assignments are not the same

```
1  char chValue = '5'; // assigns 53 (ASCII code for '5')
2  char chValue2 = 5; // assigns 5
```

Escape sequences

C and C++ have some characters that have special meaning. These characters are called **escape sequences**. An escape sequence starts with a \, and then a following letter or number.

The most common escape sequence is '\n', which can be used to embed a newline in a string of text:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "First line\nSecond line" << endl;
7      return 0;
8  }
```

This outputs:

```
First line
Second line
```

Another commonly used escape sequence is '\t', which embeds a tab:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "First part\tSecond part";
   }
```

Which outputs:

```
First part        Second part
```

Three other notable escape sequences are:
\', which prints a single quote
\", which prints a double quote
\\, which prints a backslash

Here's a table of all of the escape sequences:

Name	**Symbol**	**Meaning**
Alert	\a	Makes an alert, such as a beep
Backspace	\b	Moves the cursor back one space
Formfeed	\f	Moves the cursor to next logical page
Newline	\n	Moves cursor to next line
Carriage return	\r	Moves cursor to beginning of line
Horizontal tab	\t	Prints a horizontal tab
Vertical tab	\v	Prints a vertical tab
Single quote	\'	Prints a single quote
Double quote	\"	Prints a double quote
Backslash	\\	Prints a backslash
Question mark	\?	Prints a question mark
Octal/hex number	\(number)	Translates into char represented by octal/hex number

2.8 — Constants

C++ has two kinds of constants: literal, and symbolic.

Literal constants

Literal constants are literal numbers inserted into the code. They are constants because you can't change their values.

```
1 int x = 5; // 5 is a literal constant
```

Literal constants can have suffixes that determine their types. Integer constants can have a u or U suffix that means they are unsigned. Integer constants can also have a l or L suffix, which means they are long integers. However, these suffixes are typically optional, as the compiler can usually tell from context what kind of constant you need.

```
1 unsigned int nValue = 5u; // unsigned constant
2 long nValue2 = 5L; // long constant
```

By default, floating point literal constant have a type of *double*. To convert them into a float value, the f or F suffix can be used:

```
1 float fValue = 5.0f; // float constant
```

Floating point literal constants can also use an l or L suffix to make them long doubles.

Generally, it is a good idea to try to avoid using literal constants that aren't 0 or 1. For more detail, you can review the section on magic numbers, and why they are a bad idea.

Symbolic constants

As you learned in a previous lesson, you can use the #define preprocessor directive in order to declare a symbolic constant:

```
1 #define YEN_PER_DOLLAR  122
2 int nYen = nDollars * YEN_PER_DOLLAR;
```

There are two major problems with symbolic constants declared using #define. First, because they are resolved by the preprocessor, which replaces the symbolic name with the defined value, #defined symbolic constants do not show up in the debugger. Thus, if you only saw the statement `int nYen = nDollars * YEN_PER_DOLLAR;`, you would have to go looking for the #define declaration in order to find out what value of YEN_PER_DOLLAR was used.

Second, #defined values always have global scope (which we'll talk about in the section on local and global variables). This means a value #defined in one piece of code may have a naming conflict with a value #defined with the same name in another piece of code.

A better way to do symbolic constants is through use of the **const** keyword. Const variables must be assigned a value when declared, and then that value can not be changed. Here is the way the above snippet of code should be written:

```
1  const int nYenPerDollar = 122;
2  int nYen = nDollars * nYenPerDollar;
```

Declaring a variable as const prevents us from inadvertently changing it's value:

```
1  const int nYenPerDollar = 122;
2  nYenPerDollar = 123; // compiler error!
```

Although a constant variable might seem like an oxymoron, they can be very useful in helping to document your code and avoid magic numbers. Some programmers prefer to use all upper-case names for const variables (to match the style of #defined values). However, we will use normal variable naming conventions, which is more common. Const variables act exactly like normal variables in every case except that they can not be assigned to.

2.9 — Hungarian Notation

Hungarian Notation is a naming convention in which the type and/or scope of a variable is used as a naming prefix for that variable. For example:

```
1  int value; // non-Hungarian
2  int nValue; // the n prefix denotes an integer
3
4  double width; // non-Hungarian
5  double dWidth; // the d prefix denotes a double
```

Hungarian Notation was invented in 1972 by Charles Simonyi, a Microsoft programmer. The original idea of Hungarian Notation was to encode information about the variable's purpose, which is known as Apps Hungarian. Over time, this focus changed to encoding information about the variable's type and/or scope, which is known as Systems Hungarian.

There is a lot of controversy over whether Hungarian Notation is useful in modern programming languages and with modern IDEs. We believe the advantages still outweigh the disadvantages, though you will find plenty of programmers who disagree.

One advantage of Hungarian Notation is that the variable type can be determined from it's name. Many argue that this is an obsolete advantage, because most modern IDEs will tell you a variables type if you mouse-hover over the name. However, consider the following snippet:

```
1  float applesPerPerson = totalApples / totalPersons;
```

Casually browsing the code, this statement would probably not attract notice. But there is a good chance it's wrong. If totalApples and totalPersons are both integers, the compiler will evaluate `totalApples / totalPersons` using integer division, causing any fractions to be lost before the value is assigned to applesPerPerson. Thus, if totalApples = 5, and totalPersons = 3, applesPerPerson will be assigned 1 instead of the expected 1.66!

However, if we use Hungarian Notation variable names:

```
1  float fApplesPerPerson = nTotalApples / nTotalPersons;
```

The n prefixes make it clear from just browsing the code that this is an integer division that's going to cause us problems! Furthermore, as you code, the n prefix will remind you to watch out for integer division and overflow issues every time you use an integer variable in an expression or statement.

Another advantage of Hungarian Notation is that it gives us a way to name variables using shorthand. For example, bError is understood to mean isError, and nApples is a shorthand way of writing numberOfApples.

One perceived disadvantage of Hungarian Notation is that it leads to extra work when a variable's type changes. For example, it is common to declare an integer variable and then later change it to a double variable because you need to deal with fractional values. Without using Hungarian Notation, you could change `int value` to `double value` and go on your merry way. However, in Hungarian Notation, you'd not only have to change the declaration `int nValue` to `double dValue`, you'd have to change every use of *nValue* in your entire program to *dValue*! If you do not, your naming scheme will be misleading and inconsistent.

While replacing a potentially huge number of variable names is certainly a nuisance, we believe it is also a good thing. Because different types have different behaviors, having to explicitly replace your variable names encourages you to examine your code to ensure you're not doing anything dangerous with the new type.

For example, without Hungarian Notation, you might have written

```
1 if (value == 0)
2     // do something
```

When value is changed from an int to a double, your safe integer comparison is now an unsafe floating point comparison that may produce unexpected results! In the best case, this error shows up when testing your program, and you have to spend time debugging it. In the worst case, the bug ships out and you end up with millions of customers have software that doesn't work right!

However, if you'd used Hungarian Notation and written:

```
1 if (nValue == 0)
2     // do something
```

And were forced to change it to:

```
1 if (dValue == 0.0)
2     // do something
```

Hopefully at this point you'd say, "Hey, wait a second, I shouldn't be doing naked comparisons with floating point values!". Then you could modify it to something more appropriate and move on. In the long run, this can actually save you lots of time.

A real disadvantage of traditional Hungarian Notation is that the number of prefixes for compound types can become confusing. Wikipedia provides an appropriate example: "a_crszkvc30LastNameCol : a constant reference function argument, holding contents of a database column of type varchar(30) called LastName that was part of the table's primary key". a_crszkvc is non-trivial to decipher, and makes your code *less* clear.

As an aside, Hungarian Notation got it's name from prefixes such as *a_crszkvc* that look like they're written in Hungarian!

Caste Hungarian

Different programmers and/or companies tend to use different varieties of Systems Hungarian of varying complexity. Although most of them have some commonality (like using a *d* prefix for double, and an *n* (or *i*) prefix for integers), there is a lot of variation as to which types get what prefixes, and how those prefixes should combine.

We believe that using a different prefix for each data type is overkill, especially in the case of structs and classes, which can be user defined to a high degree. Furthermore, long Hungarian looking prefixes obscure code clarity more than they help it. Consequently we advocate a simplified version of Systems Hungarian called "Caste Hungarian". In Caste Hungarian, Hungarian Notation is used mostly to denote which "caste" of data type a variable falls into (integers, floating points, classes, etc…).

Variable prefixes are composed of 3 parts: a scope modifier, a type modifier, and a type prefix (in that order). Scope modifier and type modifier may not apply. Consequently, the overall prefix length is kept reasonable, with the average prefix length being around 2 letters. This system conveys most of the advantages of Hungarian Notation without many of it's disadvantages, and it keeps the entire system simple and easy to use.

The type prefix indicates the data type of the variable.

Type prefix	**Meaning**	**Example**
b	boolean	bool bHasEffect;
c (or none*)	class	Creature cMonster;
ch	char (used as a char)	char chLetterGrade;
d	double, long double	double dPi;
e	enum	Color eColor;
f	float	float fPercent;
n	short, int, long char used as an integer	int nValue;
s	struct	Rectangle sRect;
str	C++ string	std::string strName;
sz	Null-terminated string	char szName[20];

The following type modifiers are placed before the prefix if they apply:

Type modifier	Meaning	Example
a	array on stack	int anValue[10];
p	pointer	int* pnValue;
pa	dynamic array	int* panValue = new int[10];
r	reference	int rnValue;
u	unsigned	unsigned int unValue;

The following scope modifiers are placed before the type modifier if they apply:

Scope modifier	Meaning	Example
g_	global variable	int g_nGlobalValue;
m_	member of class	int m_nMemberValue;
s_	static member of class	int s_nValue;

A few notes:

1. This list is not exhaustive. It is meant to cover the most common cases. If you feel a variable of a different type deserves it's own prefix, give it one!
2. Use meaningful variable names and suffixes to clarify your variables. This is especially important with struct and class variables. For example, a Rectangle struct variable holding the position and size of a window is better declared as `Rectangle sWindowRect;` than `Rectangle sWindow;`
3. Char has a different prefix depending on whether it's being used as an ASCII character or integer. This helps clarify it's intended use and prevent mistakes.
4. Float has a different prefix than double because floating point literals are doubles by default. Float literals need a f suffix.
5. Typedefs don't fall very well into this system.
6. The 'c' prefix for a class can be omitted if the variable is a pointer or a reference to a class.
7. Because integer types are not differentiated, you can easily change to a larger or smaller integer as needed without changing the variable name. However, changing to a smaller integer is generally not recommended due to potential overflow issues.

Here are a few sample declarations:

```
1  int nIndex; // simple integer type prefix
2  int* pnIndex;  // a pointer to an integer
3  int m_nIndex; // an integer variable that is a member of a class
4  int* m_pnIndex; // an pointer to an integer variable that is a member of a
   class
```

2.10 — Comprehensive quiz

Quick Review

Integers are used for holding whole numbers. When using integers, keep an eye out for overflow and integer division problems.

Floating point numbers are used for holding real numbers (which can have fractional components). When using floating point numbers, keep an eye out for precision issues, rounding errors, and comparison issues.

Boolean values hold only true and false. They do not have any major issues.

Char values are integers that can be interpreted as an ASCII value. When using chars, be careful not to mix up ASCII code values and numbers, and watch for overflow and integer division problems.

Use the const keyword to declare symbolic constants instead of #define. It's safer.

Comprehensive quiz

1) Why are symbolic constants usually a better choice than literal constants? Why are const symbolic constants usually a better choice than #defined symbolic constants?

2) Pick the appropriate data type for a variable in each of the following situations. Be as specific as possible. If the answer is an integer, pick a specific integer type (eg. short) based on range. If the variable should be unsigned or const, say so.

a) The age of the user (in years)
b) Whether the user wants color or not
c) pi (3.14159265)
d) The number of pages in a textbook
e) The price of a stock in dollars (to 2 decimal places)
f) How many times you've blinked since you were born (note: answer is in the millions)
g) A user selecting an option from a menu by letter
h) The year someone was born

3) Declare each of the above using Caste Hungarian Notation. Pick a good variable name. Don't forget to assign values to any const variables.

4) Write the following program: The user is asked to enter 2 floating point numbers (use doubles). The user is then asked to enter one of the following mathematical symbols: +, -, *, or /.

The program computes the answer on the two numbers the user entered and prints the results. If the user enters an invalid symbol, the program should print nothing.

Example of program:

```
Enter a value: 7
Enter a second value: 5
Enter one of the following: +, -, *, or /: *
7 * 5 is 35
```

Hint: You can check if the user has entered a plus symbol using an if statement (briefly covered in section 2.6) and the equality operator (used to compare two values for equality): `if (chUserInput == '+') // fill in rest of code here`

Solutions

1) Answer

Using literal constants (aka. magic numbers) in your program makes your program harder to understand and harder to modify. Symbolic constants help document what the numbers actually represent, and changing a symbolic constant at it's declaration changes the value everywhere it is used. #define constants do not show up in the debugger and are more likely to have naming conflicts.

2) Answer

a) Since the oldest person alive is about 120 years old, a signed char would probably be okay. But since it's possible that someone could live to be older than 127 (though not likely), an unsigned char would be even better.
b) bool
c) const double
d) Since books can not have negative pages, an unsigned short is best
e) float
f) an unsigned long
g) char
h) a signed short. You can use positive numbers to represent AD birthdates, and negative numbers to represent BC birthdates.

3) Answer

a) unsigned char unAgeInYears;
b) bool bColor;
c) const double dPi = 3.14159265;
d) unsigned short unPages;
e) float fStockPrice;
f) unsigned long nBlinks;

g) char chSelection;
h) short nBirthYear;

4) Answer

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "Enter a value: ";
7      double dX;
8      cin >> dX;
9
10     cout << "Enter a second value: ";
11     double dY;
12     cin >> dY;
13
14     cout << "Enter one of the following: +, -, *, or /";
15     char chChoice;
16     cin >> chChoice;
17
18     if (chChoice == '+')
19         cout << dX << " + " << dY << " is " << dX + dY << endl;
20     if (chChoice == '-')
21         cout << dX << " - " << dY << " is " << dX - dY << endl;
22     if (chChoice == '*')
23         cout << dX << " * " << dY << " is " << dX * dY << endl;
24     if (chChoice == '/')
25         cout << dX << " / " << dY << " is " << dX / dY << endl;
26
27     return 0;
28 }
```

Operators

3.1 — Precedence and associativity

In order to properly evaluate an expression such as `4 + 2 * 3`, we must understand both what the operators do, and the correct order to apply them. The order in which operators are evaluated in a compound expression is called **operator precedence**. Using normal mathematical precedence rules (which states that multiplication is resolved before addition), we know that the above expression should evaluate as `4 + (2 * 3) = 10`.

In C++, all operators are assigned a level of precedence. Those with the highest precedence are evaluated first. You can see in the table below that multiplication and division (precedence level 5) have a higher precedence than addition and subtraction (precedence level 6). The compiler uses these levels to determine how to evaluate expressions it encounters.

If two operators with the same precedence level are adjacent to each other in an expression, the **associativity rules** tell the compiler whether to evaluate the operators from left to right or from right to left. For example, in the expression `3 * 4 / 2`, the multiplication and division operators are both precedence level 5. Level 5 has an associativity of left to right, so the expression is resolved from left to right: `(3 * 4) / 2 = 6`.

Prec/Ass	Operator	Description	Example
1 None	:: ::	Global scope (unary) Class scope (binary)	::g_nGlobalVar = 5; Class::m_nMemberVar = 5;

2 L->R	()
	()
	()
	[]
	.
	->
	++
	—
	typeid
	const_cast
	dynamic_cast
	reinterpret_cast
	static_cast
		Parenthesis
		Function call
		Implicit assignment
		Array subscript
		Member access from object
		Member access from object ptr
		Post-increment
		Post-decrement
		Run-time type information
		Cast away const
		Run-time type-checked cast
		Cast one type to another
		Compile-time type-checked cast
			(x + y) * 2;
			Add(x, y);
			int nValue(5);
			aValue[3] = 2;
			cObject.m_nValue = 4;
			pObject->m_nValue = 4;
			nValue++;
			nValue—;
			typeid(cClass).name();
			const_cast<int*>(pnConstValue);
			dynamic_cast<Shape*>(pShape);
			reinterpret_cast<Class2>(cClass1);
			fValue =
			static_cast<float>(nValue);

3 R->L	+
	-
	++
	—
	!
		Unary plus
		Unary minus
		Pre-increment
		Pre-decrement
		Logical NOT
			nValue = +5;M
			nValue = -1;
			++nValue;
			—nValue;
			if (!bValue)

	~
	(type)
	sizeof
	&
	*
	new
	new[]
	delete
	delete[]
		Bitwise NOT
		C-style cast
		Size in bytes
		Address of
		Dereference
		Dynamic memory allocation
		Dynamic array allocation
		Dynamic memory deletion
		Dynamic array deletion
			nFlags = ~nFlags;
			float fValue = (float)nValue;
			sizeof(int);
			address = &nValue;
			nValue = *pnValue;
			int *pnValue = new int;
			int *panValue = new int[5];
			delete pnValue;
			delete[] panValue;

4 L->R	->* .*	Member pointer selector Member object selector	pObject->*pnValue = 24; cObject->.*pnValue = 24;

5 L->R	*
	/
	%
		Multiplication
		Division
		Modulus
			int nValue = 2 * 3;
			float fValue = 5.0 / 2.0;
			int nRemainder = 10 % 3;

6 L->R	+ -	Addition Subtraction	int nValue = 2 + 3; int nValue = 2 – 3;
7 L->R	<< >>	Bitwise shift left Bitwise shift right	int nFlags = 17 << 2; int nFlags = 17 >> 2;

8 L->R	<
	<=
	>
	>=
		Comparison less than
		Comparison less than or equals
		Comparison greater than
		Comparison greater than or equals
			if (x < y)
			if (x <= y)
			if (x > y)
			if (x >= y)

9 L->R	== !=	Equality Inequality	if (x == y) if (x != y)
10 L->R	&	Bitwise AND	nFlags = nFlags & 17;
11 L->R	^	Bitwise XOR	nFlags = nFlags ^ 17;
12 L->R	|	Bitwise OR	nFlags = nFlags | 17;
13 L->R	&&	Logical AND	if (bValue1 && bValue2)
14 L->R	||	Logical OR	if (bValue1 || bValue2)
15 L->R	?:	Arithmetic if	return (x < y) ? true : false;

16 R->L	=
	*=
	/=
	%=
	+=
	-=
	<<=
	>>=
		Assignment
		Multiplication assignment
		Division assignment
		Modulus assignment
		Addition assignment
		Subtraction assignment
		Bitwise shift left assignment
		Bitwise shift right assignment
			nValue = 5;
			nValue *= 5;
			fValue /= 5.0;
			nValue %= 5;
			nValue += 5;
			nValue -= 5;
			nFlags <<= 2;
			nFlags >>= 2;

	&=
	|=
	^=
		Logical AND assingment
		Logical OR assignment
		Logical XOR assignment
			nFlags &= 17;
			nFlags |= 17;
			nFlags ^= 17;

17 L->R	,	Comma operator	iii++, jjj++, kkk++;

A few operators you should already recognize: +, -, *, /, (), =, <, >, <=, and >=. These arithmetic and relational operators have the same meaning in C++ as they do in every-day usage.

However, unless you have experience with another programming language, it's likely the majority of the operators in this table will be incomprehensible to you at this point in time. That's expected at this point. We'll cover many of them in this chapter, and the rest will be introduced as there is a need for them.

The above table is primarily meant to be a reference chart that you can refer back to in the future to resolve any precedence or associativity questions you have.

3.2 — Arithmetic operators

Unary arithmetic operators

There are two unary arithmetic operators, plus (+), and minus (-). If you remember, unary operators are operators that only take one operand.

Operator	Symbol	Form	Operation
Unary plus	+	+x	Value of x
Unary minus	-	-x	Negation of x

The unary plus operator returns the value of the operand. In other words, +5 = 5, and +x = x.

The unary minus operator returns the operand multiplied by -1. In other words, if x = 5, -x = -5.

For best effect, both of these operators should be placed immediately preceding the operand (eg. `-x`, not `- x`).

Do not confuse the unary minus operator with the binary subtraction operator, which uses the same symbol. For example, in the expression `x = 5 - -3;`, the first minus is the subtraction operator, and the second is the unary minus operator.

Binary arithmetic operators

There are 5 binary arithmetic operators. Binary operators are operators that take a left and right operand.

Operator	Symbol	Form	Operation
Addition	+	x + y	x plus y
Subtraction	-	x – y	x minus y
Multiplication	*	x * y	x multiplied by y
Division	/	x / y	x divided by y
Modulus (Remainder)	%	x % y	The remainder of x divided by y

Of these operators, the only ones that really need further explanation are division and modulus (remainder).

Integer and floating point division

It is easiest to think of the division operator as having two different "modes". If both of the operands are integers, the division operator performs integer division. Integer division drops any fractions and returns an integer value. For example, 7 / 3 = 2 because the fraction is dropped. Note that integer division does not round. For example, 3 / 4 = 0, not 1.

If either or both of the operands are floating point values, the division operator performs floating point division. Floating point division returns a floating point value, and the fraction is kept. For example, 7.0 / 3 = 2.333, 7 / 3.0 = 2.333, and 7.0 / 3.0 = 2.333.

Note that trying to divide by 0 (or 0.0) will generally cause your program to crash, as the result are undefined!

Modulus (remainer)

The modulus operator is also informally known as the remainder operator. The modulus operator only works on integer operands, and it returns the remainder after doing integer division. For example, 7 / 3 = 2 remainder 1, thus 7 % 3 = 1. As another example, 25 / 7 = 3 remainder 4, thus 25 % 7 = 4.

Modulus is very useful for testing whether a number is evenly divisible by another number: if x % y == 0, then we know that y divides evenly into x. This can be useful when trying to do something every nth iteration.

For example, say we wanted to write a program that printed every number from 1 to 100 with 20 numbers per line. We could use the modulus operator to determine where to do the line breaks. Even though you haven't seen a while statement yet, the following program should be pretty comprehensible:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6
7      // nCount holds the current number to print
8      int nCount = 1; // start at 1
9
10     // Loop continually until we pass number 100
11     while (nCount <= 100)
12     {
13         cout << nCount << " "; // print the current number
14
15         // if nCount is divisible by 20, print a new line
16         if (nCount % 20 == 0)
17             cout << endl;
18
19         nCount = nCount + 1; // go to next number
20     } // end of while
21 } // end of main()
```

```
19
20
21
```

A warning about integer division and modulus with negative numbers

If either or both operands of integer division are negative, the compiler is free to truncate up or down! Most modern compilers round towards 0. For example, -5 / 2 can evaluate to either -3 or -2, depending on whether the compiler rounds down or rounds toward 0.

If either operand of the modulus operator is negative, the results of the modulus can be either negative or positive! For example, -5 % 2 can evaluate to either 1 or -1.

Arithmetic assignment operators

Operator	Symbol	Form	Operation
Assignment	=	x = y	Assign value y to x
Addition assignment	+=	x += y	Add y to x
Subtraction assignment	-=	x -= y	Subtract y from x
Multiplication assignment	*=	x *= y	Multiply x by y
Division assignment	/=	x /= y	Divide x by y
Modulus assignment	%=	x %= y	Put the remainder of x / y in x

By now, you should already be well acquainted with the assignment operator, so no need to discuss it here. Because writing statements such as x = x + y is so common, C++ provides 5 arithmetic assignment operators for convenience. For example, instead of writing x = x + y, you can write x += y. Instead of x = x * 5, you can write x *= 5.

Quiz

1) What does the following expression evaluate to? `6 + 5 * 4 % 3`

2) Write a function called IsEven() that returns true if an integer passed to it is even. Use the modulus operator to test whether the integer parameter is even.

3) Name two things to watch out for when using integer division.

Answers

1) Answer

Because * and % have higher precedence than +, the + will evaluate last. We can rewrite our expression as 6 + (5 * 4 % 3). * and % have the same precedence, so we have to look at the

associativity to resolve them. The associativity for * and % is left to right, so we resolve the left operator first. We can rewrite our expression like this: 6 + ((5 * 4) % 3).

6 + ((5 * 4) % 3) = 6 + (20 % 3) = 6 + 2 = 8

2) Answer

```
1  bool IsEven(int x)
2  {
3      // if x % 2 == 0, 2 divides evenly into our number
4      // which means it must be an even number
5      if (x % 2 == 0)
6          return true;
7      else
8          return false;
9  }
```

The following also works:

```
1  bool IsEven(int x)
2  {
3      // if x % 2 == 0, 2 divides evenly into our number
4      // which means it must be an even number
5      return (x % 2) == 0;
6  }
```

3) Answer

When using integer division, a right operand of 0 will have undefined results and probably crash your program. Also, if either of the operands are negative, the direction of truncation is undefined.

3.3 — Increment/decrement operators, and side effects

Incrementing (adding 1 to) and decrementing (subtracting 1 from) a variable are so common that they have their own operators in C. There are actually two version of each operator — a prefix version and a postfix version.

Operator	Symbol	Form	Operation
Prefix increment	++	++x	Increment x, then evaluate x
Prefix decrement	—	—x	Decrement x, then evaluate x
Postfix increment	++	x++	Evaluate x, then increment x
Postfix decrement	—	x—	Evaluate x, then decrement x

The prefix increment/decrement operators are very straightforward. The value of x is incremented or decremented, and then x is evaluated. For example:

```
1 int x = 5;
2 int y = ++x; // x is now equal to 6, and 6 is assigned to y
```

The postfix increment/decrement operators are a little more tricky. The compiler makes a temporary copy of x, increments x, and then evaluates the temporary copy of x.

```
1 int x = 5;
2 int y = x++; // x is now equal to 6, and 5 is assigned to y
```

In the second line of the above example, x is incremented from 5 to 6, but y is assigned the value of the copy of x, which still has the original value of 5.

Here is another example showing the difference between the prefix and postfix versions:

```
1 int x = 5, y = 5;
2 cout << x << " " << y << endl;
3 cout << ++x << " " << --y << endl; // prefix
4 cout << x << " " << y << endl;
5 cout << x++ << " " << y-- << endl; // postfix
6 cout << x << " " << y << endl;
```

This produces the output:

```
5 5
```

```
6 4
6 4
6 4
7 3
```

On the third line, x and y are incremented/decremented before they are evaluated, so their new values are printed by cout. On the fifth line, a temporary copy of the original values (x=6, y=4) is sent to cout, and then the original x and y are incremented. That is why the changes from the postfix operators don't show up until the next line.

Side effects

A **side effect** is a result of an operator, expression, statement, or function that persists even after the operator, expression, statement, or function has finished being evaluated.

Side effects can be useful:

```
1  x = 5;
```

The assignment operator has the side effect of changing the value of x permanently. Even after the statement has finished executing, x will have the value 5.

Side effects can also be dangerous:

```
1  int x = 5;
2  int nValue = Add(x, ++x);
```

C++ does not define the order in which function parameters are evaluated. If the left parameter is evaluated first, this becomes a call to Add(5, 6), which equals 11. If the right parameter is evaluated first, this becomes a call to Add(6, 6), which equals 12!

As a general rule, it is a good idea to avoid the use operators that cause side effects inside of compound expressions. This includes all assignment operators, plus the increment and decrement operators. Any operator that causes a side effect should be placed in it's own statement.

Note that side effects are not confined to operators, expressions, and statements. Functions can also have side effects, which we will discuss in the section on global variables (and why they are evil).

3.4 — Sizeof, comma, and arithmetic if operators

Sizeof

Operator	Symbol	Form	Operation
Sizeof	sizeof	sizeof(type) sizeof(variable)	Returns size of type or variable in bytes

Even though we've already talked about the sizeof operator, we'll cover it again briefly for completeness. The sizeof operator returns the size, in bytes, of a type or a variable.

You can compile and run the following program to find out how large your data types are:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "bool:\t\t" << sizeof(bool) << " bytes" << endl;
7      cout << "char:\t\t" << sizeof(char) << " bytes" << endl;
8      cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes" << endl;
9      cout << "short:\t\t" << sizeof(short) << " bytes" << endl;
10     cout << "int:\t\t" << sizeof(int) << " bytes" << endl;
11     cout << "long:\t\t" << sizeof(long) << " bytes" << endl;
12     cout << "float:\t\t" << sizeof(float) << " bytes" << endl;
13     cout << "double:\t\t" << sizeof(double) << " bytes" << endl;
14     cout << "long double:\t" << sizeof(long double) << " bytes" << endl;
15     return 0;
16 }
```

Here is the output from the author's Pentium 4 machine, using Visual Studio 2005 Express:

```
bool:           1 bytes
char:           1 bytes
wchar_t:        2 bytes
short:          2 bytes
int:            4 bytes
long:           4 bytes
float:          4 bytes
double:         8 bytes
long double:    8 bytes
```

Your results may vary if you are using a different type of machine, or a different compiler.

Comma

Operator	Symbol	Form	Operation
Comma	,	x, y	Evaluate x then y, return value of y

The comma operator allows you to evaluate multiple expressions wherever a single expression is allowed. The comma operator evaluates to it's rightmost operand.

For example:

```
1  int x = 0;
2  int y = 2;
   int z = (++x, ++y); // increment x and y
```

z would be assigned the result of evaluating ++y, which equals 3.

However, in almost every case, a statement written using a comma would be better written as separate statements. For example, the above code should be written as:

```
1  int x = 0;
2  int y = 2;
3  ++x;
4  ++y;
   int z = y;
```

Most programmers do not use the comma operator at all, with the single exception of inside *for loops*, where it's use is fairly common. For example:

```
1  for (int iii = 1, jjj = 10; iii <= 10; iii++, jjj--)
```

The iii variable will count up from 1 to 10, while the jjj variable will count down from 10 to 1. Each iteration of the loop, iii is incremented and jjj is decremented. We will discuss for loops in more detail in the section on flow control statements.

Arithmetic if

Operator	Symbol	Form	Operation
Arithmetic if	?:	c ? x : y	If c is nonzero (true) then evaluate x, otherwise evaluate y

The arithmetic if operator (?:) is also known as the conditional operator, and it is C++'s only ternary operator (it takes 3 operands). The ?: operator provides a shorthand method for doing a particular type of if/else statement.

If/else statements in the following form:

```
if (condition)
    x = some value
else
```

```
    x = some other value
```

can be rewritten as:

```
x = (condition) ? some value : some other value;
```

For example, to put the larger of values x and y in variable z, we could write this:

```
1  if (x > y)
2      z = x;
3  else
       z = y;
```

Or this:

```
1  z = (x > y) ? x : y;
```

It is common to put the conditional part of the expression inside of parenthesis, both to make it easier to read, and also to make sure the precedence is correct.

Keep in mind that the ?: operator has a very low precedence. If doing anything other than using the result in an assignment statement (which has even lower precedence), the ?: statement needs to be wrapped in parenthesis.

For example to print the larger of values x and y to the screen, we could do this:

```
1  if (x > y)
2      cout << x;
3  else
       cout << y;
```

Or we could do this:

```
1  cout << ((x > y) ? x : y);
```

Because the << operator has higher precedence than the ? operator, the statement:

```
1  cout << (x > y) ? x : y;
```

would evaluate as:

```
1  (cout << (x > y)) ? x : y;
```

That would print 1 (true) if x > y, or 0 (false) otherwise!

The arithmetic if gives us a convenient way to simplify simple if/else statements. It should not be used for complex if/else statements, as it quickly becomes both unreadable and error prone.

3.5 — Relational operators (comparisons)

There are 6 relational operators:

Operator	Symbol	Form	Operation
Greater than	>	x > y	true if x is greater than y, false otherwise
Less than	<	x < y	true if x is less than y, false otherwise
Greater than or equals	>=	x >= y	true if x is greater than or equal to y, false otherwise
Less than or equals	<=	x <= y	true if x is less than or equal to y, false otherwise
Equality	==	x == y	true if x equals y, false otherwise
Inequality	!=	x != y	true if x does not equal y, false otherwise

You have already seen how all of these work, and they are pretty intuitive. Each of these operators evaluates to the boolean value true (1), or false (0).

Keep in mind that comparing floating point values using any of these operators is dangerous. This is because small rounding errors in the floating point operands may cause an unexpected result. See the section on floating point numbers for more details.

However, sometimes the need to do floating point comparisons is unavoidable. In this case, the less than and greater than operators (>, >=, <, and <=) are typically used with floating point values as normal. The operators will produce the correct result most of the time, only potentially failing when the two operands are almost identical. Due to the way these operators tends to be used, a wrong result typically only has slight consequences.

The equality operator is much more troublesome since small rounding errors make it almost useless. Consequently, using the == operator on floating point numbers is not advised. The most common method of doing floating point equality involves using a function that calculates how close the two values are to each other. If the two numbers are "close enough", then we call them equal.

Donald Knuth, a famous computer scientist, suggested the following method in his book “The Art of Computer Programming, Volume II: Seminumerical Algorithms (Addison-Wesley, 1969)”:

```
1  bool IsEqual(double dX, double dY)
2  {
3      const double dEpsilon = 0.000001; // or some other small number
4      return fabs(dX - dY) <= dEpsilon * fabs(dX);
5  }
```

dEpsilon is a very small value (eg. 0.000001) that is used to help define what “close enough” is. fabs() is a function in the standard library (#include <cmath>) that returns the absolute value of it’s double parameter.

Let’s examine how the IsEqual() function works. On the left side of the <= operator, the absolute value of dX - dY tells how close dX and dY are to each other as a positive number.

It is easiest to think of dEpsilon as a percentage. A dEpsilon of 0.01 means that dX and dY have to be within 1% of each other in order to be considered equal. On the right side of the <= operator, we multiply dEpsilon by fabs(dX) to find the largest distance the two numbers can be apart and still be considered equal. For example, if fabs(dX) evaluates to 1000, and dEpsilon is 0.01, the largest distance apart the two numbers can be is 10.

Finally, we compare the distance between dX and dY with the largest distance apart that they can still be considered "close enough". If they are close enough, the function returns true. Otherwise, it returns false.

The value for dEpsilon can be adjusted to whatever is most appropriate for the program. Often, programmers will make dEpsilon a third parameter of IsEqual() so it can be defined on a call-by-call basis.

To do inequality (!=) instead of equality, simply call this function and use the logical NOT operator (!) to flip the result:

```
1  if (!IsEqual(dX, dY))
2      cout << dX << " is not equal to " << dY << endl;
```

3.6 — Logical operators

While relational operators can be used to test whether a particular condition is true or false, they can only test one condition at a time. Often we need to know whether multiple conditions are true at once. For example, to check whether we've won the lottery, we have to compare whether all the numbers we picked match all of the winning numbers. In a lottery with 6 numbers, this would involve 6 comparisons, all of which have to be true. Other times, we need to know whether any one of multiple conditions is true. For example, we may decide to skip work today if we're sick, if we're too tired, or if won the lottery in our previous example. This would involve checking whether any of 3 comparisons is true.

Logical operators provide us with this capability.

C++ provides us with 3 logical operators, one of which you have already seen:

Operator	Symbol	Form	Operation
Logical NOT	!	!x	true if x is false, or false if x is true
Logical AND	&&	x && y	true if both x and y are true, false otherwise
Logical OR	||	x || y	true if either x or y are true, false otherwise

Logical NOT

You have already run across the logical NOT operator in the section on boolean values. We can summarize the effects of logical NOT like so:

Logical NOT (operator !)	
Right operand	**Result**
true	false
false	true

If logical NOT's operand evaluates to true, logical NOT evalutes to false. If logical NOT's operand evaluates to false, logical NOT evaluates to true. In other words, logical NOT flips a boolean value from true to false or vice-versa.

Logical not is often used in conditionals:

```
1 bool bTooLarge = (x > 100); // bTooLarge is true if x > 100
2 if (!bTooLarge)
3     // do something with x
4 else
```

```
5       // print an error
```

One thing to be wary of is that logical NOT has a very high level of precedence. New programmers often make the following mistake:

```
1  int x = 5;
2  int y = 7;
3
4  if (! x == y)
5      cout << "x does not equal y";
5  else
6      cout << "x equals y";
```

This program prints "x equals y"! But x does not equal y, so how is this possible? The answer is that because the logical NOT operator has higher precedence than the equality operator, the expression `! x == y` actually evaluates as `(!x) == y`. Since x is 5, !x evaluates to 0, and 0 == y is false, so the else statement executes!

Note: any non-zero integer value evaluates to *true* when used in a boolean context. Since x is 5, x evaluates to true, and !x evaluates to false (0). Mixing integer and boolean operations like this can be very confusing, and should be avoided!

The correct way to write the above snippet is:

```
1  int x = 5;
2  int y = 7;
3
4  if (!(x == y))
5      cout << "x does not equal y";
5  else
6      cout << "x equals y";
```

This way, x == y will be evaluated first, and then logical NOT will flip the boolean result.

Rule: If logical NOT is intended to operate on the result of other operators, the other operators and their operands need to be enclosed in parenthesis.

Simple uses of logical NOT, such as `if (!bValue)` do not need parenthesis because precedence does not come into play.

Logical OR

The logical OR operator is used to test whether either of two conditions is true. If the left operand evaluates to true, or the right operand evaluates to true, the logical OR operator returns true. If both operands are true, then logical OR will return true as well.

Logical OR (operator ||)

Left operand	Right operand	Result
false	false	false
false	true	true
true	false	true
true	true	true

For example, consider the following program:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "Enter a number: ";
7      int nValue;
8      cin >> nValue;
9
10     if (nValue == 0 || nValue == 1)
11         cout << "You picked 0 or 1" << endl;
12     else
       cout << "You did not pick 0 or 1" << endl;
       return 0;
   }
```

In this case, we use the logical OR operator to test whether either the left condition (nValue == 0) or the right condition (nValue == 1) is true. If either (or both) are true, the logical OR operator evaluates to true, which means the if statement executes. If neither are true, the logical OR operator evaluates to false, which means the else statement executes.

You can string together many logical OR statements:

```
1  if (nValue == 0 || nValue == 1 || nValue == 2 || nValue == 3)
2      cout << "You picked 0, 1, 2, or 3" << endl;
```

New programmers sometimes confuse the logical OR operator (||) with the bitwise OR operator (|). Even though they both have OR in the name, they perform different functions. Mixing them up can lead to incorrect results.

Logical AND

The logical AND operator is used to test whether both conditions are true. If both conditions are true, logical AND returns true. Otherwise, it returns false.

Logical AND (operator &&)		
Left operand	**Right operand**	**Result**

false	false	false
false	true	false
true	false	false
true	true	true

For example, we might want to know if the value of variable x is between 10 and 20. This is actually two conditions: we need to know if x is greater than 10, and also whether x is less then 20.

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "Enter a number: ";
7      int nValue;
8      cin >> nValue;
9
10     if (nValue > 10 && nValue < 20)
11         cout << "Your value is between 10 and 20" << endl;
12     else
13         cout << "You value is not between 10 and 20" << endl;
14     return 0;
15 }
```

In this case, we use the logical AND operator to test whether the left condition (nValue > 10) AND the right condition (nValue < 20) are both true. If both are true, the logical AND operator evaluates to true, and the if statement executes. If neither are true, or only one is true, the logical AND operator evaluates to false, and the else statement executes.

As with logical OR, you can string together many logical AND statements:

```
1  if (nValue1 == 0 && nValue2 == 1 && nValue3 == 4 && nValue4 == 6)
2      // do something
3  else
4      // do something else
```

If all of these conditions are true, the if statement will execute. If any of these conditions are false, the else statement will execute.

Short circuit evaluation

In order for logical AND to return true, both operands must evaluate to true. If the first operand evaluates to false, logical AND knows it must return false regardless of whether the second operand evaluates to true or false. In this case, the logical AND operator will go ahead and return false immediately without even evaluating the second operand! This is known as **short circuit evaluation**, and it is done primary for optimization purposes.

Similarly, if the first operand for logical OR is true, then the entire OR condition has to evaluate to true, and the second operand does not need to be evaluated.

Short circuit evaluation presents another opportunity to show why operators that cause side effects should not be used in compound expressions. Consider the following snippet:

```
1  if (x == 1 && y++ == 2)
2      // do something
```

if x does not equal 1, y++ never gets evaluated! Thus, y will only be incremented if x evaluates to 1, which is probably not what the programmer intended!

New programmers sometimes confuse the logical AND operator (&&) with the bitwise AND operator (&). Even though they both have AND in the name, they perform different functions. Mixing them up can lead to incorrect results.

Mixing ANDs and ORs

Mixing logical AND and logical OR operators in the same expression often can not be avoided, but it is an area full of potential dangers.

Many programmers assume that logical AND and logical OR have the same precedence (or forget that they don't), just like addition/subtraction and multiplication/division do. However, logical AND has higher precedence than logical OR, thus logical AND operators will be evaluated ahead of logical OR operators (unless they have been parenthesized).

As a result of this, new programmers will often write expressions such as `nValue1 || nValue2 && nValue3`. Because logical AND has higher precedence, this evaluates as `nValue1 || (nValue2 && nValue3)`, not `(nValue1 || nValue2) && nValue3`. Hopefully that's what the programmer wanted! If the programmer was assuming left to right evaluation (as happens with addition/subtraction, or multiplication/division), the programmer will get a result he or she was not expecting!

When mixing logical AND and logical OR in the same expression, it is a good idea to explicitly parenthesize each operator and it's operands. This helps prevent precedence mistakes, makes your code easier to read, and clearly defines how you intended the expression to evaluate. For example, rather than writing `nValue1 && nValue2 || nValue3 && nValue4`, it is better to write `(nValue1 && nValue2) || (nValue3 && nValue4)`. This makes it clear at a glance how this expression will evaluate.

De Morgan's law

Many programmers also make the mistake of thinking that `!(x && y)` is the same thing as `!x && !y`. Unfortunately, you can not "distribute" the logical NOT in that manner.

De Morgan's law tells us how the logical NOT should be distributed in these cases:

`!(x && y)` is equivalent to `!x || !y`
`!(x || y)` is equivalent to `!x && !y`

In other words, when you distribute the logical NOT, you also need to flip logical AND to logical OR, and vice-versa!

This can sometimes be useful when trying to make up complex expressions easier to read.

Quiz

Evaluate the following:
1) (true && true) || false
2) (false && true) || true
3) (false && true) || false || true
4) (5 > 6 || 4 > 3) && (7 > 8)
5) !(7 > 6 || 3 > 4)

Quiz answers

1) answer

```
(true && true) || false == true || false == true
```

2) answer

```
(false && true) || true == false || true == true
```

3) answer

```
(false && true) || false || true == false || false || true == false || true == true
```

4) answer

```
(5 > 6 || 4 > 3) && (7 > 8) == (false || true) && false == true && false == false
```

5) answer

```
!(7 > 6 || 3 > 4) == !(true || false) == !true == false
```

3.7 — Converting between binary and decimal

In order to understand the bit manipulation operators, it is first necessary to understand how integers are represented in binary. Consider a normal decimal number, such as 5623. We intuitively understand that these digits mean (5 * 1000) + (6 * 100) + (2 * 10) + (3 * 1). Because there are 10 decimal numbers, the value of each digit increases by a factor of 10.

Binary numbers work the same way, except because there are only 2 binary numbers (0 and 1), the value of each digit increases by a factor of 2. Just like commas are often used to make a large decimal number easy to read (eg. 1,427,435), we often write binary numbers in groups of 4 bits to make them easier to read.

Converting binary to decimal

In the following examples, we assume that we're dealing with unsigned numbers.

Consider the 8 bit (1 byte) binary number 0101 1110. 0101 1110 means (0 * 128) + (1 * 64) + (0 * 32) + (1 * 16) + (1 * 8) + (1 * 4) + (1 * 2) + (0 * 1). If we sum up all of these parts, we get the decimal number 64 + 16 + 8 + 4 + 2 = 94.

Here is the same process in table format. We multiple each binary digit by it's bit value (determined by it's position). Summing up all these values gives us the total.

Binary digit	0	1	0	1	1	1	1	0
* Bit value	128	64	32	16	8	4	2	1
= Total (94)	0	64	0	16	8	4	2	0

Let's convert 1001 0111 to decimal:

Binary digit	1	0	0	1	0	1	1	1
* Bit value	128	64	32	16	8	4	2	1
= Total (151)	128	0	0	16	0	4	2	1

1001 0111 binary = 151 in decimal.

This can easily be extended to 16 or 32 bit binary numbers simply by adding more columns.

Converting decimal to binary

Converting from decimal to binary is a little more tricky, but still pretty straightforward. The easiest way to do this is to work backwards to figure out what each of the bits must be.

Consider the decimal number 148.

Is 148 >= 128? Yes, so the 128 bit must be 1. 148 – 128 = 20, which means we need to find bits worth 20 more.
Is 20 >= 64? No, so the 64 bit must be 0.
Is 20 >= 32? No, so the 32 bit must be 0.
Is 20 >= 16? Yes, so the 16 bit must be 1. 20 – 16 = 4, which means we need to find bits worth 4 more.

Is 4 >= 8? No, so the 8 bit must be 0.
Is 4 >= 4? Yes, so the 4 bit must be 1. 4 – 4 = 0, which means all the rest of the bits must be 0.

148 = (1 * 128) + (0 * 64) + (0 * 32) + (1 * 16) + (0 * 8) + (1 * 4) + (0 * 2) + (0 * 1) = 1001 0100

In table format:

Binary number	1	0	0	1	0	1	0	0
* Bit value	128	64	32	16	8	4	2	1
= Total (148)	128	0	0	16	0	4	0	0

Let’s convert 117 to binary:

Is 117 >= 128? No, so the 128 bit must be 0.
Is 117 >= 64? Yes, so the 64 bit must be 1. 117 – 64 = 53.
Is 53 >= 32? Yes, so the 32 bit must be 1. 53 – 32 = 21.
Is 21 >= 16? Yes, so the 16 bit must be 1. 21 – 16 = 5.

Is 5 >= 8? No, so the 8 bit must be 0.
Is 5 >= 4? Yes, so the 4 bit must be 1. 5 – 4 = 1.
Is 1 >= 2? No, so the 2 bit must be 0.
Is 1 >= 1? Yes, so the 1 bit must be 1.

117 decimal = 0111 0101 binary.

Signed numbers

The following section is optional. Most of the time when we deal with binary numbers and bit operations, we use unsigned numbers. However, it is interesting to examine how signed numbers are dealt with.

Signed numbers are typically stored using a method known as **two's complement**. In two's complement, the leftmost (most significant) bit is used as the sign bit. A 0 bit means the number is positive, and a 1 bit means the number is negative. Positive signed numbers are stored just like positive unsigned numbers. Negative signed numbers are stored as the inverse of the positive number, plus 1.

For example, here's how we convert -5 to binary:

First we figure out the binary representation for 5: 0000 0101
Then we invert all of the bits: 1111 1010
Then we add 1: 1111 1011

Converting -76 to binary:

Positive 76 in binary: 0100 1100
Invert all the bits: 1011 0011
Add 1: 1011 0100

Why do we add 1? Consider the number 0. If a negative value was simply represented as the inverse of the positive number, 0 would have two representations: 0000 0000 (positive zero) and 1111 1111 (negative zero). By adding 1, 1111 1111 intentionally overflows and becomes 0000 0000. This prevents 0 from having two representations, and simplifies some of the internal logic needed to deal with negative numbers.

Quiz

1) Convert 0100 1101 to decimal.
2) Convert 93 to an 8-bit binary number.

Quiz answers

1) answer

Binary digit	0	1	0	0	1	1	0	1
* Bit value	128	64	32	16	8	4	2	1
= Total (77)	0	64	0	0	8	4	0	1

The answer is 77.

2) answer

Is 93 >= 128? No, so the 128 bit is 0.
Is 93 >= 64? Yes, so the 64 bit is 1. 93 – 64 = 29.
Is 29 >= 32? No, so the 32 bit is 0.
Is 29 >= 16? Yes, so the 16 bit is 1. 29 – 16 = 13.

Is 13 >= 8? Yes, so the 8 bit is 1. 13 – 8 = 5.
Is 5 >= 4? Yes, so the 4 bit is 1. 5 – 4 = 1.
Is 1 >= 2? No, so the 2 bit is 0.
Is 1 >= 1? Yes, so the 1 bit is 1.

The answer is 0101 1101.

3.8 — Bitwise operators

Bit manipulation operators manipulate individual bits within a variable.

Why bother with bitwise operators?

In the past, memory was extremely expensive, and computers did not have much of it. Consequently, there were incentives to make use of every bit of memory available. Consider the bool data type — even though it only has two possible values (true and false), which can be represented by a single bit, it takes up an entire byte of memory! This is because variables need unique addresses, and memory can only be addressed in bytes. The bool uses 1 bit and the other 7 go to waste.

Using bitwise operators, it is possible to write functions that allow us to compact 8 booleans into a single byte-sized variable, enabling significant memory savings at the expense of more complex code. In the past, this was a good trade-off. Today, it is not.

Now memory is significantly cheaper, and programmers have found that it is often a better idea to code what is easiest to understand and maintain than what is most efficient. Consequently, bitwise operators have somewhat fallen out of favor, except in certain circumstances where maximum optimization is needed (eg. scientific programs that use enormous data sets, or games where bit manipulation tricks can be used for extra speed). Nevertheless, it is good to at least know about their existence.

There are 6 bit manipulation operators:

Operator	Symbol	Form	Operation
left shift	<<	x << y	all bits in x shifted left y bits
right shift	>>	x >> y	all bits in x shifted right y bits
bitwise NOT	~	~x	all bits in x flipped
bitwise AND	&	x & y	each bit in x AND each bit in y
bitwise OR	|	x | y	each bit in x OR each bit in y
bitwise XOR	^	x ^ y	each bit in x XOR each bit in y

Note: In the following examples, we will largely be working with 4-bit binary values. This is for the sake of convenience and keeping the examples simple. In C++, the number of bits used will be based on the size of the data type (8 bits per byte).

Left shift and right shift operator

The bitwise left shift (<<) shifts operator bits to the left. For example, consider the number 3, which is binary 0011. Left shifting by 1 (3 << 1) changes 0011 to 0110, which is decimal 6. Note how each bit moved 1 place to the left. Left shifting by 2 (3 << 2) changes 0011 to 1100, which is decimal 12. Left shifting by 3 (3 << 3) changes 0011 to 1000. Note that we shifted a bit off the end of the number! Bits that are shifted off the end of the binary number are lost.

The bitwise right shift (>>) operator shifts bits to the right. Right shifting by 1 (3 >> 1) changes 0011 to 0001, or decimal 1. The rightmost bit shifted off the end and was lost!

Although our examples above are shifting literals, you can shift variables as well:

```
1 unsigned int nValue = 4;
2 nValue = nValue << 1; // nValue will be 8
```

Rule: When dealing with bit operators, use unsigned variables.

Programs today typically do not make much use of the bitwise left and right shift operator in this capacity. Rather, you tend to see the bitwise left shift operator used with cout in a way that doesn't involve shifting bits at all! If << is a bitwise left shift operator, then how does `cout << "Hello, world!";` print to the screen? The answer is that cout has **overridden** (replaced) the default meaning of the << operator and given it a new meaning. We will talk more about operator overloading in a future section!

Bitwise NOT

The bitwise NOT operator (~) is perhaps the easiest to understand of all the bitwise operators. It simply flips each bit from a 0 to a 1, or vice versa. Note that the result of a bitwise NOT is dependent on what size your data type is! For example, with 4 bits, ~4 (0100 binary) evaluates to 1011, which is 11 in decimal. In an 8 bit data type (such as an unsigned char), ~4 (represented as ~0000 0100) evaluates to 1111 1011, which is 251 in decimal!

Bitwise AND, OR, and XOR

Bitwise AND (&) and bitwise OR (|) work similarly to their logical AND and logical OR counterparts. However, rather than evaluating a single boolean value, they are applied to each bit! For example, consider the expression `5 | 6`. In binary, this is represented as 0101 | 0110. To do (any) bitwise operations, it is easiest to line the two operands up like this:

```
0 1 0 1 // 5
0 1 1 0 // 6
```

and then apply the operation to each *column* of bits. If you remember, logical OR evaluates to true (1) if either the left or the right or both operands are true (1). Bitwise OR evaluates to 1 if either bit (or both) is 1. Consequently, 5 | 6 evaluates like this:

```
0 1 0 1 // 5
0 1 1 0 // 6
```

```
-------
0 1 1 1 // 7
```

Our result is 0111 binary (7 decimal).

We can do the same thing to compound OR expressions, such as `1 | 4 | 6`. If any of the bits in a column are 1, the result of that column is 1.

```
0 0 0 1 // 1
0 1 0 0 // 4
0 1 1 0 // 6
--------
0 1 1 1 // 7
```

1 | 4 | 6 evaluates to 7.

Bitwise AND works similarly. Logical AND evaluates to true if both the left and right operand evaluate to true. Bitwise AND evaluates to true if both bits in the column are 1) Consider the expression `5 & 6`. Lining each of the bits up and applying an AND operation to each column of bits:

```
0 1 0 1 // 5
0 1 1 0 // 6
--------
0 1 0 0 // 4
```

Similarly, we can do the same thing to compound AND expressions, such as `1 & 3 & 7`. If all of the bits in a column are 1, the result of that column is 1.

```
0 0 0 1 // 1
0 0 1 1 // 3
0 1 1 1 // 7
--------
0 0 0 1 // 1
```

The last operator is the bitwise XOR (^), also known as *exclusive or*. When evaluating two operands, XOR evaluates to true (1) if one *and only one* of it's operands is true (1). If neither or both are true, it evaluates to 0. Consider the expression `6 ^ 3`:

```
0 1 1 0 // 6
0 0 1 1 // 3
-------
0 1 0 1 // 5
```

It is also possible to evaluate compound XOR expression column style, such as `1 ^ 3 ^ 7`. If there are an even number of 1 bits in a column, the result is 0. If there are an odd number of 1 bits in a column, the result is 1.

```
0 0 0 1 // 1
0 0 1 1 // 3
```

```
0 1 1 1 // 7
--------
0 1 0 1 // 5
```

Bitwise assignment operators

As with the arithmetic assignment operators, C++ provides bitwise assignment operators in order to facilitate easy modification of variables.

Operator	Symbol	Form	Operation
Left shift assignment	<<=	x <<= y	Shift x left by y bits
Right shift assignment	>>=	x >>= y	Shift x right by y bits
Bitwise OR assignment	|=	x |= y	Assign x | y to x
Bitwise AND assignment	&=	x &= y	Assign x & y to x
Bitwise XOR assignment	^=	x ^= y	Assign x ^ y to x

For example, instead of writing `nValue = nValue << 1;`, you can write `nValue <<= 1;`.

Summary

Summarizing how to evaluate bitwise operations utilizing the column method:

When evaluating bitwise OR, if any bit in a column is 1, the result for that column is 1.
When evaluating bitwise AND, if all bits in a column are 1, the result for that column is 1.
When evaluating bitwise XOR, if there are an odd number of 1 bits in a column, the result for that column is 1.

Quiz

1) What does 0110 >> 2 evaluate to in binary?
2) What does 5 | 6 evaluate to in decimal?
3) What does 5 & 6 evaluate to in decimal?
4) What does 5 ^ 6 evaluate to in decimal?

Quiz answers

1) answer

0110 >> 2 evaluates to 0001

2) answer

5 | 6 =
0 1 0 1

0 1 1 0
----
0 1 1 1 = 7

3) answer

5 & 6 =
0 1 0 1
0 1 1 0
----
0 1 0 0 = 4

4) answer

5 ^ 6 =
0 1 0 1
0 1 1 0
----
0 0 1 1 = 3

3.x — Comprehensive Quiz

Quick Review

Always use parenthesis to disambiguate the precedence of operators if there is any question or opportunity for confusion.

The arithmetic operators all work like they do in normal mathematics. The Modulus (%) operator returns the remainder from an integer division. Beware about rounding or sign errors when the operands of integer division and modulus are negative.

The increment and decrement operators can be used to easily increment or decrement numbers. Beware of side effects, particular when it comes to the order that function parameters are evaluated.

Relational operators can be used to compare floating point numbers. Beware using equality and inequality on floating point numbers.

Comprehensive quiz

1) Evaluate the following:

a) (5 > 3 && 4 < 8)
b) (4 > 6 && true)
c) (3 >= 3 || false)
d) (true || false) ? 4 : 5

2) Answer the following:

a) 7 / 4
b) 14 % 5

3) Convert the following from binary to decimal:
a) 1101
b) 101110

4) Convert the following from decimal to binary:
a) 15
b) 53

5) Why should you never do the following:
a) int y = foo(++x, x);
b) int x = 7 / -2;

c) int x = -5 % 2;
d) float x = 0.1 + 0.1; if (x == 0.2) return true; else return false;
e) int x = 3 / 0;

Solutions

1) answer

a) (5 > 3 && 4 < 8) becomes (true && true), which is true.
b) (4 > 6 && true) becomes (false && true), which is false.
c) (3 >= 3 || false) becomes (true || false), which is true.
d) (true || false) ? 4 | 5 becomes (true ? 4 : 5), which is 4.

2) answer

a) 7 / 4 = 1 remainder 3, so this equals 1.
b) 14 % 5 = 2 remainder 4, so this equals 4.

3) answer

a) 1101 is ((1 * 1) + (0 * 2) + (1 * 4) + (1 * 8)) = (1 + 0 + 4 + 8) = 13
b) 101110 is ((0 * 1) + (1 * 2) + (1 * 4) + (1 * 8) + (0 * 16) + (1 * 32)) = (0 + 2 + 4 + 8 + 0 + 32) = 46

4) answer

a) Is 15 >= 8? Yes, with 7 left over. Is 7 >= 4? Yes, with 3 left over. Is 3 >= 2? Yes, with 1 left over. Is 1 >= 1? Yes. So this number is 1111 in binary.
b) Is 53 >= 32? Yes, with 21 left over. Is 21 >= 16? Yes, with 5 left over. Is 5 >= 8? No. Is 5 >= 4? Yes, with 1 left over. Is 1 > 2? No. Is 1 >=1? Yes. So this number is 110101 in binary.

5) answer

a) The parameters to function foo() can execute in any order, so it's indeterminate whether x or ++x gets evaluated first. Because ++x change the value of x, it's unclear what values will be passed into the function.
b) It's unclear whether the compiler will round this up to -3 or down to -4.
c) It's unclear whether this will result in 1 or -1.
d) Floating point errors will cause this to evaluate as false even though it looks like it should be true.
e) Divide by 0 will crash the program.

Variables Part II

4.1 — Blocks (compound statements) and local variables

Blocks (compound statements)

A block of statements, also called a **compound statement**, is a group of statements that is treated by the compiler as if it were a single statement. Blocks begin with a { symbol, end with a } symbol, and the statements to be executed are placed in between. Blocks can be used any place where a single statement is allowed.

You have already seen an example of a block when writing the function main():

```
1  int main()
2  { // start a block
3
4      // multiple statements
5      int nValue = 0;
6      return 0;
7
   } // end a block
```

Blocks can be nested inside of other blocks. As you have seen, the *if statement* executes a single statement if the condition is true. However, because blocks can be used anywhere a single statement can, we can instead use a block of statements to make the *if statement* execute multiple statements if the condition is true!

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "Enter a number: ";
7      int nValue;
8      cin >> nValue;
9
10     if (nValue > 0)
11     { // start of nested block
12         cout << nValue << " is a positive number" << endl;
           cout << "Double this number is " << nValue * 2 << endl;
       } // end of nested block
   }
```

If the users enters the number 3, this program prints:

```
3 is a positive number
Double this number is 6
```

Note that both statements inside the nested block executed when the if statement is true!

It is even possible to put blocks inside of blocks inside of blocks:

```
1  int main()
2  {
3      using namespace std;
4      cout << "Enter a number: ";
5      int nValue;
6      cin >> nValue;
7
8      if (nValue > 0)
9      {
10         if (nValue < 10)
11         {
12             cout << nValue << " is between 0 and 10" << endl;
           }
       }
   }
```

There is no practical limit to how many nested blocks you can have. However, it is generally a good idea to try to keep the number of nested blocks to at most 3 (maybe 4) blocks deep. If your function has a need for more, it's probably time to break your function into multiple smaller functions!

Local variables

A variable's **scope** determines who can see the variable, and how long it lives for. Variables declared inside a block are called **local variables**, and local variables have **block scope** (also called local scope). Variables with block scope can be accessed only within the block that they are declared in, and are destroyed as soon as the block ends. Consider this simple function:

```
1  int main()
2  { // start main block
3
4      int nValue = 5; // nValue created here
5
6      double dValue = 4.0; // dValue created here
7
8      return 0;
9
   } // nValue and dValue destroyed here
```

Because nValue and dValue were declared inside the block that defines the main function, they are both destroyed when main() is finished executing.

Variables declared inside a block can only be seen within that block. Because each function has it's own block, variables in one function can not be seen from another function:

```
1  void someFunction()
```

```
{
    int nValue;
}

int main()
{
    // nValue can not be seen inside this function.

    someFunction();

    // nValue still can not be seen inside this function.

    return 0;
}
```

Variables declared inside nested blocks are destroyed as soon as the inner block ends:

```
int main()
{
    int nValue = 5;

    { // begin nested block
        double dValue = 4.0;
    } // dValue destroyed here

    // dValue can not be used here because it was already destroyed!

    return 0;
} // nValue destroyed here
```

Nested blocks are considered part of the outer block in which they are defined. Consequently, variables declared in the outer block can be seen inside a nested block:

```
int main()
{ // start outer block
    using namespace std;

    int x = 5;

    { // start nested block
        int y = 7;
        // we can see both x and y from here
        cout << x << " + " << y << " = " << x + y;
    } // y destroyed here

    // y can not be used here because it was already destroyed!

    return 0;
} // x is destroyed here
```

Note that variables inside nested blocks can have the same name as variable inside outer blocks. When this happens, the nested variable “hides” the outer variable:

```
1  int main()
2  { // outer block
3      int nValue = 5;
4
5      if (nValue >= 5)
6      { // nested block
7          int nValue = 10;
8          // nValue now refers to the nested block nValue.
9          // the outer block nValue is hidden
10     } // nested nValue destroyed
11
12     // nValue now refers to the outer block nValue
13
14     return 0;
15 } // outer nValue destroyed
```

This is generally something that should be avoided, as it is quite confusing!

Variables should be declared in the most limited scope in which they are used. For example, if a variable is only used within a nested block, it should be declared inside that nested block:

```
1  int main()
2  {
3      // do not declare y here
4      {
5          // y is only used inside this block, so declare it here
6          int y = 5;
7          cout << y;
8      }
9      // otherwise y could still be used here
10 }
```

By limiting the scope of a variable, you reduce the complexity of the program because the number of active variables is reduced. Further, it makes it easier to see where variables are used. A variable declared inside a block can only be used within that block (or nested sub-blocks). This can make the program easier to understand.

Summary

Blocks allow multiple statements to be used wherever a single statement can normally be used.

Variables declared inside blocks are called local variables. These variables can only be accessed inside the block in which they are defined (or in a nested sub-block), and they are destroyed as soon as the block ends.

If a variable is only used in a single block, declare it within that block.

4.2 — Global variables

In the last lesson, you learned that variables declared inside a block have block scope. Block scope variables can only be accessed within the block in which they are declared (or a nested sub-block), and are destroyed when the block ends.

Variables declared outside of a block are called **global variables**. Global variables have **program scope**, which means they can be accessed everywhere in the program, and they are only destroyed when the program ends.

Here is an example of declaring a global variable:

```
1  int g_nX; // global variable
2
3  int main()
4  {
5      int nY; // local variable nY
6
7      // global vars can be seen everywhere in program
8      // so we can change their values here
9      g_nX = 5;
   } // nY is destroyed here
```

Because global variables have program scope, they can be used across multiple files. In the section on programs with multiple files, you learned that in order to use a function declared in another file, you have to use a forward declaration, or a header file.

Similarly, in order to use a global variable that has been declared in another file, you have to use a forward declaration or a header file, along with the **extern** keyword. Extern tells the compiler that you are not declaring a new variable, but instead referring to a variable declared elsewhere.

Here is an example of using a forward declaration style extern:

global.cpp:

```
1  // declaration of g_nValue
2  int g_nValue = 5;
```

main.cpp:

```
1  // extern tells the compiler this variable is declared elsewhere
2  extern int g_nValue;
3
4  int main()
   {
```

```
    g_nValue = 7;
    return 0;
}
```

Here is an example of using a header file extern:

global.cpp:

```
// declaration of g_nValue
int g_nValue = 5;
```

global.h:

```
#ifndef GLOBAL_H // header guards
#define GLOBAL_H

// extern tells the compiler this variable is declared elsewhere
extern int g_nValue;

#endif
```

main.cpp:

```
#include "global.h"
int main()
{
    g_nValue = 7;
    return 0;
}
```

Generally speaking, if a global variable is going to be used in more than 2 files, it's better to use the header file approach. Some programmers place all of a programs global variables in a file called globals.cpp, and create a header file named globals.h to be included by other .cpp files that need to use them.

Local variables with the same name as a global variable hide the global variable inside that block. However, the global scope operator (::) can be used to tell the compiler you mean the global version:

```
int nValue = 5;

int main()
{
    int nValue = 7; // hides the global nValue variable
    nValue++; // increments local nValue, not global nValue
    ::nValue--; // decrements global nValue, not local nValue
    return 0;
} // local nValue is destroyed
```

However, having local variables with the same name as global variables is usually a recipe for trouble, and should be avoided whenever possible. Using Hungarian Notation, it is common to declare global variables with a "g_" prefix. This is an easy way to differentiate global variable from local variables, and avoid variables being hidden due to naming collisions.

New programmers are often tempted to use lots of global variables, because they are easy to work with, especially when many functions are involved. However, this is a very bad idea. In fact, global variables should generally be avoided completely!

Why global variables are evil

Global variables should be avoided for several reasons, but the primary reason is because they increase your program's complexity immensely. For example, say you were examining a program and you wanted to know what a variable named g_nValue was used for. Because g_nValue is a global, and globals can be used anywhere in the entire program, you'd have to examine every single line of every single file! In a computer program with hundreds of files and millions of lines of code, you can imagine how long this would take!

Second, global variables are dangerous because their values can be changed by any function that is called, and there is no easy way for the programmer to know that this will happen. Consider the following program:

```
1  // declare global variable
2  int g_nMode = 1;
3
4  void doSomething()
5  {
6      g_nMode = 2;
7  }
8
9  int main()
10 {
11     g_nMode = 1;
12
13     doSomething();
14
15     // Programmer expects g_nMode to be 1
16     // But doSomething changed it to 2!
17
18     if (g_nMode == 1)
19         cout << "No threat detected." << endl;
20     else
21         cout << "Launching nuclear missiles..." << endl;
22
23     return 0;
24 }
```

Note that the programmer set g_nMode to 1, and then called doSomething(). Unless the programmer had explicit knowledge that doSomething() was going to change the value of

g_nMode, he or she was probably not expecting doSomething() to change the value! Consequently, the rest of main() doesn't work like the programmer expects (and the world is obliterated).

Global variables make every function call potentially dangerous, and the programmer has no easy way of knowing which ones are dangerous and which ones aren't! Local variables are much safer because other functions can not affect them directly. Consequently, global variables should not be used unless there is a very good reason!

4.3 — File scope and the static keyword

In previous lessons, you learned about local variables (which have block scope) and global variables (which have program scope). There is one other level of scoping that variables can have: file scope.

A variable with **file scope** can be accessed by any function or block within a single file. To declare a file scoped variable, simply declare a variable outside of a block (same as a global variable) but use the **static** keyword:

```
1  static int nValue; // file scoped variable
2  float fValue; // global variable
3
4  int main()
5  {
6      double dValue; // local variable
7  }
```

File scoped variables act exactly like global variables, except their use is restricted to the file in which they are declared (which means you can not extern them to other files). File scoped variables are not seen very often in C++ because they have most of the downsides of global variables, just on a smaller scale.

The static keyword

The static keyword is probably the most confusing keyword in the C++ language. This is because it has different meanings depending on where it is used. When applied to a variable declared outside of a block, it changes the variable from a global variable to a file scoped variable. When applied to a variable declared inside a block, it has a different meaning entirely!

By default, local variables have **automatic duration**, which means they are destroyed when the block they are declared in goes out of scope. You can explicitly declare a variable as having automatic duration by using the **auto** keyword, though this is practically never done because local variables are automatic by default, and it would be redundant.

Using the static keyword on local variables changes them from automatic duration to fixed duration (also called static duration). A **fixed duration** variable is one that retains it's value even after the scope in which it has been created has been exited! Fixed duration variables are only created (and initialized) once, and then they are persisted throughout the life of the program.

The easiest way to show the difference between automatic and fixed duration variables is by example.

Automatic duration (default):

```
1  #include <iostream>
2
3  void IncrementAndPrint()
4  {
5      using namespace std;
6      int nValue = 1; // automatic duration by default
7      ++nValue;
8      cout << nValue << endl;
9  } // nValue is destroyed here
10
11 int main()
12 {
13     IncrementAndPrint();
14     IncrementAndPrint();
15     IncrementAndPrint();
16 }
```

Each time IncrementAndPrint is called, a variable named nValue is created and assigned the value of 1. IncrementAndPrint increments nValue to 2, and then prints the value of 2. When IncrementAndPrint is finished running, the variable goes out of scope and is destroyed. Consequently, this program outputs:

```
2
2
2
```

Now consider the fixed scope version of this program. The only difference between this and the above program is that we've changed the local variable nValue from automatic to fixed duration by using the static keyword.

Fixed duration (using static keyword):

```
1  #include <iostream>
2
3  void IncrementAndPrint()
4  {
5      using namespace std;
6      static int s_nValue = 1; // fixed duration
7      ++s_nValue;
8      cout << s_nValue << endl;
9  } // s_nValue is not destroyed here, but becomes inaccessible
10
11 int main()
12 {
13     IncrementAndPrint();
14     IncrementAndPrint();
15     IncrementAndPrint();
16 }
```

In this program, because s_nValue has been declared as static, s_nValue is only created and initialized (to 1) once. When it goes out of scope, it is not destroyed. Each time the function IncrementAndPrint() is called, the value of s_nValue is whatever we left it at previously. Consequently, this program outputs:

```
2
3
4
```

Using hungarian notation, it is common to prefix fixed duration variables with "s_". Some programmers use "s" (which we don't like as much because that letter is better used for structs) or "c_" (which we don't like as much because it's not as mnemonic).

One of the most common uses for fixed duration local variables is for unique identifier generators. When dealing with a large number of similar objects within a program, it is often useful to assign each one a unique ID number so they can be identified. This is very easy to do with a fixed duration local variable:

```
1  int GenerateID()
2  {
3      static int nNextID = 0;
4      return nNextID++;
   }
```

The first time this function is called, it returns 0. The second time, it returns 1. Each time it is called, it returns a number one higher than the previous time it was called. You can assign these numbers as unique IDs for your objects. Because nNextID is a local variable, it can not be "tampered with" by other functions.

4.4 — Type conversion and casting

Previously, you learned that the value of a variable is stored as a sequence of bits, and the data type of the variable tells the compiler how to translate those bits into meaningful values. Often it is the case that data needs to be converted from one type to another type. This is called **type conversion**.

Implicit type conversion is done automatically by the compiler whenever data from different types is intermixed. When a value from one type is assigned to another type, the compiler implicitly converts the value into a value of the new type. For example:

```
1 double dValue = 3; // implicit conversion to double value 3.0
2 int nValue = 3.14156; // implicit conversion to integer value 3
```

In the top example, the value 3 is promoted to a double value and then assigned to dValue. The compiler will not complain about doing this. However, some type conversions are inherently unsafe, and if the compiler can detect that an unsafe conversion is being implicitly requested, it will issue a warning. In the second example, the fractional part of the double value is dropped because integers can not support fractional values. Because converting a double to an int usually causes data loss (making it unsafe), compilers such as Visual Studio Express 2005 will typically issue a warning. Other unsafe conversions involve assigning unsigned variables to signed variables (and vice-versa), and assigning large integers (eg. a 4-byte long) to integer variables of a smaller size (eg. a 2-byte short).

Warning: Microsoft's Visual C++ 2005 does not seem to issue warnings for unsafe signed/unsigned conversions.

When evaluating expressions, the compiler breaks each expression down into individual subexpressions. Typically, these subexpressions involve a unary or binary operator and some operands. Most binary operators require their operands to be of the same type. If operands of mixed types are used, the compiler will convert one operand to agree with the other. To do this, it uses a heirarchy of data types:

Long double (highest)
Double
Float
Unsigned long int
Long int
Unsigned int
Int (lowest)

For example, in the expression `2 + 3.14159`, the + operator requires both operands to be the same type. In this case, the left operand is an int, and the right operand is a double. Because

double is higher in the heirarchy, the int gets converted to a double. Consequently, this expression is evaluated as `2.0 + 3.14159`, which evaluates to 5.14159.

A good question is, "why is integer at the bottom of the tree? What about char and short?". Char and short are always implicitly promoted to integers (or unsigned integers) before evaluation. This is called **widening**.

This heirarchy can cause some interesting issues. For example, you might expect the expression `5u - 10` to evalute to -5 (5u means 5 as an unsigned integer). But in this case, the signed integer (10) is promoted to an unsigned integer, and the result of this expression is the unsigned integer 4294967291!

Many mixed conversion work as expected. For example, `int nValue = 10 * 2.7` yields the result 27. 10 is promoted to a float, 10.0 * 2.7 evaluates to 27.0, and 27.0 is truncated into an integer (which the compiler will complain about).

Many new programmers try something like this: `float fValue = 10 / 4;`. However, because 10 and 4 are both integers, no promotion takes place. Integer division is performed on 10 / 4, resulting in the value of 2, which is then implicitly converted to 2.0 and assigned to fValue!

In the case where you are using literal values (such as 10, or 4), replacing one or both of the integer literal value with a floating point literal value (10.0 or 4.0) will cause both operands to be converted to floating point values, and the division will be done using floating point math.

But what if you are using variables? Consider this case:

```
1  int nValue1 = 10;
2  int nValue2 = 4;
3  float fValue = nValue1 / nValue2;
```

fValue will end up with the value of 2. How do we tell the compiler that we want to use floating point division instead of integer division? The answer is by using a cast.

Casting

Casting represents a request by the programmer to do an explicit type conversion. In standard C programming, casts are done via the () operator, with the name of the type to cast to inside. For example:

```
1  int nValue1 = 10;
2  int nValue2 = 4;
3  float fValue = (float)nValue1 / nValue2;
```

In the above program, we use a float cast to tell the compiler to promote nValue1 to a floating point value. Because nValue1 is a floating point value, nValue2 will then be promoted to a

floating point value as well, and the division will be done using floating point division instead of integer division!

C++ will also let you use a C style cast with a more function-call like syntax:

```
1  int nValue1 = 10;
2  int nValue2 = 4;
3  float fValue = float(nValue1) / nValue2;
```

The C style cast can be inherently misused, because it will let you do things that may not make sense, such as getting rid of a const or changing a data type without changing the underlying representation. C++ introduces a new casting operator called **static_cast**. A static cast works similarly to the C style cast, except it will only do standard type conversions, which reduces the potential for inadvertant misuse:

```
1  int nValue1 = 10;
2  int nValue2 = 4;
3  float fValue = static_cast<float>(nValue1) / nValue2;
```

As mentioned above, compilers will often complain when an unsafe implicit cast is performed. For example, consider the following program:

```
1  int nValue = 48;
2  char ch = nValue; // implicit cast
```

Casting an int (4 bytes) to a char (1 byte) is potentially unsafe, and the compiler will typically complain. In order to announce to the compiler that you are explicitly doing something you recognize is potentially unsafe (but want to do anyway), you should use a static_cast:

```
1  int nValue = 48;
2  char ch = static_cast<char>(nValue);
```

In the following program, the compiler will typically complain that converting a double to an int may result in loss of data:

```
1  int nValue = 100;
2  nValue = nValue / 2.5;
```

To tell the compiler that we explicitly mean to do this:

```
1  int nValue = 100;
2  nValue = static_cast<int>(nValue / 2.5);
```

Casting should be avoided if at all possible, because any time a cast is used, there is potential for trouble. But there are many times when it can not be avoided. In these cases, the C++ static_cast should be used instead of the C-style cast.

4.5 — Enumerated types

C++ allows programmers to create their own data types. Perhaps the simplest method for doing so is via an enumerated type. An **enumerated type** is a data type where every possible value is defined as a symbolic constant (called an **enumerator**). Enumerated types are declared via the **enum** keyword. Let's look at an example:

```
1  // define a new enum named Color
2  enum Color
3  {
4      // Here are the enumerators
5      // These define all the possible values this type can hold
6      COLOR_BLACK,
7      COLOR_RED,
8      COLOR_BLUE,
9      COLOR_GREEN,
10     COLOR_WHITE,
11     COLOR_CYAN,
12     COLOR_YELLOW,
13     COLOR_MAGENTA
14 };
15
16 // Declare a variable of enumerated type Color
17 Color eColor = COLOR_WHITE;
```

Defining an enumerated type does not allocate any memory. When a variable of the enumerated type is declared (such as eColor in the example above), memory is allocated for that variable at that time.

Enum variables are the same size as an int variable. This is because each enumerator is automatically assigned an integer value based on it's position in the enumeration list. By default, the first enumerator is assigned the integer value 0, and each subsequent enumerator has a value one greater than the previous enumerator:

```
1  enum Color
2  {
3      COLOR_BLACK, // assigned 0
4      COLOR_RED, // assigned 1
5      COLOR_BLUE, // assigned 2
6      COLOR_GREEN, // assigned 3
7      COLOR_WHITE, // assigned 4
8      COLOR_CYAN, // assigned 5
9      COLOR_YELLOW, // assigned 6
10     COLOR_MAGENTA // assigned 7
11 };
```

```
10
11 Color eColor = COLOR_WHITE;
12 cout << eColor;
13
14
```

The cout statement above prints the value 4.

It is possible to explicitly define the value of enumerator. These integer values can be positive or negative and can be non-unique. Any non-defined enumerators are given a value one greater than the previous enumerator.

```
1  // define a new enum named Animal
2  enum Animal
3  {
4      ANIMAL_CAT = -3,
5      ANIMAL_DOG, // assigned -2
6      ANIMAL_PIG, // assigned -1
7      ANIMAL_HORSE = 5,
8      ANIMAL_GIRAFFE = 5,
9      ANIMAL_CHICKEN // assigned 6
10 };
```

Because enumerated values evaluate to integers, they can be assigned to integer variables:

```
1 int nValue = ANIMAL_PIG;
```

However, the compiler will not implicitly cast an integer to an enumerated value. The following will produce a compiler error:

```
1 Animal eAnimal = 5; // will cause compiler error
```

It is possible to use a static_cast to force the compiler to put an integer value into an enumerated type, though it's generally bad style to do so:

```
1 Animal eAnimal = static_cast<Animal>(5); // compiler won't complain, but
  bad style
```

Each enumerated type is considered a distinct type. Consequently, trying to assign enumerators from one enum type to another enum type will cause a compile error:

```
1 Animal eAnimal = COLOR_BLUE; // will cause compile error
```

Enumerated types are incredibly useful for code documentation and readability purposes when you need to represent a specific number of states.

For example, functions often return integers to the caller to represent error codes when something went wrong inside the function. Typically, small negative numbers are used to represent different possible error codes. For example:

```
1  int ParseFile()
2  {
3      if (!OpenFile())
4          return -1;
5      if (!ReadFile())
6          return -2;
7      if (!Parsefile())
8          return -3;
9
10     return 0; // success
11 }
```

However, using magic numbers like this isn't very descriptive. An alternative method would be through use of an enumerated type:

```
1  enum ParseResult
2  {
3      SUCCESS = 0,
4      ERROR_OPENING_FILE = -1,
5      ERROR_READING_FILE = -2,
6      ERROR_PARSING_FILE = -3,
7  };
8
9  ParseResult ParseFile()
10 {
11     if (!OpenFile())
12         return ERROR_OPENING_FILE;
13     if (!ReadFile())
14         return ERROR_READING_FILE;
15     if (!Parsefile())
16         return ERROR_PARSING_FILE;
17
18     return SUCCESS;
19 }
```

This is much easier to read and understand than using magic number return values. Furthermore, the caller can test the function's return value against the appropriate enumerator, which is easier to understand than testing the return result for a specific integer value.

```
1  if (ParseFile() == SUCCESS)
2      {
3      // do something
       }
```

```
4  else
5      {
6      // print error message
7      }
8
```

Another use for enums is as array indices, because enumerator indices are more descriptive than integer indices. We will cover this in more detail in the section on arrays.

Finally, as with constant variables, enumerated types show up in the debugger, making them more useful than #defined values in this regard.

Quiz

1) Define an enumerated type to choose between the following monster types: orcs, goblins, trolls, ogres, and skeletons.

2) Declare a variable of the enumerated type you defined in question 1 and assign it the troll type.

3) True or false. Enumerators can be:
3a) explicitly assigned integer values
3b) not explicitly assigned a value
3c) explicitly assigned floating point values
3d) negative
3e) non-unique
3f) assigned the value of prior enumerators (eg. COLOR_MAGENTA = COLOR_RED)

Quiz answers

1) answer

```
1
2 enum MonsterType
3 {
4     MONSTER_ORC,
5     MONSTER_GOBLIN,
6     MONSTER_TROLL,
7     MONSTER_OGRE,
8     MONSTER_SKELETON,
9 };
10
```

2) answer

```
1 MonsterType eMonsterType = MONSTER_TROLL;
```

3) answer

3a) True
3b) True
3c) False
3d) True
3e) True
3f) True. Since enumerators evaluate to integers, and integers can be assigned to enumerators, enumerators can be assigned to other enumerators (though there is typically little reason to do so!).

4.6 — Typedefs

Typedefs allow the programmer to create an alias for a data type, and use the aliased name instead of the actual type name. To declare a typedef, simply use the **typedef** keyword, followed by the type to alias, followed by the alias name:

```
1  typedef long miles; // define miles as an alias for long
2
3  // The following two statements are equivalent:
4  long nDistance;
   miles nDistance;
```

A typedef does not define new type, but is just another name for an existing type. A typedef can be used anywhere a regular type can be used.

Typedefs are used mainly for documentation and legibility purposes. Data type names such as char, int, long, double, and bool are good for describing what type of variable something is, but more often we want to know what purpose a variable serves. In the above example, `long nDistance` does not give us any clue what units nDistance is holding. Is it inches, feet, miles, meters, or some other unit? `miles nDistance` makes it clear what the unit of nDistance is.

This is also true of function return types. Which of the following is easier to decipher?

```
1  typedef int testScore;
2
3  int GradeTest();
   testScore GradeTest();
```

What is the first GradeTest() returning? A grade? The number of questions missed? The student's ID number? An error code? Who knows! Int does not tell us anything. Using a return type of testScore makes it obvious that the function is returning a type that represents a test score.

Furthermore, typedefs allow you to change the underlying type of an object without having to change lots of code. For example, if you were using a short to hold a student's ID number, but then decided you needed a long instead, you'd have to comb through lots of code and replace short with long. It would probably be difficult to figure out which shorts were being used to hold ID numbers and which were being used for other purposes.

However, with a typedef, all you have to do is change `typedef short studentID` to `typedef long studentID` and you're done. Precaution is mandatory when changing the type of a typedef! The new type may have comparison or integer/floating point division issues that the old type did not.

Note that typedefs don't mix particularly well with Hungarian Notation, and allow you to skirt some of the issues that using Hungarian Notation tries to prevent (such as being able to change a variable's type without having to examine the code for areas where changing the type will be problematic).

Because typedefs do not define new types, they can be intermixed like normal data types. Even though the following does not make sense conceptually, syntactically it is valid C++:

```
1  typedef long miles;
2  typedef long speed;
3
4  miles nDistance = 5;
5  speed nMhz = 3200;
6
7  // The following is okay, because nDistance and nMhz are both type long
   nDistance = nMhz;
```

Platform independent coding

One big advantage of typedefs is that they can be used to hide platform specific details. On some platforms, an integer is 2 bytes, and on others, it is 4. Thus, using int to store more than 2 bytes of information can be potentially dangerous when writing platform independent code.

Because char, short, int, and long give no indication of their size, it is fairly common for cross-platform programs to use typedefs to define aliases that include the type's size in bits. For example, int8 would be an 8-bit integer, int16 a 16-bit integer, and int32 a 32-bit integer. Using typedef names in this manner helps prevent mistakes and makes it more clear about what kind of assumptions have been made about the size of the variable.

In order to make sure each typedef type resolves to a type of the right size, typedefs of this kind are typically used in conjunction with the preprocessor:

```
1  #ifdef INT_2_BYTES
2  typedef char int8;
3  typedef int int16;
4  typedef long int32;
5  #else
6  typedef char int8;
7  typedef short int16;
   typedef int int32;
   #endif
```

On machines where integers are only 2 bytes, INT_2_BYTES can be #defined, and the program will be compiled with the top set of typedefs. On machines where integers are 4 bytes, leaving INT_2_BYTES undefined will cause the bottom set of typedefs to be used. In this way, int8 will resolve to a 1 byte integer, int16 will resolve to a 2 bytes integer, and int32 will resolve to a 4 byte integer using the combination of char, short, int, and long that is appropriate for the machine the program is being compiled on.

4.7 — Structs

There are many instances in programming where we need more than one variable in order to represent something. For example, to represent yourself, you might want to store your name, your birthday, your height, your weight, or any other number of characteristics about yourself. You could do so like this:

```
1  char strName[20];
2  int nBirthYear;
3  int nBirthMonth;
4  int nBirthDay;
5  int nHeight; // in inches
   int nWeight; // in pounds
```

However, you now have 6 independent variables that are not grouped in any way. If you wanted to pass information about yourself to a function, you'd have to pass each variable individually. Furthermore, if you wanted to store information about more people, you'd have to declare 6 more variables for each additional person! As you can see, this can quickly get out of control.

Fortunately, C++ allows us to create our own user-defined aggregate data types. An **aggregate data type** is a data type that groups multiple individual variables together. One of the simplest aggregate data type is the struct. A **struct** (short for structure) allows us to group variables of mixed data types together into a single unit.

Because structs are user-defined, we first have to tell the compiler what our struct looks like before we can begin using it. To do this, we declare our struct using the *struct* keyword. Here is an example of a struct declaration:

```
1  struct Employee
2  {
3      int nID;
4      int nAge;
5      float fWage;
   };
```

This tells the compiler that we are defining a struct named Employee. The Employee struct contains 3 variables inside of it: two ints and a float. These variables are called **members** (or fields). Keep in mind that the above is just a declaration — even though we are telling the compiler that the struct will have variables, no memory is allocated at this time.

In order to use the Employee struct, we simply declare a variable of type Employee:

```
1  Employee sJoe;
```

sJoe is a variable of type Employee. As with normal variables, declaring a variable allocates memory for that variable. Typically, the size of a struct is the sum of the size of all it's members. In this case, since each integer is 4 bytes and a float is 4 bytes, Employee would be 12 bytes. However, some platforms have specific rules about how variables must be laid out in memory — consequently, the compiler may leave gaps between the variables. As a result, we can say the struct will be at minimum 12 bytes.

To find out the exact size of Employee, we can use the sizeof operator:

```
1  cout << "The size of Employee is " << sizeof(Employee);
```

On the author's Pentium 4, this prints `The size of Employee is 12`.

When we declare a variable such as `Employee sJoe`, sJoe refers to the entire struct (which contains the member variables). In order to access the individual members, we use the **member selection operator** (which is a period). As with normal variables, struct member variables are not initialized, and will typically contain junk. We must initialize them manually. Here is an example of using the member selection operator to initialize each member variable:

```
1  Employee sJoe;
2  sJoe.nID= 14; // initialize nID within sJoe
3  sJoe.nAge = 32; // initialize nAge within sJoe
   sJoe.fWage = 24.15; // initialize fWage within sJoe
```

It is possible to declare multiple variables of the same struct type:

```
1  Employee sJoe; // create an Employee struct for Joe
2  sJoe.nID = 14;
3  sJoe.nAge = 32;
4  sJoe.fWage = 24.15;
5
6  Employee sFrank; // create an Employee struct for Frank
7  sFrank.nID = 15;
   sFrank.nAge = 28;
   sFrank.fWage = 18.27;
```

In the above example, it is very easy to tell which member variables belong to Joe and which belong to Frank. This provides a much higher level of organization than individual variables would. Furthermore, because the members all have the same name, this provides consistency across multiple variables of the same type.

Struct member variables act just like normal variables, so it is possible to do normal operations on them:

```
1  int nTotalAge = sJoe.nAge + sFrank.nAge;
2
3  if (sJoe.fWage > sFrank.fWage)
4      cout << "Joe makes more than Frank" << endl;
```

```
// Frank got a promotion
sFrank.fWage += 2.50;

// Today is Joe's birthday
sJoe.nAge++;
```

Another big advantage of using structs over individual variables is that we can pass the entire struct to a function that needs to work with the members:

```
#include <iostream>

void PrintInformation(Employee sEmployee)
{
    using namespace std;
    cout << "ID:   " << sEmployee.nID << endl;
    cout << "Age:  " << sEmployee.nAge << endl;
    cout << "Wage: " << sEmployee.fWage << endl << endl;
}

int main()
{
    Employee sJoe; // create an Employee struct for Joe
    sJoe.nID = 14;
    sJoe.nAge = 32;
    sJoe.fWage = 24.15;

    Employee sFrank; // create an Employee struct for Frank
    sFrank.nID = 15;
    sFrank.nAge = 28;
    sFrank.fWage = 18.27;

    // Print Joe's information
    PrintInformation(sJoe);

    // Print Frank's information
    PrintInformation(sFrank);

    return 0;
}
```

In the above example, we pass an entire Employee struct to PrintInformation(). This prevents us from having to pass each variable individually. Furthermore, if we ever decide to add new members to our Employee struct, we will not have to change the function declaration or function call!

PrintInformation() uses the Employee struct passed to it to print out employee information to the screen. The above program outputs:

```
ID:   14
Age:  32
Wage: 24.15
```

```
ID:   15
Age:  28
Wage: 18.27
```

Structs can contain other structs. For example:

```
1  struct Company
2  {
3      Employee sCEO; // Employee is a struct within the Company struct
4      int nNumberOfEmployees;
5  };

   Company sMyCompany;
```

In this case, if we wanted to know what the CEO's salary was, we simply use the member selection operator twice: `sMyCompany.sCEO.fWage;`

This selects the sCEO member from sMyCompany, and then selects the fWage member from within sCEO.

Initializer lists

Initializing structs member by member is a little cumbersome, so C++ supports a faster way to initialize structs using an **initializer list**. This allows you to initialize some or all the members of a struct at declaration time.

```
1  struct Employee
2  {
3      int nID;
4      int nAge;
5      float fWage;
6  };

   Employee sJoe = {1, 42, 60000.0f}; // nID=1, nAge=42, fWage=60000.0
```

You can use nested initializer lists for nested structs:

```
1  struct Employee
2  {
3      int nID;
4      int nAge;
5      float fWage;
6  };
7
8  struct Company
9  {
10     Employee sCEO; // Employee is a struct within the Company struct
11     int nNumberOfEmployees;
12 };

   Company sCo1 = {{1, 42, 60000.0f}, 5};
```

```
13
14
```

A few final notes on structs

The "m_" Hungarian Notation prefix for members is typically not used for structs, even though structs contain members. This is (in part) because all variables in a struct are members! Consequently, labeling them with a "m_" prefix is somewhat redundant.

It is common to declare structs in a header file, so they can be accessed by multiple source files.

The class aggregate data type builds on top of the functionality offered by structs. Classes are at the heart of C++ object-oriented programming. Understanding structs is the first step towards object-oriented programming!

Quiz

1) You are running a website, and you are trying to keep track of how much money you make per day from advertising. Declare an advertising struct that keeps track of how many ads you've shown to readers, what percentage of users clicked on ads (as a floating point number between 0 and 1), and how much you earned on average from each ad that was clicked. Read in values for each of these fields from the user. Pass the advertising struct to a function that prints each of the values, and then calculates how much you made for that day (multiply all 3 fields together).

2) Write a struct to hold a fraction. The struct should have a integer numerator and a integer denominator member. Declare 2 fraction variables and read them in from the user. Write a function called multiply that takes both fractions, multiplies them together, and prints the result out as a decimal number.

Quiz Answers

1) Answer

```
1  #include <iostream>
2
3  // First we need to define our Advertising struct
4  struct Advertising
5  {
6      int nAdsShown;
7      float fClickThroughRate; // as a %
8      float fAverageEarningsPerClick;
9  };
10
11 void PrintAdvertising(Advertising sAd)
12 {
13     using namespace std;
       cout << "Number of ads shown: " << sAd.nAdsShown << endl;
       cout << "Click through rate: " << sAd.fClickThroughRate << endl;
       cout << "Average earnings per click: $" <<
```

```
sAd.fAverageEarningsPerClick << endl;

    // The following line is split up to reduce the length
    cout << "Total Earnings: $" <<
        (sAd.nAdsShown * sAd.fClickThroughRate *
sAd.fAverageEarningsPerClick) << endl;
}

int main()
{
    using namespace std;
    // Declare an Advertising struct variable
    Advertising sAd;

    cout << "How many ads were shown today? ";
    cin >> sAd.nAdsShown;
    cout << "What was the click through rate? ";
    cin >> sAd.fClickThroughRate;
    cout << "What was the average earnings per click? ";
    cin >> sAd.fAverageEarningsPerClick;

    PrintAdvertising(sAd);
    return 0;
}
```

2) Answer

```
#include <iostream>

struct Fraction
{
    int nNumerator;
    int nDenominator;
};

void Multiply(Fraction sF1, Fraction sF2)
{
    using namespace std;

    // Don't forget the static cast, otherwise the compiler will do
integer division!
    cout << static_cast<float>(sF1.nNumerator * sF2.nNumerator) /
        (sF1.nDenominator * sF2.nDenominator);
}

int main()
{
    using namespace std;

    // Allocate our first fraction
    Fraction sF1;
    cout << "Input the first numerator: ";
    cin >> sF1.nNumerator;
    cout << "Input the first denominator: ";
```

```
24      cin >> sF1.nDenominator;
25
26      // Allocate our second fraction
27      Fraction sF2;
28      cout << "Input the second numerator: ";
29      cin >> sF2.nNumerator;
30      cout << "Input the second denominator: ";
31      cin >> sF2.nDenominator;
32
33      Multiply(sF1, sF2);
34
        return 0;
}
```

Control Flow

5.1 — Control flow introduction

When a program is run, the CPU begins execution at the top of main(), executes some number of statements, and then terminates at the end of main(). The sequence of statements that the CPU executes is called the program's **path**. Most of the programs you have seen so far have been **straight-line programs**. Straight-line programs have **sequential flow** — that is, they take the same path (execute the same statements) every time they are run (even if the user input changes).

However, often this is not what we desire. For example, if we ask the user to make a selection, and the user enters an invalid choice, ideally we'd like to ask the user to make another choice. This is not possible in a straight-line program.

Fortunately, C++ provides **control flow statements** (also called *flow control statements*), which allow the programmer to change the CPU's path through the program. There are quite a few different types of control flow statements, so we will cover them briefly here, and then in more detail throughout the rest of the section.

Halt

The most basic control flow statement is the **halt**, which tells the program to quit running immediately. In C++, a halt can be accomplished through use of the exit() function that is defined in the cstdlib header. The exit function takes an integer parameter that is returned to the operating system as an exit code, much like the return value of main().

Here is an example of using exit():

```
1  #include <cstdlib>
2  #include <iostream>
3
4  int main()
5  {
6      using namespace std;
7      cout << 1;
8      exit(0); // terminate and return 0 to operating system
9
10     // The following statements never execute
11     cout << 2;
12     return 0;
13 }
```

Jumps

The next most basic flow control statement is the jump. A **jump** unconditionally causes the CPU to jump to another statement. The *goto*, *break*, and *continue* keywords all cause different types of jumps — we will discuss the difference between these in upcoming sections.

Function calls also cause jump-like behavior. When a function call is executed, the CPU jumps to the top of the function being called. When the called function ends, execution returns to the statement after the function call.

Conditional branches

A **conditional branch** is a statement that causes the program to change the path of execution based on the value of an expression. The most basic conditional branch is an *if statement*, which you have seen in previous examples. Consider the following program:

```
1  int main()
2  {
3      // do A
4      if (bCondition)
5          // do B
6      else
7          // do C
8
9      // do D
10 }
```

This program has two possible paths. If bCondition is true, the program will execute A, B, and D. If bCondition is false, the program will execute A, C, and D. As you can see, this program is no longer a straight-line program — it's path of execution depends on the value of bCondition.

The *switch* keyword also provides a mechanism for doing conditional branching. We will cover if statements and switch statements in more detail shortly.

Loops

A **loop** causes the program to repeatedly execute a series of statements until a given condition is false. For example:

```
1  int main()
2  {
3      // do A
4      // loop on B
5      // do C
6  }
```

This program might execute as ABC, ABBC, ABBBC, ABBBBC, or even AC. Again, you can see that this program is no longer a straight-line program — it's path of execution depends on how many times (if any) the looped portion executes.

C++ provides 3 types of loops: *while*, *do while*, and *for* loops. Unlike more modern programming languages, such as C# or D, C++ does not provide a foreach keyword. We will discuss loops at length toward the end of this section.

Exceptions

Finally, **exceptions** offer a mechanism for handling errors that occur in functions. If an error occurs that the function can not handle, it can raise an exception, and control jumps to the nearest block of code that has declared it is willing to catch exceptions of that type. Exception handling is a fairly advanced feature of C++, and is the only type of control flow statement that we won't be discussing in this section.

Conclusion

Using program flow statements, you can affect the path the CPU takes through the program and control under what conditions it will terminate. This makes any number of interesting things possible, such as displaying a menu repeatedly until the user makes a valid choice, printing every number between x and y, or determining the factors of a number.

Once you understand program flow, the things you can do with a C++ program really open up. No longer will you be restricted to toy programs and academic exercises — you will be able to write programs that have real applications. So let's get to it!

5.2 — If statements

The most basic kind of conditional branch in C++ is the **if statement**. An *if statement* takes the form:

```
if (expression)
    statement
```

or

```
if (expression)
    statement
else
    statement2
```

If the expression evalutes to true (non-zero), the statement executes. If the expression evaluates to false, the *else statement* is executed if it exists.

Here is a simple program that uses an *if statement*:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "Enter a number: ";
7      int nX;
8      cin >> nX;
9
10     if (nX > 10)
11         cout << nX << "is greater than 10" << endl;
12     else
13         cout << nX << "is not greater than 10" << endl;
14
15     return 0;
16 }
```

Note that the *if statement* only executes a single statement if the expression is true, and the *else* only executes a single statement if the expression is false. In order to execute multiple statements, we can use a block:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      cout << "Enter a number: ";
```

```
    int nX;
    cin >> nX;

    if (nX > 10)
        {
        // both statements will be executed if nX > 10
        cout << "You entered " << nX << endl;
        cout << nX << "is greater than 10" << endl;
        }
    else
        {
        // both statements will be executed if nX <= 10
        cout << "You entered " << nX << endl;
        cout << nX << "is not greater than 10" << endl;
        }

    return 0;
}
```

It is possible to chain if-else statements together:

```
int main()
{
    using namespace std;
    cout << "Enter a number: ";
    int nX;
    cin >> nX;

    if (nX > 10)
        cout << nX << "is greater than 10" << endl;
    else if (nX < 5)
        cout << nX << "is less than 5" << endl;
    // could add more else if statements here
    else
        cout << nX << "is between 5 and 10" << endl;

    return 0;
}
```

It is also possible to nest if statements within other if statements:

```
#include <iostream>

int main()
{
    using namespace std;
    cout << "Enter a number: ";
    int nX;
    cin >> nX;

    if (nX > 10)
        // it is bad coding style to nest if statements this way
        if (nX < 20)
            cout << nX << "is between 10 and 20" << endl;
```

```
12
13          // who does this else belong to?
14          else
15              cout << nX << "is greater than 20" << endl;
16
    return 0;
}
```

The above program introduces a source of potential ambiguity called a **dangling else** problem. Is the *else statement* in the above program matched up with the outer or inner *if statement*?

The answer is that an *else statement* is paired up with the last unmatched *if statement* in the same block. Thus, in the program above, the *else* is matched up with the inner *if statement*.

To avoid such ambiguities when nesting complex statements, it is generally a good idea to enclose the statement within a block. Here is the above program written without ambiguity:

```
1   #include <iostream>
2
3   int main()
4   {
5       using namespace std;
6       cout << "Enter a number: ";
7       int nX;
8       cin >> nX;
9
10      if (nX > 10)
11      {
12          if (nX < 20)
13              cout << nX << "is between 10 and 20" << endl;
14          else // attached to inner if statement
15              cout << nX << "is greater than 20" << endl;
16      }
17
18      return 0;
19  }
```

Now it is much clearer that the *else statement* belongs to the inner *if statement*.

Encasing the inner *if statement* in a block also allows us to explicitly attach an *else* to the outer *if statement*:

```
1   #include <iostream>
2
3   int main()
4   {
5       using namespace std;
6       cout << "Enter a number: ";
7       int nX;
8       cin >> nX;
```

```
    if (nX > 10)
    {
        if (nX < 20)
            cout << nX << "is between 10 and 20" << endl;
    }
    else // attached to outer if statement
        cout << nX << "is less than 10" << endl;

    return 0;
}
```

The use of a block tells the compiler that the *else statement* should attach to the *if statement* before the block. Without the block, the *else statement* would attach to the nearest unmatched *if statement*, which would be the inner *if statement*.

If statements are commonly used to do error checking. For example, to calculate a square root, the value passed to the square root function should be a non-negative number:

```
#include <iostream>
#include <cmath> // for sqrt()

void PrintSqrt(double dValue)
{
    using namespace std;
    if (dValue >= 0.0)
        cout << "The square root of " << dValue << " is " << sqrt(dValue)
<< endl;
    else
        cout << "Error: " << dValue << " is negative" << endl;
}

```

If statements can also be used to do **early returns**, where a function returns control to the caller before the end of the function. In the following program, if the parameter nValue is negative, the function returns a symbolic constant or enumerated value error code to the caller right away.

```
int DoCalculation(int nValue)
{
    // if nValue is a negative number
    if (nValue < 0)
       // early return an error code
        return ERROR_NEGATIVE_NUMBER;

    // Do calculations on nValue here

    return nValue;
}

```

If statements are also commonly used to do simple math functionality, such as a min() or max() function that returns the minimum or maximum of it's parameters:

```
1  int min(int nX, int nY)
2  {
3      if (nX > nY)
4          return nY;
5      else
6          return nX;
7  }
```

Note that this last function is so simple, it can also be written using the arithmetic if operator (?:):

```
1  int min(int nX, int nY)
2  {
3      return nX > nY ? nY : nX;
4  }
```

5.3 — Switch statements

Although it is possible to chain many if-else statements together, this is difficult to read. Consider the following program:

```
1  enum Colors
2  {
3      COLOR_BLACK,
4      COLOR_WHITE,
5      COLOR_RED,
6      COLOR_GREEN,
7      COLOR_BLUE,
8  };
9
10 void PrintColor(Colors eColor)
11 {
12     using namespace std;
13     if (eColor == COLOR_BLACK)
14         cout << "Black";
15     else if (eColor == COLOR_WHITE)
16         cout << "White";
17     else if (eColor == COLOR_RED)
18         cout << "Red";
19     else if (eColor == COLOR_GREEN)
20         cout << "Green";
21     else if (eColor == COLOR_BLUE)
22         cout << "Blue";
23     else
24         cout << "Unknown";
25 }
```

Because doing if-else chains on a single variable testing for equality is so common, C++ provides an alternative conditional branching operator called a **switch**. Here is the same program as above in switch form:

```
1  void PrintColor(Colors eColor)
2  {
3      using namespace std;
4      switch (eColor)
5      {
6          case COLOR_BLACK:
7              cout << "Black";
8              break;
9          case COLOR_WHITE:
10             cout << "White";
11             break;
12         case COLOR_RED:
13             cout << "Red";
14             break;
15         case COLOR_GREEN:
```

```
            cout << "Green";
            break;
        case COLOR_BLUE:
            cout << "Blue";
            break;
        default:
            cout << "Unknown";
            break;
    }
}
```

The overall idea behind switch statements is simple: the switch expression is evalutated to produce a value, and each case label is tested against this value for equality. If a case label matches, the statements after the case label are executed. If no case label matches the switch expression, the code under the default label is executed (if it exists).

Let's examine each of these concepts in more detail.

We start a switch statement by using the **switch** keyword, followed by the expression that we would like to evaluate. Typically this expression is just a single variable, but it can be something more complex like `nX + 2` or `nX - nY`. The one restriction on this expression is that it must evaluate to an integral type (that is, char, short, int, long, or enum). Floating point variables and other non-integral types may not be used here.

Following the switch expression, we declare a block. Inside the block, we use **labels** to define all of the values we want to test for equality. There are two kinds of labels.

The first kind of label is the **case label**, which is declared using the **case** keyword, and followed by a constant expression. A constant expression is one that evaluates to a constant value — in other words, either a literal (such as 5), an enum (such as COLOR_RED), or a constant integral variable (such as nX, when nX has been defined as a const int).

The constant expression following the case label is tested for equality against the expression following the switch keyword. If they match, the code under the case label is executed. Typically, we end each case with a break statement, which tells the compiler that we are done with the case.

It is worth noting that all case label expressions must evaluate to a unique value. That is, you can not do this:

```
switch (nX)
{
    case 4:
    case 4:  // illegal -- already used value 4!
    case COLOR_BLUE: // illegal, COLOR_BLUE evaluates to 4!
};
```

It is possible to have multiple case labels refer to the same statements. The following function uses multiple cases to test if the cChar parameter is an ASCII number.

```
1  bool IsNumber(char cChar)
2  {
3      switch (cChar)
4      {
5          case '0':
6          case '1':
7          case '2':
8          case '3':
9          case '4':
10         case '5':
11         case '6':
12         case '7':
13         case '8':
14         case '9':
15             return true;
16         default:
17             return false;
18     }
19 }
```

If any of the case labels match, the next statement is executed. In this case, that is the statement `return true;`, which returns the value true to the caller.

The second kind of label is the **default label**, which is declared using the **default** keyword. The code under this label gets executed if none of the cases match the switch expression. The default label is optional. It is also typically declared as the last label in the switch block, though this is not strictly necessary.

One of the trickiest things about case statements is the way in which execution proceeds when a case is matched. When a case is matched (or the default is executed), execution begins at the first statement following that label and continues until one of the following conditions is true:
1) The end of the switch block is reached
2) A return statement occurs
3) A goto statement occurs
4) A break statement occurs

Note that if none of these conditions are met, cases will overflow into other cases! Consider the following program:

```
1  switch (2)
2  {
3     case 1: // Does not match -- skipped
4         cout << 1 << endl;
5     case 2: // Match!  Execution begins at the next statement
6         cout << 2 << endl; // Execution begins here
7     case 3:
8         cout << 3 << endl; // This is also executed
9     case 4:
10        cout << 4 << endl; // This is also executed
11    default:
12        cout << 5 << endl; // This is also executed
13 }
```

This program prints the result:

```
2
3
4
5
```

Probably not what we wanted! When execution flows from one case into another case, this is called **fall-through**. Fall-through is almost never desired by the programmer, so in the rare case where it is, it is common practice to leave a comment stating that the fall-through is intentional.

In order to prevent fall-through, we have to use a break, a return, or a goto statement at the end of our case statements. A return statement terminates the current function immediately, and a value is possibly returned to the caller. This makes a lot of sense in short functions with a single purpose, such as the IsNumber() function example above.

However, it is often the case that we want to terminate the case statements without also terminating the entire function. To do this, we use a break statement. A **break statement** (declared using the **break** keyword) tells the compiler that we are done with this switch (or while, do while, or for loop), and execution continues with the statement after the end of the switch block.

Let's look at our last example with break statements properly inserted:

```
1  switch (2)
2  {
3     case 1: // Does not match -- skipped
4         cout << 1 << endl;
5         break;
6     case 2: // Match!  Execution begins at the next statement
7         cout << 2 << endl; // Execution begins here
8         break; // Break terminates the switch statement
9     case 3:
10        cout << 3 << endl;
11        break;
12    case 4:
13        cout << 4 << endl;
14        break;
15    default:
16        cout << 5 << endl;
17        break;
18 }
19 // Execution resumes here
```

Warning: Forgetting the break statements in a switch block is one of the most common C++ mistakes made!

Quiz

1) Write a function called Calculate() that takes two integers and a char representing one of the following mathematical operations: +, -, /, or *. Use a switch statement to perform the appropriate mathematical operation on the integers, and return the result. If an invalid operator is passed into the function, the function should print "Error" and the program should exit (use the exit() function).

Quiz answers

1) answer

```
1  using namespace std;
2  int Calculate(int nX, int nY, char chOperator)
3  {
4      switch (chOperator)
5      {
6          case '+':
7              return nX + nY;
8          case '-':
9              return nX - nY;
10         case '*':
11             return nX * nY;
12         case '/':
13             return nX / nY;
14         default:
15             cout << "Error" << endl;
16             exit(1);
17     }
18 }
```

5.4 — Goto statements

The **goto statement** is a control flow statement that causes the CPU to jump to another spot in the code. This spot is identified through use of a **statement label**. The following is an example of a goto statement and statement label:

```
1  #include <iostream>
2  #include <cmath>
3
4  int main()
5  {
6      using namespace std;
7  tryAgain: // this is a statement label
8      cout << "Enter a non-negative number";
9      double dX;
10     cin >> dX;
11
12     if (dX < 0.0)
13         goto tryAgain; // this is the goto statement
14
15     cout << "The sqrt of " << dX << " is " << sqrt(dX) << endl;
16 }
```

In this program, the user is asked to enter a non-negative number. However, if a negative number is entered, the program utilizes a goto statement to jump back to the tryAgain label. The user is then asked again to enter a new number. In this way, we can continually ask the user for input until he or she enters something valid.

In the section on variables, we covered three kinds of scope: local (block) scope, file scope, and global scope. Statement labels utilize a fourth kind of scope: function scope. The goto statement and it's corresponding statement label must appear in the same function.

There are some restrictions on goto use. For example, you can't jump forward over a variable that's initialized in the same block as the goto:

```
1  int main()
2  {
3      goto skip; // invalid forward jump
4      int x = 5;
5  skip:
6      x += 3; // what would this even evaluate to?
7      return 0;
8  }
```

In general, use of goto is shunned in C++ (and most other high level languages as well). Edsger W. Dijkstra, a noted computer scientist, laid out the case in a famous but difficult to read paper

called Go To Statement Considered Harmful. Almost any program written using a goto statement can be more clearly written using loops. The primary problem with goto is that it allows a programmer to cause the point of execution to jump around the code arbitrarily. This creates what is not-so-affectionately known as **spaghetti code**. Spaghetti code has a path of execution that resembles a bowl of spaghetti (all tangled and twisted), making it extremely difficult to follow the logic of such code.

As Dijkstra says somewhat humorously, "the quality of programmers is a decreasing function of the density of go to statements in the programs they produce".

Rule: Avoid use of goto unless necessary

5.5 — While statements

The **while statement** is the simplest of the three loops that C++ provides. It's definition is very similar to that of an *if statement*:

```
while (expression)
    statement;
```

A while statement is declared using the **while** keyword. When a while statement is executed, the expression is evaluated. If the expression evaluates to true (non-zero), the statement executes.

However, unlike an *if statement*, once the statement has finished executing, control returns to the top of the while statement and the process is repeated.

Let's take a look at a simple while loop. The following program prints all the numbers from 0 and 9:

```
1  int iii = 0;
2  while (iii < 10)
3      {
4      cout << iii << " ";
5      iii++;
6      }
   cout << "done!";
```

This outputs:

```
0 1 2 3 4 5 6 7 8 9 done!
```

iii is initialized to 0. 0 < 10 evaluates to true, so the statement block executes. The first statement prints 0, and the second increments iii to 1. Control then returns back to the top of the while statement. 1 < 10 evaluates to true, so the code block is executed again. The code block will repeatedly execute until iii == 10, at which point 10 < 10 will evaluate to false, and the loop will exit.

It is possible that a while statement executes 0 times. Consider the following program:

```
1  int iii = 15;
2  while (iii < 10)
3      {
4      cout << iii << " ";
5      i++;
6      }
   cout << "done!";
```

The condition 15 < 10 evaluates to false, so the while statement is skipped. The only thing this program prints is `done!`.

On the other hand, if the expression always evaluates to true, the while loop will execute forever. This is called an **infinite loop**. Here is an example of an infinite loop:

```
1  int iii = 0;
2  while (iii < 10)
3      cout << iii << " ";
```

Because iii is never incremented in this program, iii < 10 will always be true. Consequently, the loop will never terminate, and the program will hang. We can declare an intentional infinite loop like this:

```
1  while (1)
2  {
3    // this loop will execute forever
4  }
```

The only way to exit an infinite loop is through a return statement, a break statement, an exception being thrown, or the user killing the program.

Often, we want a loop to execute a certain number of times. To do this, it is common to use a **loop variable**. A loop variable is an integer variable that is declared for the sole purpose of counting how many times a loop has executed. Loop variables are often given simple names, such as i, j, or k. Hungarian Notation is often ignored for loop variables (though whether it should be is another question altogether).

However, naming variables i, j, or k has one major problem. If you want to know where in your program a loop variable is used, and you use the search function on i, j, or k, the search function will return half your program! Many words have an i, j, or k in them. Consequently, a better idea is to use iii, jjj, or kkk as your loop variable names. Because these names are more unique, this makes searching for loop variables much easier, and helps them stand out as loop variables. An even better idea is to use "real" variable names, such as nCount, nLoop, or a name that gives more detail about what you're counting.

Each time a loop executes, it is called an **iteration**. Often, we want to do something every n iterations, such as print a newline. To have something happen every n interations, we can use the modulus operator:

```
1  // Loop through every number between 1 and 50
2  int iii = 1;
3  while (iii <= 50)
4  {
5      // print the number
6      cout << iii << " ";
```

```
7       // if the loop variable is divisible by 10, print a newline
8       if (iii % 10 == 0)
9           cout << endl;
10
11      // increment the loop counter
12      iii++;
    }
```

This program produces the result:

```
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

It is also possible to nest loops inside of other loops. In the following example, the inner loop and outer loops each have their own counters. However, note that the loop expression for the inner loop makes use of the outer loop's counter as well!

```
1   // Loop between 1 and 5
2   int iii=1;
3   while (iii<=5)
4   {
5       // loop between 1 and iii
6       int jjj = 1;
7       while (jjj <= iii)
8           cout << jjj++;
9
10      // print a newline at the end of each row
11      cout << endl;
12      iii++;
    }
```

This program prints:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Quiz

1) In the above program, why is jjj declared inside the while block instead of following immediately following the declaration of iii?

2) Write a program that prints out the letters a-z along with their ASCII codes. Hint: to print characters as integers, you have to use a static_cast.

Quiz Answers

1) Answer

jjj is declared inside the while block so that it is recreated (and reinitialized to 1) each time the outer loop executes. If jjj were declared before the outer while loop, it's value would never be reset to 1, or we'd have to do it with an assignment statement. Furthermore, because jjj is only used inside the outer while loop block, it makes sense to declare it there. Remember, declare your variables in the smallest scope possible!

2) Answer

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6
7      char chValue = 'a';
8      while (chValue <= 'z')
9      {
10         cout << chValue << " " << static_cast<int>(chValue) << endl;
11         chValue++;
       }
   }
```

5.6 — Do while statements

One interesting thing about the while loop is that if the loop condition is false, the while loop may not execute at all. It is sometimes the case that we want a loop to execute at least once, such as when displaying a menu. To facilitate this, C++ offers the do while loop:

```
do
    statement;
while (condition);
```

The statement in a do while loop always executes at least once. After the statement has been executed, the do while loop checks the condition. If the condition is true, the CPU jumps back to the top of the do while loop and executes it again.

Here is an example of using a do while loop to display a menu to the user and wait for the user to make a valid choice:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6
7      // nSelection must be declared outside do/while loop
8      int nSelection;
9
10     do
11     {
12         cout << "Please make a selection: " << endl;
13         cout << "1) Addition" << endl;
14         cout << "2) Subtraction" << endl;
15         cout << "3) Multiplication" << endl;
16         cout << "4) Division" << endl;
17         cin >> nSelection;
18     } while (nSelection != 1 && nSelection != 2 &&
19             nSelection != 3 && nSelection != 4);
20
21     // do something with nSelection here
22     // such as a switch statement
23
24     return 0;
25 }
```

One interesting thing about the above example is that the nSelection variable must be declared outside of the do block. Think about it for a moment and see if you can figure out why that is.

If the nSelection variable is declared inside the do block, it will be destroyed when the do block terminates, which happens before the while conditional is executed. But we need the variable to use in the while conditional — consequently, the nSelection variable must be declared outside the do block.

Generally it is good form to use a do while loop instead of a while loop when you intentionally want the loop to execute at least once, as it makes this assumption explicit — however, it's not that big of a deal either way.

5.7 — For statements

By far, the most utilized looping statement in C++ is the *for statement*. The for statement is ideal when we know exactly how many times we need to iterate, because it lets us easily declare, initialize, and change the value of loop variables after each iteration.

The for statement looks pretty simple:

```
for (init-statement; expr1; expr2)
   statement;
```

The easiest way to think about for loops is convert them into equivalent while loops. In older versions of C++, the above for loop was exactly equivalent to:

```
// older compilers
init-statement;
while (expr1)
{
    statement;
    expr2;
}
```

However, in newer compilers, variables declared during the init-statement are now considered to be scoped inside the while block rather than outside of it. This is known as loop scope. Variables with **loop scope** exist only within the loop, and are not accessible outside of it. Thus, in newer compilers, the above for loop is effectively equivalent to the following while statement:

```
// newer compilers
{
    init-statement;
    while (expr1)
    {
        statement;
        expr2;
    }
} // variables declared in init-statement go out of scope here
```

A *for statement* is evaluated in 3 parts:

1) Init-statement is evaluated. Typically, the init-statement consists of variable declarations and assignments. This statement is only evaluated once, when the loop is first executed.

2) The expression expr1 is evaluated. If expr1 is false, the loop terminates immediately. If expr1 is true, the statement is executed.

3) After the statement is executed, the expression expr2 is evaluated. Typically, this expression consists of incremented/decrementing the variables declared in init-statement. After expr2 has been evaluated, the loop returns to step 2.

Let's take a look at an example of a for loop:

```
1  for (int iii=0; iii < 10; iii++)
2      cout << iii << " ";
```

What does this do? Although this looks somewhat confusing, let's take it piece by piece.

First, we declare a loop variable named iii, and assign it the value 0.

Second, the iii < 10 is evaluated, and since iii is 0, 0 < 10 evaluates to true. Consequently, the statement executes, which prints 0.

Third, After the statement executes, iii++ is evaluated, which increments iii to 1. Then the loop goes back to the second step.

1 < 10 is evaluates to true, so the loop iterates again. The statement prints 1, and iii is incremented to 2. 2 < 10 evaluates to true, the statement prints 2, and iii is incremented to 3. And so on.

Eventually iii is incremented to 10, 10 < 10 evaluates to false, and the loop exits.

Consequently, this program prints the result:

```
0 1 2 3 4 5 6 7 8 9
```

For loops can be hard for new programmers to read -- however, experienced programmers love them because they are a very compact way to do loops of this nature. Let's uncompact the above for loop by converting it into it's while-statement equivalent:

```
1  {
2      int iii = 0;
3      while (iii < 10)
4      {
5          cout << iii << " ";
6          iii++;
7      }
   }
```

That doesn't look so bad, does it? Note that the outer braces are necessary here, because iii goes out of scope when the loop ends (in newer compilers).

Here is an example of a for loop affecting a variable declared outside the for loop:

```
// returns the value nBase ^ nExp
int Exponent(int nBase, int nExp)
{
    int nValue = 1;
    for (int iii=0; iii < nExp; iii++)
        nValue *= nBase;

    return nValue;
}
```

This function returns the value nBase^nExp (nBase to the nExp power).

This is a straightforward incrementing for loop, with iii looping from 0 up to (but excluding) nExp.

If nExp is 0, the for loop will execute 0 times, and the function will return 1.
If nExp is 1, the for loop will execute 1 time, and the function will return 1 * nBase.
If nExp is 2, the for loop will execute 2 times, and the function will return 1 * nBase * nBase.

Although most for loops increment the loop variable by 1, we can decrement it as well:

```
for (int iii = 9; iii >= 0; iii--)
    cout << iii << " ";
```

This prints the result:

```
9 8 7 6 5 4 3 2 1 0
```

Alternately, we can change the value of our loop variable by more than 1 with each iteration:

```
for (int iii = 9; iii >= 0; iii -= 2)
    cout << iii << " ";
```

This prints the result:

```
9 7 5 3 1
```

Off-by-one errors

One of the biggest problems that new programmers have with for loops is off-by-one errors. Off-by-one errors occur when the loop iterates one too many or one too few times. This generally happens because the wrong relational operator is used in expr1 (eg. > instead of >=). These errors can be hard to track down because the compiler will not complain about them -- the program will run fine, but it will produce the wrong result.

When writing for loops, remember that the loop will execute as long as expr1 is true. Generally it is a good idea to test your loops using known values to make sure that they work as expected. If

your loop produces the right result when it iterates 0, 1, and 2 times, it will probably work for all number of iterations.

Omitted expressions

It is possible to write for loops that omit any or all of the expressions. For example:

```
1  int iii=0;
2  for ( ; iii < 10; )
3  {
4      cout << iii << " ";
5      iii++;
6  }
```

This for loop produces the result:

```
0 1 2 3 4 5 6 7 8 9
```

Rather than having the for loop do the initialization and incrementing, we've done it manually. We have done so purely for academic purposes in this example, but there are cases where not declaring a loop variable (because you already have one) or incrementing it (because you're incrementing it some other way) are desired.

Although you do not see it very often, it is worth noting that the following example produces an infinite loop:

```
for (;;)
    statement;
```

The above example is equivalent to:

```
while (true)
    statement;
```

Null statements

It is also possible to omit the statement part of a for loop. This is called a **null statement**, and it is declared by using a single semicolon.

```
1  for (int iii=0; iii < 10; iii++)
2      ;
```

This loop increments iii using the ++ operator 10 times. When the statement is executed, the null statement evaluates to nothing, and consequently, doesn't do anything. For readability purposes, the semicolon of a null statement is typically placed on it's own line. This indicates that the use of a null statment was intentional, and makes it harder to overlook the use of the null statement.

Null statements can actually be used anywhere a regular statement can (though they typically aren't, since they serve no purpose other than as a do-nothing placeholder). Because of this, it is easy to make the following mistake:

```
1  if (nValue == 0);
2      nValue = 1;
```

The programmer's intent was to assign nValue the value of 1 only if nValue had the value 0. However, due to the misplaced semicolon after the if statement, this actually executes as:

```
1  if (nValue == 0)
2      ;
3  nValue = 1;
```

Consequently, nValue is set to 1 regardless of it's previous value!

Multiple declarations

Although for loops typically iterate over only one variable, sometimes for loops need to work with multiple variables. When this happens, the programmer can make use of the comma operator in order to initialize or change the value of multiple variables:

```
1  for (int iii=0, jjj=9; iii < 10; iii++, jjj--)
2      cout << iii << " " << jjj << endl;
```

This loop initializes two variable: iii to 0, and jjj to 9. It iterates iii over the range 0 to 9, and each iteration iii is incremented and jjj is decremented.

This program produces the result:

```
0 9
1 8
2 7
3 6
4 5
5 4
6 3
7 2
8 1
9 0
```

This is the only place in C++ where the comma operator typically gets used.

Conclusion

For loops are the most commonly used loop in the C++ language. Even though it's syntax is typically a bit confusing to new programmers, you will see for loops so often that you will understand them in no time at all!

Quiz

1) Write a for loop that prints every other number from 0 to 20.

2) Write a function named SumTo() that takes an integer parameter named nValue, and returns the sum of all the numbers from 1 to nValue.

For example, SumTo(5) should return 15, which is 1 + 2 + 3 + 4 + 5.

Hint: Use a non-loop variable to accumulate the sum as you iterate from 1 to nValue, much like the Exponent() example above uses nValue to accumulate the return value each iteration.

3) What's wrong with the following for loop?

```
1  // Print all numbers from 9 to 0
2  for (unsigned int nCount = 9; nCount >= 0; nCount--)
3      cout << nCount << " ";
```

Quiz solutions

1) answer

```
1  for (int iii=0; iii <= 20; iii += 2)
2      cout << iii << endl;
```

2) answer

```
1  int SumTo(int nSumTo)
2  {
3      int nSum = 0;
4      for (int iii=1; iii <= nSumTo; iii++)
5          nSum += iii;
6
7      return nSum;
8  }
```

3) answer

This for loop executes as long as nCount >= 0. In other words, it runs until nCount is negative. However, because nCount is unsigned, nCount can never go negative. Consequently, this for loop will run for-ever! Generally, it's a good idea to avoid looping on unsigned variables unless necessary.

5.8 — Break and continue

Break

Although you have already seen the break statement in the context of switch statements, it deserves a fuller treatment since it can be used with other types of loops as well.

The break statement causes a switch statement, while loop, do while loop, or for loop to terminate. In the context of a switch statement, a break is typically used at the end of each case to signify the case is finished (which prevents fall-through):

```
switch (chChar)
{
    case '+':
        DoAddition(x, y);
        break;
    case '-':
        DoSubtraction(x, y);
        break;
    case '*':
        DoMultiplication(x, y);
        break;
    case '/':
        DoDivision(x, y);
        break;
}
```

In the context of a loop statement, a break can be used to cause the loop to terminate early:

```
#include <cstdio> // for getchar()
#include <iostream>

using namespace std;

int main()
{
    // count how many spaces the user has entered
    int nSpaceCount = 0;

    // loop 80 times
    for (int nCount=0; nCount < 80; nCount++)
    {
        char chChar = getchar(); // read a char from user

        // exit loop if user hits enter
        if (chChar == '\n')
            break;

        // increment count if user entered a space
```

```
        if (chChar == ' ')
            nSpaceCount++;
    }

    cout << "You typed " << nSpaceCount << " spaces" << endl;

    return 0;
}
```

This program allows the user to type up to 80 characters (the standard length of a console line). If the user hits enter, the break causes the loop to terminate early.

Note that break can be used to get out of an infinite loop. The following program loops until the user hits enter:

```
while (1)
{
    char chChar = getchar();
    if (chChar == '\n')
        break;
}
```

Continue

The continue statement provides a convenient way to jump back to the top of a loop earlier than normal, which can be used to bypass the remainder of the loop for an iteration. Here's an example of using continue:

```
for (int iii=0; iii < 20; iii++)
{
    // if the number is divisible by 4, skip this iteration
    if ((iii % 4) == 0)
        continue;

    cout << iii << endl;
}
```

This program prints all of the numbers from 0 to 19 that aren't divisible by 4.

Be careful when using continue with while or do while loops. Because these loops typically iterate the loop variables in the body of the loop, using continue can cause the loop to become infinite! Consider the following program:

```
int iii=0;
while (iii < 10)
{
    if (iii==5)
        continue;
    cout << iii << " ";
    iii++;
}
```

This program is intended to print every number between 0 and 9 except 5. But it actually prints:

```
0 1 2 3 4
```

and then goes into an infinite loop. When iii is 5, the if statement is true, and the loop returns back to the top. iii is never incremented. Consequently, on the next pass, iii is still 5, the if statement is still true, and the program continues to loop forever.

Using break and continue

Many textbooks caution readers not to use break and continue because it causes the execution flow to jump around. While this is certainly true, judicious use of break and continue can actually help make loops much more readable. For example, the following program prints all numbers from 0 to 99 which are not divisible by 3 or 4, and then prints out how many numbers were found that meet this criteria:

```
1  int nPrinted = 0;
2
3  for (int iii=0; iii < 100; iii++)
4  {
5      // messy!
6      if ((iii % 3) && (iii % 4))
7      {
8          cout << iii << endl;
9          nPrinted++;
10     }
11 }

cout << nPrinted << " numbers were found" << endl;
```

However, this can be rewritten as the following, which is easier to read:

```
1  int nPrinted = 0;
2
3  for (int iii=0; iii < 100; iii++)
4  {
5      // if the number is divisible by 3 or 4, skip this iteration
6      if ((iii % 3)==0 || (iii % 4)==0)
7          continue;
8
9      cout << iii << endl;
10     nPrinted++;
11 }

cout << nPrinted << " numbers were found" << endl;
```

Keeping the number of nested blocks down often improves code readability more than a break or continue harms it. For that reason, your author is generally in favor of using break and continue when and where it makes the code easier to understand.

5.9 — Random number generation

The ability to generate random numbers can be useful in certain kinds of programs, particularly in games, statistics modeling programs, and scientific simulations that need to model random events. Take games for example — Without random events, monsters would always attack you the same way, you'd always find the same treasure, the dungeon layout would never change, etc… and that would not make for a very good game.

So how do we generate random numbers? In real life, we often generate random results by doing things like flipping a coin, rolling a dice, or shuffling a deck of cards. These events involve so many physical variables (eg. gravity, friction, air resistance, momentum, etc…) that they become almost impossible to predict or control, and produce results that are for all intents and purposes random.

However, computers aren't designed to take advantage of physical variables — your computer can't toss a coin, throw a dice, or shuffle real cards. Computers live in a very controlled electrical world where everything is binary (false or true) and there is no in-between. By their very nature, computers are designed to produce results that are as predictable as possible. When you tell the computer to calculate 2 + 2, you ALWAYS want the answer to be 4. Not 3 or 5 on occasion.

Consequently, computers are generally incapable of generating random numbers. Instead, they must simulate randomness, which is most often done using pseudo-random number generators.

A **pseudo-random number generator (PRNG)** is a program that takes a starting number (called a **seed**), and performs mathematical operations on it to transform it into some other number that appears to be unrelated to the seed. It then takes that generated number and performs the same mathematical operation on it to transform it into a new number that appears unrelated to the number it was generated from. By continually applying the algorithm to the last generated number, it can generate a series of new numbers that will appear to be random if the algorithm is complex enough.

It's actually fairly easy to write a PRNG. Here's a short program that generates 100 pseudo-random numbers:

```
1  #include <stdafx.h>
2  #include <iostream>
3  using namespace std;
4
5  unsigned int PRNG()
6  {
7      // our initial starting seed is 5323
8      static unsigned int nSeed = 5323;
9
       // Take the current seed and generate a new value from it
       // Due to our use of large constants and overflow, it would be
```

```
    // very hard for someone to predict what the next number is
    // going to be from the previous one.
    nSeed = (8253729 * nSeed + 2396403);

    // Take the seed and return a value between 0 and 32767
    return nSeed  % 32767;
}

int main()
{
    // Print 100 random numbers
    for (int nCount=0; nCount < 100; ++nCount)
    {
        cout << PRNG() << "\t";

        // If we've printed 5 numbers, start a new column
        if ((nCount+1) % 5 == 0)
            cout << endl;
    }
}
```

The result of this program is:

```
20433   22044   9937    30185   29341
14783   29730   8430    3076    28768
18053   16066   26537   100     30493
4943    19511   19251   6669    32117
31575   3373    32383   30496   12710
23999   11929   5425    9938    12107
28541   1938    3450    20283   16726
6440    4938    26094   24391   12248
24803   30416   16244   19590   6644
24646   4873    2841    23831   23476
17958   8827    17400   32129   32760
25744   25405   13591   8859    15932
19086   19666   19265   14179   1165
27168   20996   29427   5857    3434
18964   11980   564     4620    400
17362   16934   11889   419     9714
19808   29699   3694    25612   5512
20256   10009   10247   1860    1846
1487    14030   2615    16035   8107
28736   267     29395   9438    20294
```

Each number appears to be pretty random with respect to the previous one. As it turns out, our algorithm actually isn't very good, for reasons we will discuss later. But it does effectively illustrate the principle of PRNG number generation.

Generating random numbers in C++

C (and by extension C++) comes with a built-in pseudo-random number generator. It is implemented as two separate functions that live in the cstdlib header:

srand() sets the initial seed value. srand() should only be called once.

rand() generates the next random number in the sequence (starting from the seed set by srand()).

Here's a sample program using these functions:

```
1  #include <stdafx.h>
2  #include <iostream>
3  #include <cstdlib> // for rand() and srand()
   using namespace std;
4
5  int main()
6  {
7      srand(5323); // set initial seed value to 5323
8
9      // Print 100 random numbers
10     for (int nCount=0; nCount < 100; ++nCount)
11     {
12         cout << rand() << "\t";
13
14         // If we've printed 5 numbers, start a new column
15         if ((nCount+1) % 5 == 0)
               cout << endl;
16     }
   }
```

Here's the output of this program:

```
17421   8558    19487   1344    26934
7796    28102   15201   17869   6911
4981    417     12650   28759   20778
31890   23714   29127   15819   29971
1069    25403   24427   9087    24392
15886   11466   15140   19801   14365
18458   18935   1746    16672   22281
16517   21847   27194   7163    13869
5923    27598   13463   15757   4520
15765   8582    23866   22389   29933
31607   180     17757   23924   31079
30105   23254   32726   11295   18712
29087   2787    4862    6569    6310
21221   28152   12539   5672    23344
28895   31278   21786   7674    15329
10307   16840   1645    15699   8401
22972   20731   24749   32505   29409
17906   11989   17051   32232   592
17312   32714   18411   17112   15510
8830    32592   25957   1269    6793
```

The range of rand()

rand() generates pseudo-random integers between 0 and RAND_MAX, a constant in cstdlib that is typically set to 32767.

Generally, we do not want random numbers between 0 and RAND_MAX — we want numbers between two other values, which we'll call nLow and nHigh. For example, if we're trying to simulate the user rolling a dice, we want random numbers between 1 and 6.

It turns out it's quite easy to take the result of rand() can convert it into whatever range we want:

```
1  // Generate a random number between nLow and nHigh (inclusive)
2  unsigned int GetRandomNumber(int nLow, int nHigh)
3  {
       return (rand() % (nHigh - nLow + 1)) + nLow;
4  }
```

PRNG sequences

If you run the rand() sample program above multiple times, you will note that it prints the same result every time! This means that while each number in the sequence is seemingly random with regards to the previous ones, the entire sequence is not random at all! And that means our program ends up totally predictable (the same inputs lead to the same outputs every time). There are cases where this can be useful or even desired (eg. you want a scientific simulation to be repeatable, or you're trying to debug why your random dungeon generator crashes).

But often, this is not what is desired. If you're writing a game of hi-lo (where the user has 10 tries to guess a number, and the computer tells them whether their guess is too high or to low), you don't want the program picking the same numbers each time. So let's take a deeper look at why this is happening, and how we can fix it.

Remember that each number in a PRNG sequence is generated from the previous number, in a deterministic way. Thus, given any starting seed number, PRNGs will always generate the same sequence of numbers from that seed as a result! We are getting the same sequence because our starting seed number is always 5323.

In order to make our entire sequence randomized, we need some way to pick a seed that's not a fixed number. The first answer that probably comes to mind is that we need a random number! That's a good thought, but if we need a random number to generate random numbers, then we're in a catch-22. It turns out, we really don't need our seed to be a random number — we just need to pick something that changes each time the program is run. Then we can use our PRNG to generate a unique sequence of pseudo-random numbers from that seed.

The commonly accepted method for doing this is to enlist the system clock. Each time the user runs the program, the time will be different. If we use this time value as our seed, then our program will generate a different sequence of numbers each time it is run!

C comes with a function called time() that returns the number of seconds since midnight on Jan 1, 1970. To use it, we merely need to include the ctime header, and then initialize srand() with a call to time(0):

```
1  #include <stdafx.h>
2  #include <iostream>
3  #include <cstdlib> // for rand() and srand()
4  #include <ctime> // for time()
5  using namespace std;
6
7  int main()
8  {
9
10     srand(time(0)); // set initial seed value to system clock
11     for (int nCount=0; nCount < 100; ++nCount)
12     {
13         cout << rand() << "\t";
14
15         if ((nCount+1) % 5 == 0)
               cout << endl;
    }
}
```

Now our program will generate a different sequence of random numbers every time!

What is a good PRNG?

As I mentioned above, the PRNG we wrote isn't a very good one. This section will discuss the reasons why. It is optional reading because it's not strictly related to C or C++, but if you like programming you will probably find it interesting anyway.

In order to be a good PRNG, the PRNG needs to exhibit a number of properties:

First, the PRNG should generate each number with approximately the same probability. This is called distribution uniformity. If some numbers are generated more often than others, the result of the program that uses the PRNG will be biased!

For example, let's say you're trying to write a random item generator for a game. You'll pick a random number between 1 and 10, and if the result is a 10, the monster will drop a powerful item instead of a common one. You would expect a 1 in 10 chance of this happening. But if the underlying PRNG is not uniform, and generates a lot more 10s than it should, your players will end up getting more rare items than you'd intended, possibly trivializing the difficulty of your game.

Generating PRNGs that produce uniform results is difficult, and it's one of the main reasons the PRNG we wrote at the top of this lesson isn't a very good PRNG.

Second, the method by which the next number in the sequence shouldn't be obvious or predictable. For example, consider the following PRNG algorithm: `nNum = nNum + 1`. This PRNG is perfectly uniform, but it's not very useful as a sequence of random numbers!

Third, the PRNG should have a good dimensional distribution of numbers. This means it should return low numbers, middle numbers, and high numbers seemingly at random. A PRNG that returned all low numbers, then all high numbers may be uniform and non-predictable, but it's still going to lead to biased results, particularly if the number of random numbers you actually use is small.

Fourth, all PRNGs are periodic, which means that at some point the sequence of numbers generated will eventually begin to repeat itself. As mentioned before, PRNGs are deterministic, and given an input number, a PRNG will produce the same output number every time. Consider what happens when a PRNG generates a number it has previously generated. From that point forward, it will begin to duplicate the sequence between the first occurrence of that number and the next occurrence of that number over and over. The length of this sequence is known as the **period**

For example, here are the first 100 numbers generated from a PRNG with poor periodicity:

```
112     9       130     97      64
31      152     119     86      53
20      141     108     75      42
9       130     97      64      31
152     119     86      53      20
141     108     75      42      9
130     97      64      31      152
119     86      53      20      141
108     75      42      9       130
97      64      31      152     119
86      53      20      141     108
75      42      9       130     97
64      31      152     119     86
53      20      141     108     75
42      9       130     97      64
31      152     119     86      53
20      141     108     75      42
9       130     97      64      31
152     119     86      53      20
141     108     75      42      9
```

You will note that it generated 9 as the second number, and 9 again as the 16th number. The PRNG gets stuck generating the sequence in-between these two 9's repeatedly: 9-130-97-64-31-152-119-86-53-20-141-108-75-42-(repeat).

A good PRNG should have a long period for all seed numbers. Designing an algorithm that meets this property can be extremely difficult — most PRNGs will have long periods for some seeds and short periods for others. If the user happens to pick that seed, then the PRNG won't be doing a good job.

Despite the difficulty in designing algorithms that meet all of these criteria, a lot of research has been done in this area because of it's importance to scientific computing.

rand() is a mediocre PRNG

The algorithm used to implement rand() can vary from compiler to compiler, leading to results that may not be consistent across compilers. Most implementations of rand() use a method called a Linear Congruential Generator (LCG). If you have a look at the first example in this lesson, you'll note that it's actually a LCG, though one with intentionally poorly picked bad constants. LCGs tend to have shortcomings that make them not good choices for certain kinds of problems.

One of the main shortcomings of rand() is that RAND_MAX is usually set to 32767 (essentially 16-bits). This means if you want to generate numbers over a larger range (eg. 32-bit integers), the algorithm is not suitable. Also, rand() isn't good if you want to generate random floating point numbers (eg. between 0.0 and 1.0), which is often useful when doing statistical modelling. Finally, rand() tends to have a relatively short period compared to other algorithms.

That said, rand() is entirely suitable for learning how to program, and for programs in which a high-quality PRNG is not a necessity. For such applications, I would highly recommend Mersenne Twister, which produces great results and is relatively easy to use.

Arrays, Strings, Pointers, and References

6.1 — Arrays (Part I)

In a previous lesson, you learned that you can use structs to aggregate many different data types into one variable. However, structs are not the only aggregate data type. An **array** is an aggregate data type that lets you access multiple variables through a single name by use of an index. In C++, all of these variables must have the same type.

Consider the case where you want to record the test scores for 30 students in a class. To do so, you would have to allocate 30 variables!

```
1 int nTestScoreStudent1;
2 int nTestScoreStudent2;
3 int nTestScoreStudent3;
4 // ...
  int nTestScoreStudent30;
```

Arrays give us a much easier way to do this:

```
1 int anTestScores[30]; // allocate 30 integers
```

In the above example, we declare an array named anTestScores. When used in an array definition, the **subscript operator** ([]) is used to tell the compiler how many variables to allocate. In this case, we're allocating 30 integers. Each of these variables in an array is called an **element**.

To access each of our 30 integer array elements, we use the subscript operator with an integer parameter called an **index** to tell the compiler which variable we want. The first element of our array is named anTestScores[0]. The second is anTestScores[1]. The tenth is anTestScores[9]. Note that in C++, arrays always count starting from zero! This means an array of size N has array elements 0 through N-1. This can be awkward for new programmers who are used to counting starting at 1.

Note that the subscript operator actually has two uses here: in the variable declaration, the subscript tells how many elements to allocate. When using the array, the subscript tells which array element to access.

```
1 int anArray[5]; // allocate 5 integers
2 anArray[0] = 7; // put the value 7 in element 0
```

Let's take a look at a simple program that uses arrays:

```
1 int anArray[3]; // allocate 3 integers
2 anArray[0] = 2;
3 anArray[1] = 3;
  anArray[2] = 4;
```

```
4
5  int nSum = anArray[0] + anArray[1] + anArray[2];
6  cout << "The sum is " << nSum << endl;
```

This program produces the result:

```
The sum is 9
```

Array elements may be accessed by a non-constant integer variable:

```
1  int anArray[5];
2  int nIndex = 3;
3  anArray[nIndex] = 7;
```

However, when doing array declarations, the size of the array must be a constant.

```
1  int anArray[5]; // Ok -- 5 is a literal constant
2
3  #define ARRAY_SIZE 5
4  int anArray[ARRAY_SIZE]; // Ok -- ARRAY_SIZE is a symbolic constant
5
6  const int nArraySize = 5;
7  int anArray[nArraySize]; // Ok -- nArraySize is a variable constant
8
9  enum ArrayElements
10 {
11     MAX_ARRAY_SIZE = 5;
12 };
13 int anArray[MAX_ARRAY_SIZE]; // Ok -- MAX_ARRAY_SIZE is an enum constant
14
15 int nSize = 5;
16 int anArray[nSize]; // Not ok! -- nSize is not a constant!
```

To summarize, array elements can be *indexed* with constants or non-constants, but arrays must be *declared* using constants. This means that the array's size must be known at compile time!

Arrays can hold any data type, including floating point values and even structs:

```
1  double adArray[5]; // declare an array of 5 doubles
2  adArray[2] = 7.0; // assign 7.0 to array element 2
3
4  struct sRectangle
5  {
6      int nLength;
7      int nWidth;
8  };
9  sRectangle asArray[5]; // declare an array of 5 sRectangle
```

To access a struct member of an array element, first pick which array element you want, and then use the member selection operator to select the member you want:

```
1  // sets the nLength member of array element 0
2  asArray[0].nLength = 24;
```

Elements of an array are treated just like normal variables, and as such have all of the same properties.

6.2 — Arrays (Part II)

Initializing Arrays

Because array variables are treated just like normal variables, they are not initialized when created. C++ provides a convenient way to initialize entire arrays via use of an **initializer list**.

```
1 int anArray[5] = { 3, 2, 7, 5, 8 };
2 cout << anArray[0] << endl;
3 cout << anArray[1] << endl;
4 cout << anArray[2] << endl;
5 cout << anArray[3] << endl;
  cout << anArray[4] << endl;
```

Which prints:

```
3
2
7
5
8
```

What happens if you don't initialize all of the elements in an array? The remaining elements are initialized to 0:

```
1 int anArray[5] = { 3, 2, 7 };
2 cout << anArray[0] << endl;
3 cout << anArray[1] << endl;
4 cout << anArray[2] << endl;
5 cout << anArray[3] << endl;
  cout << anArray[4] << endl;
```

Which prints:

```
3
2
7
0
0
```

Consequently, to initialize all the elements of an array to 0, you can do this:

```
1 // Initialize all elements to 0
2 int anArray[5] = { 0 };
```

Omitted Size

If you are initializing an array of elements using an initializer list, the compiler can figure out the size of the array for you, and you can omit explicitly declaring the size of the array:

```
1  int anArray[] = { 0, 1, 2, 3, 4 }; // declare array of 5 elements
```

Sizeof

The sizeof operator can be used with arrays. It returns the total size allocated for the entire array:

```
1  int anArray[] = { 0, 1, 2, 3, 4 }; // declare array of 5 elements
2  cout << sizeof(anArray); // prints 20 (5 elements * 4 bytes each)
```

In C++, there is no direct way to ask an array how many elements it contains. However, using the sizeof operator, we can figure it out:

```
1  int nElements = sizeof(anArray) / sizeof(anArray[0]);
```

Because all of the elements of the array have the same size, dividing the total size of the array by the size of any one of the elements yields the number of elements in the array! We use element 0 because it is the only element guaranteed to exist, as arrays must have at least one element.

Arrays and Enums

One of the big documentation problems with arrays is that that integer indices do not provide any information to the programmer about the meaning of the variable. Consider a class of 5 students:

```
1  const int nNumberOfStudents = 5;
2  int anTestScores[nNumberOfStudents];
3  anTestScores[2] = 76;
```

Who is represented by array element 2? It's not clear. Consequently, when known in advance, it is common to use enumerated values to index the array:

```
1  enum StudentNames
2  {
3      KENNY, // 0
4      KYLE, // 1
5      STAN, // 2
6      BUTTERS, // 3
7      CARTMAN, // 4
8      MAX_STUDENTS // 5
9  };
10
11 int anTestScores[MAX_STUDENTS]; // allocate 5 integers
12 anTestScores[STAN] = 76;
```

In this way, it's much clearer what each of the array elements represents. Note that an extra enumerator named MAX_STUDENTS has been added. This enumerator is used during the array

declaration to allocate one slot for each enum. This is useful for documentation purposes, and because the array will automatically be resized if another enumerator is added:

```
1  enum StudentNames
2  {
3      KENNY, // 0
4      KYLE, // 1
5      STAN, // 2
6      BUTTERS, // 3
7      CARTMAN, // 4
8      WENDY, // 5
9      MAX_STUDENTS // 6
10 };
11
12 int anTestScores[MAX_STUDENTS]; // allocate 6 integers
13 anTestScores[STAN] = 76;
```

Note that this "trick" only works if you do not change the enumerator values manually!

Quiz

1) Declare an array to hold the high temperature (to the nearest tenth of a degree) for each day of a year. Assign a value of 0 to each day.

2) Set up an enum with the names of the following animals: chicken, dog, cat, elephant, duck, and snake. Allocate an array with an element for each of these animals, and use an initializer list to initialize each element to hold the number of legs that animal has.

Quiz answers

1) answer

```
1  double adTemperature[365] = { 0 };
```

2) answer

```
1  enum Animals
2  {
3      CHICKEN,
4      DOG,
5      CAT,
6      ELEPHANT,
7      DUCK,
8      SNAKE,
9      MAX_ANIMALS
10 };
11
12 int anLegs[MAX_ANIMALS] = { 2, 4, 4, 4, 2, 0 };
```

6.3 — Arrays and loops

Loops and arrays

Because array elements can be accessed by a variable, it is common to use a loop to access or manipulate each array element in turn. Wherever you find arrays, you will almost certainly find loops as well.

Consider the case where we want to find the average test score of a class of students. Using individual variables:

```
1  const int nNumStudents = 5;
2  int nScore0 = 84;
3  int nScore1 = 92;
4  int nScore2 = 76;
5  int nScore3 = 81;
6  int nScore4 = 56;
7
8  int nTotalScore = nScore0 + nScore1 + nScore2 + nScore3 + nScore4;
   double dAverageScore = static_cast<double>(nTotalScore) / nNumStudents;
```

That's a lot of variables and a lot of typing — and this is just 5 students! Imagine how much work we'd have to do for 30 students, or 150.

Plus, if a new student is added, a new variable has to be declared, initialized, and added to the nTotalScore calculation. Any time you have to adjust old code, you run the risk of introducing errors.

Using arrays without loops offers a slightly better solution:

```
1  const int nNumStudents = 5;
2  int anScores[nNumStudents] = { 84, 92, 76, 81, 56 };
3  int nTotalScore = anScores[0] + anScores[1] + anScores[2] + anScores[3] +
   anScores[4];
4  double dAverageScore = static_cast<double>(nTotalScore) / nNumStudents;
```

This cuts down on the number of variables declared significantly, but nTotalScore still requires each array element be listed individually. Furthermore, changing the number of students means the nTotalScore formula needs to be adjusted.

Using arrays with loops:

```
1  const int nNumStudents = 5;
2  int anScores[nNumStudents] = { 84, 92, 76, 81, 56 };
3  int nTotalScore = 0;
   for (int nStudent = 0; nStudent < nNumStudents; nStudent++)
```

```
    nTotalScore += anScores[nStudent];

double dAverageScore = static_cast<double>(nTotalScore) / nNumStudents;
```

This solution is ideal in terms of both readability and maintenance. Because the loop does all of our array element accesses, the formulas adjust automatically to account for the number of elements in the array. This means the formulas do not have to be manually altered to account for new students, and we do not have to manually enter the name of each array element!

Here's another example of using a loop with an array to determine the best score in the class:

```
const int nNumStudents = 5;
int anScores[nNumStudents] = { 84, 92, 76, 81, 56 };
int nMaxScore = 0;
for (int nStudent = 0; nStudent < nNumStudents; nStudent++)
    if (anScores[nStudent] > nMaxScore)
        nMaxScore = anScores[nStudent];

cout << "The best score was " << nMaxScore << endl;
```

In this example, we use a non-loop variable called nMaxScore. nMaxScore is initialized to 0 to represent that we have not seen any scores yet. We then iterate through each element of the array, and if we find a score that is higher than any we've seen before, we set nMaxScore to that value. Thus, nMaxScore always represents the best score out of all the elements we've searched so far. By the time we reach the end of the array, nMaxScore holds the highest score in the entire array.

Loops are typically used with arrays to do one of three things:
1) Search for a value (eg. highest, lowest).
2) Calculate a value (eg. average, total)
3) Reorganize the array (eg. sort from lowest to highest)

When searching for a value, a variable is typically used to hold the best candidate value seen so far (or the array index of the best candidate). In the above example where we use a loop to find the best score, this variable is used to hold the highest score encountered so far.

When calculating a value, a variable is typically used to hold an intermediate result that is used to calculate the final value. In the above example where we are calculating an average score, nTotalScore holds the total score for all the elements examined so far. This value is then later used to calculate the overall average score.

Sorting an array is a bit more tricky, as it typically involves nested loops. We will cover sorting an array in the next lesson.

Off-by-one errors

One of the trickiest parts of using loops with arrays is making sure the loop iterates the proper number of times. Off-by-one errors are easy to make, and trying to access an element that is larger than the size of the array can have dire consequences. Consider the following program:

```
1  const int nArraySize = 5;
2  int anArray[nArraySize ] = { 6, 8, 2, 4, 9 };
3  int nMaxValue = 0;
4  for (int nIndex = 0; nIndex <= nArraySize; nIndex++)
5      if (anArray[nIndex] > nMaxValue)
6          nMaxValue = anArray[nIndex];
```

The problem with this program is that the conditional in the for loop is wrong! The array declared has 5 elements, indexed from 0 to 4. However, this array loops from 0 to 5. Consequently, on the last iteration, the array will execute this:

```
1  if (anArray[5] > nMaxValue)
2      nMaxValue = anArray[5];
```

But anArray[5] is undefined! This can cause all sorts of issues, with the most likely being that anArray[5] results in a garbage value. In this case, the probable result is that nMaxValue will be wrong.

However, imagine what would happen if we inadvertently assigned a value to anArray[5]! We might overwrite another variable (or part of it), or perhaps corrupt something — these types of bugs can be very hard to track down!

Consequently, when using loops with arrays, always double-check your loop conditions to make sure you do not introduce off-by-one errors.

Quiz

1) Print the following array to the screen using a loop:

```
1int anArray[9] = { 4, 6, 7, 3, 8, 2, 1, 9, 5 };
```

2) Ask the user for a number between 1 and 9. If the user does not enter a number between 1 and 9, repeatedly ask for a number until they do. Once they have entered a number between 1 and 9, print the array. Then search the array for the number that the user entered and print the index of that element.

Quiz solutions

1) answer

```
1  #include <iostream>
2  int main()
   {
```

```
    using namespace std;
    int anArray[9] = { 4, 6, 7, 3, 8, 2, 1, 9, 5 };
    for (int iii=0; iii < 9; iii++)
        cout << anArray[iii] << " ";
    return 0;
}
```

2) answer

```
#include <iostream>
int main()
{
    using namespace std;
    int nNumber;
    do
    {
        cout << "Enter a number: ";
        cin >> nNumber;
    } while (nNumber < 1 || nNumber > 9);

    int anArray[9] = { 4, 6, 7, 3, 8, 2, 1, 9, 5 };
    for (int iii=0; iii < 9; iii++)
        cout << anArray[iii] << " ";

    cout << endl;

    for (int jjj=0; jjj< 9; jjj++)
    {
        if (anArray[jjj] == nNumber)
        {
            cout << "The number " << nNumber << " has index " << jjj <<
endl;
            break; // since each # in the array is unique, no need to
search rest of array
        }
    }

    return 0;
}
```

6.4 — Sorting an array using selection sort

There are many different cases in which sorting an array can be useful. Algorithms (such as searching to see if a number exists in an array) can often be made simpler and/or more efficient when the input data is sorted. Furthermore, sorting is often useful for human readability, such as when printing a list of names in alphabetical order.

Sorting is generally performed by repeatedly comparing pairs of array elements, and swapping them if they meet some criteria. The order in which these elements are compared differs depending on which sorting algorithm is used, and the criteria depends on how the list will be sorted (ascending or descending order). To swap two elements, we can use the swap() function from the C++ standard library. Swap is defined in the algorithm header, and lives in the std namespace.

```
1  #include <algorithm> // for swap
2  #include <iostream>
3
4  int main()
5  {
6      using namespace std;
7
8      int x = 2;
9      int y = 4;
10     cout << "Before swap: x = " << x << ", y = " << y << endl;
11     swap(x, y); // swap also lives in std namespace
12     cout << "After swap:  x = " << x << ", y = " << y << endl;
13 }
```

This program prints:

```
Before swap: x = 2, y = 4
After swap:  x = 4, y = 2
```

Note that after the swap, the values of x and y have been interchanged!

Selection sort

There are many ways to sort an array. Selection sort is probably the easiest sort to understand, which makes it a good candidate for teaching even though it is one of the slower sorts.

Selection sort performs the following steps:
1) Starting at index 0, search the entire array to find the smallest value
2) Swap the smallest value found with the value at index 0
3) Repeat steps 1 & 2 starting from the next index

In other words, we're going to find the smallest element in the array, and put it in the first position. Then we're going to find the next smallest element, and put it in the second position. This process will be repeated until we run out of elements.

Here is an example of this algorithm working on 5 elements. Let's start with a sample array:

{ 30, 50, 20, 10, 40 }

First, we find the smallest element, starting from index 0:

{ 30, 50, 20, **10**, 40 }

We then swap this with the element at index 0:

{ **10**, 50, 20, **30**, 40 }

Now that the first element is sorted, we can ignore it. Consequently, we find the smallest element, starting from index 1:

{ 10, 50, **20**, 30, 40 }

And swap it with the element in index 1:

{ 10, **20**, **50**, 30, 40 }

Find the smallest element starting at index 2:

{ 10, 20, 50, **30**, 40 }

And swap it with the element in index 2:

{ 10, 20, **30**, **50**, 40 }

Find the smallest element starting at index 3:

{ 10, 20, 30, 50, **40** }

And swap it with the element in index 3:

{ 10, 20, 30, **40**, **50** }

Finally, find the smallest element starting at index 4:

{ 10, 20, 30, 40, **50** }

And swap it with the element in index 4 (which doesn't do anything):

{ 10, 20, 30, 40, **50** }

Done!

{ 10, 20, 30, 40, 50 }

Here's how this algorithm is implemented in C++:

```
1  const int nSize = 5;
2  int anArray[nSize] = { 30, 50, 20, 10, 40 };
3
4  // Step through each element of the array
5  for (int nStartIndex = 0; nStartIndex < nSize; nStartIndex++)
6  {
7      // nSmallestIndex is the index of the smallest element
8      // we've encountered so far.
9      int nSmallestIndex = nStartIndex;
10
11     // Search through every element starting at nStartIndex+1
12     for (int nCurrentIndex = nStartIndex + 1; nCurrentIndex < nSize;
13 nCurrentIndex++)
14     {
15         // If the current element is smaller than our previously found
16 smallest
17         if (anArray[nCurrentIndex] < anArray[nSmallestIndex])
18             // Store the index in nSmallestIndex
19             nSmallestIndex = nCurrentIndex;
20     }

       // Swap our start element with our smallest element
       swap(anArray[nStartIndex], anArray[nSmallestIndex]);
   }
```

The most confusing part of this algorithm is the nested loop. The outside loop (nStartIndex) steps through each element one by one. The inner loop (nCurrentIndex) finds the smallest element in the array starting from nStartIndex and sets the variable nSmallestIndex to point to it. The smallest index is then swapped with the start index. Then the outer loop (nStartIndex) advances one element, and the process is repeated.

Quiz

1) Selection sort the following array: { 30, 60, 20, 50, 40, 10 }. Show the array after each swap that takes place.

2) Rewrite the selection sort code above to sort in descending order (largest numbers first). Although this may seem complex, it is actually surprisingly simple.

Quiz solutions

1) answer

30 60 20 50 40 10
10 60 20 50 40 **30**
10 **20** **60** 50 40 30
10 20 **30** 50 40 **60**
10 20 30 **40** **50** 60
10 20 30 40 **50** 60 (self-swap)
10 20 30 40 50 **60** (self-swap)

2) answer

Simply change:

```
1  if (anArray[nCurrentIndex] < anArray[nSmallestIndex])
```

to:

```
1  if (anArray[nCurrentIndex] > anArray[nSmallestIndex])
```

nSmallestIndex should probably be renamed nLargestIndex as well.

```
1  const int nSize = 5;
2  int anArray[nSize] = { 30, 50, 20, 10, 40 };
3
4  // Step through each element of the array
5  for (int nStartIndex = 0; nStartIndex < nSize; nStartIndex++)
6  {
7      // nLargestIndex is the index of the largest element
8      // we've encountered so far.
9      int nLargestIndex = nStartIndex;
10
11     // Search through every element starting at nStartIndex+1
12     for (int nCurrentIndex = nStartIndex + 1; nCurrentIndex < nSize;
13 nCurrentIndex++)
14     {
15         // If the current element is smaller than our previously found
16 smallest
17         if (anArray[nCurrentIndex] > anArray[nLargestIndex])
18             // Store the index in nLargestIndex
19             nLargestIndex = nCurrentIndex;
20     }

     // Swap our start element with our largest element
     swap(anArray[nStartIndex], anArray[nLargestIndex]);
}
```

6.5 — Multidimensional Arrays

The elements of an array can be of any data type, including arrays! An array of arrays is called a **multidimensional array**.

```
1 int anArray[3][5]; // a 3-element array of 5-element arrays
```

In this case, since we have 2 subscripts, this is a two-dimensional array. In a two-dimensional array, it is convenient to think of the first subscript as being the row, and the 2nd subscript as being the column. Conceptually, the above two-dimensional array is laid out as follows:

```
[0][0]  [0][1]  [0][2]  [0][3]  [0][4]
[1][0]  [1][1]  [1][2]  [1][3]  [1][4]
[2][0]  [2][1]  [2][2]  [2][3]  [2][4]
```

To access the elements of a two-dimensional array, simply use two subscripts:

```
1anArray[2][3] = 7;
```

To initialize a two-dimensional array, it is easiest to use nested braces, with each set of numbers representing a row:

```
1 int anArray[3][5] =
2 {
3 { 1, 2, 3, 4, 5, }, // row 0
4 { 6, 7, 8, 9, 10, }, // row 1
5 { 11, 12, 13, 14, 15 } // row 2
  };
```

When the C++ compiler processes this list, it actually ignores the inner braces altogether. However, we highly recommend you use them anyway for readability purposes.

Two-dimensional arrays with initializer lists can omit (only) the first size specification:

```
1 int anArray[][5] =
2 {
3 { 1, 2, 3, 4, 5, },
4 { 6, 7, 8, 9, 10, },
  { 11, 12, 13, 14, 15 }
  };
```

The compiler can do the math to figure out what the array size is. However, the following is not allowed:

```
1 int anArray[][] =
2 {
```

```
3   { 1, 2, 3, 4 },
    { 5, 6, 7, 8 }
    };
```

Because the inner parenthesis are ignored, the compiler can not tell whether you intend to declare a 1×8, 2×4, 4×2, or 8×1 array in this case.

Just like normal arrays, multidimensional arrays can still be initialized to 0 as follows:

```
1   int anArray[3][5] = { 0 };
```

Note that this only works if you explicitly declare the size of the array! Otherwise, you will get a two-dimensional array with 1 row.

Accessing all of the elements of a two-dimensional array requires two loops: one for the row, and one for the column. Since two-dimensional arrays are typically accessed row by row, generally the row index is used as the outer loop.

```
1   for (int nRow = 0; nRow < nNumRows; nRow++)
2       for (int nCol = 0; nCol < nNumCols; nCol++)
            cout << anArray[nRow][nCol];
```

Multidimensional arrays may be larger than two dimensions. Here is a declaration of a three-dimensional array:

```
1   int anArray[5][4][3];
```

Three-dimensional arrays are hard to initialize in any kind of intuitive way using initializer lists, so it's typically better to initialize the array to 0 and explicitly assign values using nested loops.

Let's take a look at a practical example of a two-dimensional array:

```
1   // Declare a 10x10 array
2   const int nNumRows = 10;
3   const int nNumCols = 10;
4   int nProduct[nNumRows ][nNumCols ] = { 0 };
5
6   // Calculate a multiplication table
7   for (int nRow = 0; nRow < nNumRows; nRow++)
8       for (int nCol = 0; nCol < nNumCols; nCol++)
9           nProduct[nRow][nCol] = nRow * nCol;
10
11  // Print the table
12  for (int nRow = 1; nRow < nNumRows; nRow++)
13  {
14      for (int nCol = 1; nCol < nNumCols; nCol++)
15          cout << nProduct[nRow][nCol] << "\t";

        cout << endl;
    }
```

This program calculates and prints a multiplication table for all values between 1 and 9 (inclusive). Note that when printing the table, the for loops start from 1 instead of 0. This is to omit printing the 0 column and 0 row, which would just be a bunch of 0s! Here is the output:

```
1    2    3    4    5    6    7    8    9
2    4    6    8    10   12   14   16   18
3    6    9    12   15   18   21   24   27
4    8    12   16   20   24   28   32   36
5    10   15   20   25   30   35   40   45
6    12   18   24   30   36   42   48   54
7    14   21   28   35   42   49   56   63
8    16   24   32   40   48   56   64   72
9    18   27   36   45   54   63   72   81
```

6.6 — C-style strings

Under regular C (and hence also C++), it is possible to use arrays to represent strings. A **string** is a sequence of chars that are interpreted as a piece of text. You have already seen string literals:

```
1  cout << "This is a string literal";
```

In C and C++, strings are typically represented as char arrays that have a null terminator. A **null terminator** means that the string ends with a '\0′ character (which has ASCII code 0). Arrays that are null terminated in this manner are often named using the Hungarian Notation prefix "sz".

To *declare* a C-style string, simply declare a char array and assign it a value:

```
1  char szString[] = "string";
```

Although "string" is only 6 letters, this actually declares an array of length 7. The following program prints out the length of the string, and then the ASCII values of all of the characters:

```
1  cout << sizeof(szString) << endl;
2  for (int nChar = 0; nChar < sizeof(szString); nChar++)
3      cout << static_cast<int>(szString[nChar]) << " ";
```

This produces the result:

```
7
115 116 114 105 110 103 0
```

That 0 is the ASCII code of the null terminator that has been appended to the end of the string.

Just like with normal arrays, once an array is declared to be a particular size, it can not be changed. Our szString above is of length 7 — which means it can fit 6 chars of our choice and the null terminator. If you try to stick more than 6 chars in the array, you will overwrite the null terminator and the CPU won't know where the string ends. If you try to print a string with no null terminator, you'll not only get the string, you'll also get everything in the adjacent memory slots until you happen to hit a 0.

When declaring strings in this manner, it is always a good idea to use [] and let the compiler calculate the size of the array. That way if you change the string later, you won't have to manually adjust the size.

It is important to realize that a single char (eg. 'a') is typically only allocated one byte, but the equivalent string (eg. "a") is allocated two bytes — one for the char, and one for the null terminator.

Since C-style strings are arrays, you can use the [] operator to change individual characters in the string:

```
1  char szString[] = "string";
2  szString[1] = 'p';
3  cout << szString;
```

This snippet prints:

```
spring
```

One important point to note is that strings follow ALL the same rules as arrays. This means you can initialize the string upon creation, but you can not assign values to it using the assignment operator after that!

```
1  char szString[] = "string"; // ok
2  szString = "rope"; // not ok!
```

This would be the conceptual equivalent of the following nonsensical example:

```
1  int anArray[] = { 3, 5, 7, 9 };
2  anArray = 8; // what does this mean?
```

Buffers and buffer overflow

You can read text into a string using cin:

```
1  char szString[255];
2  cin >> szString;
3  cout << "You entered: " << szString << endl;
```

Why did we declare the string to be 255 characters long? The answer is that we don't know how many characters the user is going to enter. We are using this array of 255 characters as a buffer. A **buffer** is memory set aside temporarily to hold data. In this case, we're temporarily holding the user input before we write it out using cout.

If the user were to enter more characters than our array could hold, we would get a buffer overflow. A **buffer overflow** occurs when the program tries to store more data in a buffer than the buffer can hold. Buffer overflow results in other memory being overwritten, which usually causes a program crash, but can cause any number of other issues. By making our buffer 255 charaters long, we are guessing that the user will not enter this many characters. Although this is commonly seen in C/C++ programming, it is poor programming.

The recommended way of reading strings using cin is as follows:

```
1  char szString[255];
```

```
2  cin.getline(szString, 255);
3  cout << "You entered: " << szString << endl;
```

This call to cin.getline() will read up to 254 characters into szString (leaving room for the null terminator!). Any excess characters will be discarded. In this way, we guarantee that buffer overflow will not occur.

Manipulating C-style strings

C++ provides many functions to manipulate C-style strings. For example, strcpy() allows you to make a copy of a string.

```
1  char szSource[] = "Copy this!";
2  char szDest[50];
3  strcpy(szDest, szSource);
   cout << szDest; // prints "Copy this!"
```

However, strcpy() can cause buffer overflows! In the following program, szDest isn't big enough to hold the entire string, so buffer overflow results.

```
1  char szSource[] = "Copy this!";
2  char szDest[4];
3  strcpy(szDest, szSource); // buffer overflow!
   cout << szDest;
```

It is better to use strncpy(), which takes a length parameter to prevent buffer overflow:

```
1  char szSource[] = "Copy this!";
2  char szDest[50];
3  strncpy(szDest, szSource, 49); // copy at most 49 characters (indices 0-48)
4  szDest[49] = 0; // ensures the last character is a null terminator
   cout << szDest; // prints "Copy this!"
```

Other useful functions:
strcat() — Appends one string to another (dangerous)
strncat() — Appends one string to another (with buffer length check)
strcmp() — Compare two strings (returns 0 if equal)
strncmp() — Compare two strings up to a specific number of characters (returns 0 if equal)
strlen() — Returns the length of a string (excluding the null terminator)

Here's an example program using some of the concepts in this lesson:

```
1  // Ask the user to enter a string
2  char szBuffer[255];
3  cout << "Enter a string: ";
4  cin.getline(szBuffer, 255);
5
6  int nSpacesFound = 0;
   // Loop through all of the characters the user entered
   for (int nChar = 0; nChar < strlen(szBuffer); nChar++)
```

```
{
    // If the current character is a space, count it
    if (szBuffer[nChar] == ' ')
        nSpacesFound++;
}

cout << "You typed " << nSpacesFound << " spaces!" << endl;
```

std::string

It is important to know about C-style strings because they are used in a lot of code. However, we recommend avoiding them altogether whenever possible!

A better idea is to use the string class in the standard library (std::string), which lives in the string header. std::string lets you work with strings in a way that is much more intuitive. You can assign strings to them using the assignment operator and they will automatically resize to be as large or small as needed.

Here is a quick example using std::string:

```
#include <string> // for std::string
#include <iostream>

int main()
{
    using namespace std; // for both cout and string
    cout << "Enter your name: ";
    string strString;
    cin >> strString;
    cout << "Hello, " << strString << "!" << endl;

    cout << "Your name has: " << strString.length() <<
            " characters in it" << endl;
    cout << "The 2nd character is: " << strString[1] << endl;

    strString = "Dave";
    cout << "Your name is now " << strString << endl;
    cout << "Goodbye, " << strString << endl;

    return 0;
}
```

One extremely useful function to use with std::string is getline(). This allows you to read an entire string in, even if it includes whitespace:

```
cout << "Enter your full name: ";

string strName;
getline(cin, strName);

cout << "You entered: "<< strName <<endl;
```

For example:

```
Enter your full name: John Smith
You entered: John Smith
```

The nice thing about std::string is that you don't have to guess how large the input string is likely to be in advance!

We will talk more about std::string in future lessons. But feel free to experiment with it in the meantime.

6.7 — Introduction to pointers

Pointers are one of the most powerful and confusing aspects of the C language. A **pointer** is a variable that holds the address of another variable. To declare a pointer, we use an asterisk between the data type and the variable name:

```
1  int *pnPtr; // a pointer to an integer value
2  double *pdPtr; // a pointer to a double value
3
4  int* pnPtr2; // also valid syntax
5  int * pnPtr3; // also valid syntax
```

Note that an asterisk placed between the data type and the variable name means the variable is being declared as a pointer. In this context, the asterisk is not a multiplication. It does not matter if the asterisk is placed next to the data type, the variable name, or in the middle — different programmers prefer different styles, and one is not inherently better than the other.

Since pointers only hold addresses, when we assign a value to a pointer, the value has to be an address. To get the address of a variable, we can use the **address-of operator (&)**:

```
1  int nValue = 5;
2  int *pnPtr = &nValue; // assign address of nValue to pnPtr
```

Conceptually, you can think of the above snippet like this:

It is also easy to see using code:

```
1  int nValue = 5;
2  int *pnPtr = &nValue; // assign address of nValue to pnPtr
3
4  cout << &nValue << endl; // print the address of variable nValue
5  cout << pnPtr << endl; // print the address that pnPtr is holding
```

On the author's machine, this printed:

```
0012FF7C
0012FF7C
```

The type of the pointer has to match the type of the variable being pointed to:

```
1  int nValue = 5;
2  double dValue = 7.0;
```

```
3
4  int *pnPtr = &nValue; // ok
5  double *pdPtr = &dValue; // ok
6  pnPtr = &dValue; // wrong -- int pointer can not point to double value
7  pdPtr = &nValue; // wrong -- double pointer can not point to int value
```

Dereferencing pointers

The other operator that is commonly used with pointers is the **dereference operator (*)**. A dereferenced pointer evaluates to the *contents* of the address it is pointing to.

```
1  int nValue = 5;
2  cout << &nValue; // prints address of nValue
3  cout << nValue; // prints contents of nValue
4
5  int *pnPtr = &nValue; // pnPtr points to nValue
6  cout << pnPtr; // prints address held in pnPtr, which is &nValue
7  cout << *pnPtr; // prints contents pointed to by pnPtr, which is contents
   of nValue
```

The above program prints:

```
0012FF7C
5
0012FF7C
5
```

In other words, when pnPtr is assigned to &nValue:
pnPtr is the same as &nValue
*pnPtr is the same as nValue

Because *pnPtr is the same as nValue, you can assign values to it just as if it were nValue! The following program prints 7:

```
1  int nValue = 5;
2  int *pnPtr = &nValue; // pnPtr points to nValue
3
4  *pnPtr = 7; // *pnPtr is the same as nValue, which is assigned 7
   cout << nValue; // prints 7
```

Pointers can also be assigned and reassigned:

```
1  int nValue1 = 5;
2  int nValue2 = 7;
3
4  int *pnPtr;
5
6  pnPtr = &nValue1; // pnPtr points to nValue1
7  cout << *pnPtr; // prints 5
```

```
8 pnPtr = &nValue2; // pnPtr now points to nValue2
9 cout << *pnPtr; // prints 7
```

The null pointer

Sometimes it is useful to make our pointers point to nothing. This is called a **null pointer**. We assign a pointer a null value by setting it to address 0:

```
1 int *pnPtr;
2 pnPtr = 0; // assign address 0 to pnPtr
```

or shorthand:

```
1 int *pnPtr = 0;  // assign address 0 to pnPtr
```

Note that in the last example, the * is not a dereference operator. It is a pointer declaration. Thus we are assigning address 0 to pnPtr, not the value 0 to the variable that pnPtr points to.

C (but not C++) also defines a special preprocessor define called NULL that evaluates to 0. Even though this is not technically part of C++, it's usage is common enough that it will work in every C++ compiler:

```
1 int *pnPtr = NULL; // assign address 0 to pnPtr
```

Because null pointers point to 0, they can be used inside conditionals:

```
1 if (pnPtr)
2     cout << "pnPtr is pointing to an integer.";
3 else
4     cout << "pnPtr is a null pointer.";
```

Null pointers are mostly used with dynamic memory allocation, which we will talk about in a few lessons.

The size of pointers

The size of a pointer is dependent upon the architecture of the computer — a 32-bit computer uses 32-bit memory addresses — consequently, a pointer on a 32-bit machine is 32 bits (4 bytes). On a 64-bit machine, a pointer would be 64 bits (8 bytes). Note that this is true regardless of what is being pointed to:

```
1 char *pchValue; // chars are 1 byte
2 int *pnValue; // ints are usually 4 bytes
3 struct Something
4 {
5     int nX, nY, nZ;
  };
```

```
Something *psValue; // Something is probably 12 bytes

cout << sizeof(pchValue) << endl; // prints 4
cout << sizeof(pnValue) << endl; // prints 4
cout << sizeof(psValue) << endl; // prints 4

```

As you can see, the size of the pointer is always the same. This is because a pointer is just a memory address, and the number of bits needed to access a memory address on a given machine is always constant.

Quiz

1) What values does this program print? Assume a short is 2 bytes, and a 32-bit machine

```
short nValue = 7; // &nValue = 0012FF60
short nOtherValue = 3; // &nOtherValue = 0012FF54
short *pnPtr = &nValue;

cout << &nValue << endl;
cout << nValue << endl;
cout << pnPtr << endl;
cout << *pnPtr << endl;
cout << endl;

*pnPtr = 9;

cout << &nValue << endl;
cout << nValue << endl;
cout << pnPtr << endl;
cout << *pnPtr << endl;
cout << endl;

pnPtr = &nOtherValue;

cout << &nOtherValue << endl;
cout << nOtherValue << endl;
cout << pnPtr << endl;
cout << *pnPtr << endl;
cout << endl;

cout << sizeof(pnPtr) << endl;
cout << sizeof(*pnPtr) << endl;
```

Quiz solutions

1) answer

```
0012FF60
7
0012FF60
7
```

```
0012FF60
9
0012FF60
9

0012FF54
3
0012FF54
3

4
2
```

A short explanation about the 4 and the 2. A 32-bit machine means that pointers will be 32 bits in length, but sizeof() always prints the size in bytes. 32 bits is 4 bytes. Thus the sizeof(pnPtr) is 4. Because pnPtr is a pointer to a short, *pnPtr is a short. The size of a short in this example is 2 bytes. Thus the sizeof(*pnPtr) is 2.

6.8 — Pointers, arrays, and pointer arithmetic

Pointers and arrays
Pointers and arrays are intricately linked in the C language. In previous lessons, you learned how to declare an array of variables:

```
int anArray[5]; // declare array of 5 integers
```

anArray is actually a pointer that points to the first element of the array! Because the array variable is a pointer, you can dereference it, which returns array element 0:

```
int anArray[5] = { 9, 7, 5, 3, 1 };

// dereferencing an array returns the first element (element 0)
cout << *anArray; // prints 9!

char szName[] = "Jason"; // C-style string (also an array)
cout << *szName; // prints 'J'

```

Pointer arithmetic

The C language allows you to perform integer addition or subtraction operations on pointers. If pnPtr points to an integer, `pnPtr + 1` is the address of the next integer in memory after pnPtr. `pnPtr - 1` is the address of the previous integer before pnPtr.

Note that `pnPtr+1` does not return the *address* after pnPtr, but the *next object of the type* that pnPtr points to. If pnPtr points to an integer (assuming 4 bytes), `pnPtr+3` means 3 integers after pnPtr, which is 12 addresses after pnPtr. If pnPtr points to a char, which is always 1 byte, `pnPtr+3` means 3 chars after pnPtr, which is 3 addresses after pnPtr.

When calculating the result of a pointer arithmetic expression, the compiler always multiplies the integer operand by the size of the object being pointed to. This is called **scaling**.

The following program:

```
int nValue = 7;
int *pnPtr = &nValue;

cout << pnPtr << endl;
cout << pnPtr+1 << endl;
cout << pnPtr+2 << endl;
cout << pnPtr+3 << endl;
```

```
7
```

Outputs:

```
0012FF7C
0012FF80
0012FF84
0012FF88
```

As you can see, each of these addresses differs by 4 (7C + 4 = 80 in hexadecimal). This is because an integer is 4 bytes on the author's machine.

The same program using short instead of int:

```
1  short nValue = 7;
2  short *pnPtr = &nValue;
3
4  cout << pnPtr << endl;
5  cout << pnPtr+1 << endl;
6  cout << pnPtr+2 << endl;
   cout << pnPtr+3 << endl;
```

Outputs:

```
0012FF7C
0012FF7E
0012FF80
0012FF82
```

Because a short is 2 bytes, each address differs by 2.

It is rare to see the + and – operator used in such a manner with pointers. However, it is more common to see the ++ or — operator being used to increment or decrement a pointer to point to the next or previous element in an array.

Pointer arithmetic and arrays

If anArray is a pointer that points to the first element (element 0) of the array, and adding 1 to a pointer already returns the next object, then anArray+1 must point to the second element (element 1) of the array! We can verify experimentally that this is true:

```
1  int anArray[5] = { 9, 7, 5, 3, 1 };
2  cout << *(anArray+1) << endl; // prints 7
```

The parentheses are necessary to ensure the operator precedence is correct — operator * has higher precedence than operator +.

Note that *(anArray+1) has the same effect as anArray[1]. It turns out that the array indexing operator ([]) actually does an implicit pointer addition and dereference! It just looks prettier.

We can use a pointer and pointer arithmetic to loop through an array. Although not commonly done this way (using indices is generally easier to read and less error prone), the following example goes to show it is possible:

```
1  const int nArraySize = 7;
2  char szName[nArraySize] = "Mollie";
3  int nVowels = 0;
4  for (char *pnPtr = szName; pnPtr < szName + nArraySize; pnPtr++)
5  {
6      switch (*pnPtr)
7      {
8          case 'A':
9          case 'a':
10         case 'E':
11         case 'e':
12         case 'I':
13         case 'i':
14         case 'O':
15         case 'o':
16         case 'U':
17         case 'u':
18             nVowels++;
19             break;
       }
   }

   cout << szName << " has " << nVowels << " vowels" << endl;
```

This program uses a pointer to step through each of the elements in an array. Each element is dereferenced by the switch expression, and if the element is a vowel, nVowels is incremented. The for loop then uses the ++ operator to advance the pointer to the next character in the array. The for loop terminates when all characters have been examined.

The above program produces the result:

```
Mollie has 3 vowels
```

6.9 — Dynamic memory allocation with new and delete

All of the variables used up to this point in the tutorial have one thing in common: the variables must be declared at compile time. This leads to two issues: First, it's difficult to conditionally declare a variable, outside of putting it in an if statement block (in which case it will go out of scope when the block ends). Second, the size of all arrays must be decided upon in advance of the program being run. For example, the following is not legal:

```
1  cout << "How many variables do you want? ";
2  int nVars;
3  cin >> nVars;
4
5  int anArray[nVars]; // wrong!  The size of the array must be a constant
```

However, there are many cases where it would be useful to be able to size or resize arrays while the program is being run. For example, we may want to use a string to hold someone's name, but we do not know how long their name is until they enter it. Or we may want to read in a number of records from disk, but we don't know in advance how many records there are. Or we may be creating a game, with a variable number of monsters chasing the player.

If we have to declare the size of everything at compile time, the best we can do is try to make a guess the maximum number of variables we'll need and hope that's enough:

```
1  char szName[25]; // let's hope their name is less than 25 chars!
2  Record asRecordArray[500]; // let's hope there are less than 500 records!
3  Monster asMonsterArray[20]; // 20 monsters maximum
```

This is a poor solution for several reasons. First, it leads to wasted memory if the variables aren't actually used. For example, if we allocate 25 chars for every name, but names on average are only 12 chars long, we're allocating over twice what we really need! Second, it can lead to artificial limitations and/or buffer overflows. What happens when the user tries to read in 600 records from disk? Because we've only allocated 500 spaces, either we have to give the user an error, only read the first 500 records, or (in the worst case where we don't handle this case at all), we overflow the record buffer and our program crashes.

Fortunately, these problems are easily solved via dynamic memory allocation. **Dynamic memory allocation** allows us to allocate memory of whatever size we want when we need it.

Dynamically allocating single variables

To allocate a *single* variable dynamically, we use the scalar (non-array) form of the **new** operator:

```
1  int *pnValue = new int; // dynamically allocate an integer
```

The new operator returns the *address* of the variable that has been allocated. This address can be stored in a pointer, and the pointer can then be dereferenced to access the variable.

```
1  int *pnValue = new int; // dynamically allocate an integer
2  *pnValue = 7; // assign 7 to this integer
```

When we are done with a dynamically allocated variable, we need to explicitly tell C++ to free the memory for reuse. This is done via the scalar (non-array) form of the **delete** operator:

```
1  delete pnValue; // unallocate memory assigned to pnValue
2  pnValue = 0;
```

Note that the delete operator does not delete the pointer — it deletes the memory that the pointer points to!

Dynamically allocating arrays

Declaring arrays dynamically allows us to choose their size while the program is running. To allocate an array dynamically, we use the array form of new and delete (often called new[] and delete[]):

```
1  int nSize = 12;
2  int *pnArray = new int[nSize]; // note: nSize does not need to be constant!
3  pnArray[4] = 7;
4  delete[] pnArray;
```

Because we are allocating an array, C++ knows that it should use the array version of new instead of the scalar version of new. Essentially, the new[] operator is called, even though the [] isn't placed next to the new keyword.

When deleting a dynamically allocated array, we have to use the array version of delete, which is delete[]. This tells the CPU that it needs to clean up multiple variables instead of a single variable.

Note that array access is done the same way with dynamically allocated arrays as with normal arrays. While this might look slightly funny, given that pnArray is explicitly declared as a pointer, remember that arrays are really just pointers in C++ anyway.

One of the most common mistakes that new programmers make when dealing with dynamic memory allocation is to use delete instead of delete[] when deleting a dynamically allocated

array. Do not do this! Using the scalar version of delete on an array can cause data corruption or other problems.

Memory leaks

Dynamically allocated memory effectively has no scope. That is, it stays allocated until it is explicitly deallocated or until the program ends. However, the pointers used to access dynamically allocated memory follow the scoping rules of normal variables. This mismatch can create interesting problems.

Consider the following function:

```
1  void doSomething()
2  {
3      int *pnValue = new int;
4  }
```

This function allocates an integer dynamically, but never frees it using delete. Because pointers follow all of the same rules as normal variables, when the function ends, pnValue will go out of scope. Because pnValue is the only variable holding the address of the dynamically allocated integer, when pnValue is destroyed there are no more references to the dynamically allocated memory. This is called a **memory leak**. As a result, the dynamically allocated integer can not be deleted, and thus can not be reallocated or reused. Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well. Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash.

Memory leaks can also result if the pointer holding the address of the dynamically allocated memory is reassigned to another value:

```
1  int nValue = 5;
2  int *pnValue = new int;
3  pnValue = &nValue; // old address lost, memory leak results
```

It is also possible to get a memory leak via double-allocation:

```
1  int *pnValue = new int;
2  pnValue = new int; // old address lost, memory leak results
```

The address returned from the second allocation overwrites the address of the first allocation. Consequently, the first allocation becomes a memory leak!

Null pointers (part II)

Null pointers (pointers set to address 0) are particularly useful when dealing with dynamic memory allocation. A null pointer basically says "no memory has been allocated yet". This allows us to do things like conditionally allocate memory:

```
1  // If pnValue isn't already allocated, allocate it
2  if (!pnValue)
3      pnValue = new int;
```

Keep in mind that just like normal variables, when a pointer is created, it's value is undefined. Consequently, it is a good idea to set all pointers that are not used right away to 0:

```
1  int *pnValue = new int;
   int *pnOtherValue = 0; // will allocate later
```

Similarly, when a dynamically allocated variable is deleted, the pointer pointing to it is not zero'd. Consider the following snippet:

```
1  int *pnValue = new int;
2  delete pnValue; // pnValue not set to 0
3
4  if (pnValue)
5      *pnValue = 5; // will cause a crash
```

Because pnValue has not been set to 0, the if statement condition evaluates to true, and the program tries to assign 5 to deallocated memory. This almost inevitably will cause a program to crash. It is never a good idea to leave a pointer pointing to deallocated memory. When deallocating memory, set the pointer that has been deallocated to 0 immediately afterward. This helps ensure the program does not try and access memory that has already been deallocated. The above program should be written as:

```
1  int *pnValue = new int;
2  *pnValue = 7;
3  delete pnValue;
4  pnValue = 0;
5
6  if (pnValue)
7      *pnValue = 5;
```

Get in the habit of assigning your pointers to 0 both when they are declared (unless assigned to another address), and after they are deleted. It will save you a lot of grief.

Finally, deleting a null pointer has no effect. Thus, there is no need for the following:

```
1  if (pnValue)
2      delete pnValue;
```

Instead, you can just write:

```
1  delete pnValue;
```

If pnValue is non-null, the dynamically allocated variable will be deleted. If it is null, nothing will happen.

6.10 — Pointers and const

Just like normal variables, pointers can be declared constant. There are two different ways that pointers and const can be intermixed, and they are very easy to mix up.

To declare a const pointer, use the *const* keyword between the asterisk and the pointer name:

```
1 int nValue = 5;
2 int *const pnPtr = &nValue;
```

Just like a normal const variable, a const pointer must be initialized to a value upon declaration, and its value can not be changed. This means a const pointer will always point to the same value. In the above case, pnPtr will always point to the address of nValue. However, because the value being pointed to is still non-const, it is possible to change the value being pointed to via dereferencing the pointer:

```
1 *pnPtr = 6; // allowed, since pnPtr points to a non-const int
```

It is also possible to declare a pointer to a constant variable by using the const before the data type.

```
1 int nValue = 5;
2 const int *pnPtr = &nValue;
```

Note that the pointer to a constant variable does not actually have to point to a constant variable! Instead, think of it this way: a pointer to a constant variable treats the variable as constant when it is accessed through the pointer.

Thus, the following is okay:

```
1 nValue = 6; // nValue is non-const
```

But the following is not:

```
1 *pnPtr = 6; // pnPtr treats its value as const
```

Because a pointer to a const value is a non-const pointer, the pointer can be redirected to point at other values:

```
1 int nValue = 5;
2 int nValue2 = 6;
3
4 const int *pnPtr = &nValue;
  pnPtr = &nValue2; // okay
```

```
5
```

Confused? To summarize:

- A non-const pointer can be redirected to point to other addresses.
- A const pointer always points to the same address, and this address can not be changed.

- A pointer to a non-const value can change the value it is pointing to.
- A pointer to a const value treats the value as const (even if it is not), and thus can not change the value it is pointing to.

Finally, it is possible to declare a const pointer to a const value:

```
1 const int nValue;
2 const int *const pnPtr = &nValue;
```

A const pointer to a const value can not be redirected to point to another address, nor can the value it is pointing to be changed.

Const pointers are primarily used for passing variables to functions. We will discuss this further in the section on functions.

6.11 — References

References are a type of C++ variable that act as an alias to another variable. A reference variable acts just like the original variable it is referencing. References are declared by using an ampersand (&) between the reference type and the variable name:

```
1 int nValue = 5; // normal integer
2 int &rnRef = nValue; // reference to nValue
```

The ampersand in this context does not mean "address of", it means "reference to". Let's take a look at references in use:

```
1 nValue = 6; // nValue is now 6
2 rnRef = 7; // nValue is now 7
3
4 cout << nValue; // prints 7
5 nValue++;
  cout << rnRef; // prints 8
```

Using the address-of operator on a reference returns the address of the value being referenced:

```
1 cout << &nValue; // prints 0012FF7C
2 cout << &rnRef; // prints 0012FF7C
```

References are implicitly const. Like normal constant objects, references must be given a value upon declaration:

```
1 int nValue = 5;
2 int &rnRef = nValue; // valid reference
3
4 int &rnInvalidRef; // invalid, needs to reference something
```

Furthermore, the reference can not be "redirected" to another variable. Consider the following snippet:

```
1 int nValue = 5;
2 int nValue2 = 6;
3
4 int &rnRef = nValue;
5 rnRef = nValue2; // assigns value 6 to nValue -- does NOT change the
  reference!
```

Const references

It is possible to declare a const reference. A const reference will not let you change the value it references:

```
1  int nValue = 5;
2  const int &rnRef = nValue;
3
4  rnRef = 6; // illegal -- rnRef is const
```

You can assign const references to literal values, though there is typically not much need to do so:

```
1  const int &rnRef = 6;
```

Typical use of references

References are typically used for one of two purposes.

First, const references are often used as function parameters, which we will talk about more in the next section on functions. Because const references allow us to access but not change the value of an object, they can be used to give a function access to an object, but give assurance to the caller that the function will not change the object. This helps prevent inadvertent side effects.

Another primary use of references is to provide easier access to nested data. Consider the following struct:

```
1  struct Something
2  {
3      int nValue;
4      float fValue;
5  };
6
7  struct Other
8  {
9      Something sSomething;
10     int nOtherValue;
11 };
12
13 Other sOther;
```

Let's say we needed to work with the nValue field of the Something struct of sOther. Normally, we'd access that member as `sOther.sSomething.nValue`. If there are many separate accesses to this member, the code can become messy. References allow you to more easily access the member:

```
1  int &rnValue = sOther.sSomething.nValue;
2  // rnValue can now be used in place of sOther.sSomething.nValue
```

The following two statements are thus identical:

```
1 sOther.sSomething.nValue = 5;
1 rnValue = 5;
```

This can help keep your code cleaner and more readable.

6.12 — References vs. pointers, and member selection

References and pointers

References and pointers have an interesting relationship — a reference acts like a const pointer that is implicitly dereferenced. Thus given the following:

```
1  int nValue = 5;
2  int *const pnValue = &nValue;
3  int &rnValue = nValue;
```

`*pnValue` and `rnValue` evaluate identically. As a result, the following two statements produce the same effect:

```
1  *pnValue = 6;
2  rnValue = 6;
```

Similarly, a const reference acts just like a const pointer to a const object that is implicitly dereferenced.

Because references always "point" to valid objects, and can never be pointed to deallocated memory, references are safer to use than pointers. If a task can be solved with either a reference or a pointer, the reference should generally be preferred. Pointers should generally only be used in situations where references are not sufficient (such as dynamically allocating memory).

Member selection

It is common to have either a pointer or a reference to a struct (or class). As you learned previously, you can select the member of a struct using the **member selection operator (.)**:

```
1  struct Something
2  {
3      int nValue;
4      float fValue;
5  };
6
7  // Member selection using actual struct variable
8  Something sSomething;
9  sSomething.nValue = 5;
10
11 // Member selection using reference to struct
12 Something &rsSomething = sSomething;
13 rsSomething.nValue = 5;
```

```
12
13 // Member selection using pointer to struct
14 Something *psSomething = &sSomething;
   (*psSomething).nValue = 5;
```

Note that the pointer dereference must be enclosed in parenthesis, because the member selection operator has a higher precedence than the dereference operator.

Because the syntax for access to structs and class members through a pointer is awkward, C++ offers a second member selection operator (->) for doing member selection from pointers. The following two lines are equivalent:

```
1 (*psSomething).nValue = 5;
2 psSomething->nValue = 5;
```

This is not only easier to type, but is also much less prone to error because there are no precedence issues to worry about. Consequently, when doing member access through a pointer, always use the -> operator.

6.13 — Void pointers

The **void pointer**, also known as the generic pointer, is a special type of pointer that can be pointed at objects of any data type! A void pointer is declared like a normal pointer, using the void keyword as the pointer's type:

```
1void *pVoid; // pVoid is a void pointer
```

A void pointer can point to objects of any data type:

```
1  int nValue;
2  float fValue;
3
4  struct Something
5  {
6      int nValue;
7      float fValue;
8  };
9
10 Something sValue;
11
12 void *pVoid;
13 pVoid = &nValue; // valid
14 pVoid = &fValue; // valid
15 pVoid = &sValue; // valid
```

However, because the void pointer does not know what type of object it is pointing to, it can not be dereferenced! Rather, the void pointer must first be explicitly cast to another pointer type before it is dereferenced.

```
1  int nValue = 5;
2  void *pVoid = &nValue;
3
4  // can not dereference pVoid because it is a void pointer
5
6  int *pInt = static_cast<int*>(pVoid); // cast from void* to int*
7
8  cout << *pInt << endl; // can dereference pInt
```

Similarly, it is not possible to do pointer arithmetic on a void pointer. Note that since void pointers can't be dereferenced, there is no such thing as a void reference.

The next obvious questions is: If a void pointer doesn't know what it's pointing to, how do we know what to cast it to? Ultimately, that is up to you to keep track of.

Here's an example of a void pointer in use:

```
1  #include <iostream>
2
3  enum Type
4  {
5      INT,
6      FLOAT,
7      STRING,
8  };
9
10 void Print(void *pValue, Type eType)
11 {
12     using namespace std;
13     switch (eType)
14     {
15         case INT:
16             cout << *static_cast<int*>(pValue) << endl;
17             break;
18         case FLOAT:
19             cout << *static_cast<float*>(pValue) << endl;
20             break;
21         case STRING:
22             cout << static_cast<char*>(pValue) << endl;
23             break;
24     }
25 }
26
27 int main()
28 {
29     int nValue = 5;
30     float fValue = 7.5;
31     char *szValue = "Mollie";
32
33     Print(&nValue, INT);
34     Print(&fValue, FLOAT);
35     Print(szValue, STRING);
36     return 0;
37 }
```

This program prints:

```
5
7.5
Mollie
```

In general, it is a good idea to avoid using void pointers unless absolutely necessary, as they effectively allow you to avoid type checking. This allows you to inadvertently do things that make no sense, and the compiler won't complain about it. For example, the following would be valid:

```
1  Print(&nValue, STRING);
```

But who knows what the result would actually be!

Although the above function seems like a neat way to make a single function handle multiple data types, C++ actually offers a much better way to do the same thing (via function overloading) that retains type checking to help prevent misuse. Many other places where void pointers would once be used to handle multiple data types are now better done using templates, which also offer strong type checking.

However, very occasionally, you may still find a reasonable use for the void pointer. Just make sure there isn't a better (safer) way to do the same thing using other language mechanisms first!

Functions

7.1 — Function parameters and arguments

In Chapter 1, we covered function basics in the following sections:

- A first look at functions
- Forward declarations
- Programs with multiple files
- Header files

You should be familiar with the concepts discussed in those lessons before proceeding.

Parameters vs Arguments

Up until now, we have not differentiated between function parameters and arguments. In common usage, the terms parameter and argument are often interchanged. However, for the purposes of further discussion, we will make a distinction between the two:

A **function parameter** is a variable declared in the prototype or declaration of a function:

```
1  void foo(int x); // prototype -- x is a parameter
2
3  void foo(int x) // declaration -- x is a parameter
4  {
   }
```

An **argument** is the value that is passed to the function in place of a parameter:

```
1  foo(6); // 6 is the argument passed to parameter x
2  foo(y+1); // the value of y+1 is the argument passed to parameter x
```

When a function is called, all of the parameters of the function are created as variables, and the value of the arguments are copied into the parameters. For example:

```
1  void foo(int x, int y)
2  {
3  }
4
   foo(6, 7);
```

When foo() is called with arguments 6 and 7, foo's parameter x is created and assigned the value of 6, and foo's parameter y is created and assigned the value of 7.

Even though parameters are not declared inside the function block, function parameters have local scope. This means that they are created when the function is invoked, and are destroyed when the function block terminates:

```
1  void foo(int x, int y) // x and y are created here
2  {
   } // x and y are destroyed here
```

There are 3 primary methods of passing arguments to functions: pass by value, pass by reference, and pass by address. The following sections will address each of those cases individually.

7.2 — Passing arguments by value

Pass by value

By default, arguments in C++ are passed by value. When arguments are **passed by value**, a copy of the argument is passed to the function.

Consider the following snippet:

```
1  void foo(int y)
2  {
3      using namespace std;
4      cout << "y = " << y << endl;
5  }
6
7  int main()
8  {
9      foo(5); // first call
10
11     int x = 6;
12     foo(x); // second call
13     foo(x+1); // third call
14
15     return 0;
16 }
```

In the first call to foo(), the argument is the literal 5. When foo() is called, variable y is created, and the value of 5 is copied into y. Variable y is then destroyed when foo() ends.

In the second call to foo(), the argument is the variable x. x is evaluated to produce the value 6. When foo() is called for the second time, variable y is created again, and the value of 6 is copied into y. Variable y is then destroyed when foo() ends.

In the third call to foo(), the argument is the expression x+1. x+1 is evaluated to produce the value 7, which is passed to variable y. Variable y is once again destroyed when foo() ends.

Thus, this program prints:

```
y = 5
y = 6
y = 7
```

Because a copy of the argument is passed to the function, the original argument can not be modified by the function. This is shown in the following example:

```
1  void foo(int y)
```

```
{
    using namespace std;
    cout << "y = " << y << endl;

    y = 6;

    cout << "y = " << y << endl;
} // y is destroyed here

int main()
{
    using namespace std;
    int x = 5;
    cout << "x = " << x << endl;

    foo(x);

    cout << "x = " << x << endl;
    return 0;
}

```

This snippet outputs:

```
x = 5
y = 5
y = 6
x = 5
```

At first, x is 5. When foo() is called, the value of x (5) is passed to variable y inside foo(). y is assigned the value of 6, and then destroyed. The value of x is unchanged, even though y was changed.

Advantages of passing by value:

- Arguments passed by value can be variables (eg. x), literals (eg. 6), or expressions (eg. x+1).
- Arguments are never changed by the function being called, which prevents side effects.

Disadvantages of passing by value:

- Copying large structs or classes can take a lot of time to copy, and this can cause a performance penalty, especially if the function is called many times.

In most cases, pass by value is the best way to pass arguments to functions — it is flexible and safe.

7.3 — Passing arguments by reference

Pass by reference

When passing arguments by value, the only way to return a value back to the caller is via the function's return value. While this is suitable in many cases, there are a few cases where better options are available. One such case is when writing a function that needs to modify the values of an array (eg. sorting an array). In this case, it is more efficient and more clear to have the function modify the actual array passed to it, rather than trying to return something back to the caller.

One way to allow functions to modify the value of argument is by using pass by reference. In **pass by reference**, we declare the function parameters as references rather than normal variables:

```
1  void AddOne(int &y) // y is a reference variable
2  {
3      y = y + 1;
4  }
```

When the function is called, y will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument!

The following example shows this in action:

```
1  void foo(int &y) // y is now a reference
2  {
3      using namespace std;
4      cout << "y = " << y << endl;
5      y = 6;
6      cout << "y = " << y << endl;
7  } // y is destroyed here
8
9  int main()
10 {
11     int x = 5;
12     cout << "x = " << x << endl;
13     foo(x);
14     cout << "x = " << x << endl;
15     return 0;
16 }
```

This program is the same as the one we used for the pass by value example, except foo's parameter is now a reference instead of a normal variable. When we call foo(x), y becomes a reference to x. This snippet produces the output:

```
x = 5
y = 5
y = 6
x = 6
```

Note that the value of x was changed by the function!

Here is another example:

```
1  void AddOne(int &y)
2  {
3      y++;
4  }
5
6  int main()
7  {
8      int x = 5;
9
10     cout << "x = " << x << endl;
11     AddOne(x);
12     cout << "x = " << x << endl;
13
14     return 0;
15 }
```

This example prints:

```
x = 5;
x = 6;
```

As you can see, the function was able to change the value of the argument.

Sometimes we need a function to return multiple values. However, functions can only have one return value. One way to return multiple values is using reference parameters:

```
1  #include <iostream>
2  #include <math.h>    // for sin() and cos()
3
4  void GetSinCos(double dX, double &dSin, double &dCos)
5  {
6      dSin = sin(dX);
7      dCos = cos(dX);
8  }
9
10 int main()
11 {
12     double dSin = 0.0;
13     double dCos = 0.0;
14
15     // GetSinCos will return the sin and cos in dSin and dCos
16     GetSinCos(30.0, dSin, dCos);
17
```

```
    std::cout << "The sin is " << dSin << std::endl;
    std::cout << "The cos is " << dCos << std::endl;
    return 0;
}

```

This function takes one parameter (by value) as input, and "returns" two parameters (by reference) as output.

Pass by const reference

One of the major disadvantages of pass by value is that all arguments passed by value are copied to the parameters. When the arguments are large structs or classes, this can take a lot of time. References provide a way to avoid this penalty. When an argument is passed by reference, a reference is created to the actual argument (which takes minimal time) and no copying of values takes place. This allows us to pass large structs and classes with a minimum performance penalty.

However, this also opens us up to potential trouble. References allow the function to change the value of the argument, which in many cases is undesirable. If we know that a function should not change the value of an argument, but don't want to pass by value, the best solution is to pass by const reference.

You already know that a const reference is a reference that does not allow the variable being referenced to be changed. Consequently, if we use a const reference as a parameter, we guarantee to the caller that the function will not (and can not) change the argument!

The following function will produce a compiler error:

```
void foo(const int &x)
{
    x = 6;  // x is a const reference and can not be changed!
}

```

Using const is useful for several reasons:

- It enlists the compilers help in ensuring values that shouldn't be changed aren't changed.
- It tells the coder whether they need to worry about the function changing the value of the argument
- It helps the coder debug incorrect values by telling the coder whether the function might be at fault or not

Rule: Always pass by const reference unless you need to change the value of the argument

Summary

Advantages of passing by reference:

- It allows us to have the function change the value of the argument, which is sometimes useful.
- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
- We can pass by const reference to avoid unintentional changes.
- We can return multiple values from a function.

Disadvantages of passing by reference:

- Because a non-const reference can not be made to a literal or an expression, reference arguments must be normal variables.
- It can be hard to tell whether a parameter passed by reference is meant to be input, output, or both.
- It's impossible to tell from the function call that the argument may change. An argument passed by value and passed by reference looks the same. We can only tell whether an argument is passed by value or reference by looking at the function declaration. This can lead to situations where the programmer does not realize a function will change the value of the argument.
- Because references are typically implemented by C++ using pointers, and dereferencing a pointer is slower than accessing it directly, accessing values passed by reference is slower than accessing values passed by value.

7.4 — Passing arguments by address

There is one more way to pass variables to functions, and that is by address. **Passing an argument by address** involves passing the address of the argument variable rather than the argument variable itself. Because the argument is an address, the function parameter must be a pointer. The function can then dereference the pointer to access or change the value being pointed to.

Here is an example of a function that takes a parameter passed by address:

```
1  void foo(int *pValue)
2  {
3      *pValue = 6;
4  }
5
6  int main()
7  {
8      int nValue = 5;
9
10     cout << "nValue = " << nValue << endl;
11     foo(&nValue);
12     cout << "nValue = " << nValue << endl;
13     return 0;
14 }
```

The above snippet prints:

```
nValue = 5
nValue = 6
```

As you can see, the function foo() changed the value of nValue through the pointer parameter pValue.

Pass by address is typically used with dynamically allocated variables and arrays. For example, the following function will print all the values in an array:

```
1  void PrintArray(int *pnArray, int nLength)
2  {
3      for (int iii=0; iii < nLength; iii++)
4          cout << pnArray[iii] << endl;
5  }
```

Here is an example program that calls this function:

```
1  int main()
2  {
3      int anArray[6] = { 6, 5, 4, 3, 2, 1 };
```

```
    PrintArray(anArray, 6);
}
```

This program prints the following:

```
6
5
4
3
2
1
```

Note that the length of the array must be passed in as a parameter, because arrays don't keep track of how long they are. Otherwise the PrintArray() function would not know how many elements to print.

It is always a good idea to ensure parameters passed by address are not null pointers before dereferencing them. Dereferencing a null pointer will typically cause the program to crash. Here is our PrintArray() function with a null pointer check:

```
void PrintArray(int *pnArray, int nLength)
{
    // if user passed in a null pointer for pnArray, bail out early!
    if (!pnArray)
        return;

    for (int iii=0; iii < nLength; iii++)
        cout << pnArray[iii] << endl;
}
```

Advantages of passing by address:

- It allows us to have the function change the value of the argument, which is sometimes useful
- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
- We can return multiple values from a function.

Disadvantages of passing by address:

- Because literals and expressions do not have addresses, pointer arguments must be normal variables.
- All values must be checked to see whether they are null. Trying to dereference a null value will result in a crash. It is easy to forget to do this.
- Because dereferencing a pointer is slower than accessing a value directly, accessing arguments passed by address is slower than accessing arguments passed by value.

As you can see, pass by address and pass by reference have almost identical advantages and disadvantages. Because pass by reference is generally safer than pass by address, pass by reference should be preferred in most cases.

Passing by reference, address, and value is actually not so different

Now that you understand the basic differences between passing by reference, address, and value, let's complicate things by simplifying them. :)

In the lesson on passing arguments by reference, we briefly mentioned that references are typically implemented by the compiler as pointers. Because of this, the only real difference between pointers and references is that references have a cleaner but more restrictive syntax. This makes references easier and safer to use, but also less flexible. This also means that pass by reference and pass by address are essentially identical in terms of efficiency.

Here's the one that may surprise you. When you pass an address to a function, that *address* is actually passed by value! Because the address is passed by value, if you change the value of that address within the function, you are actually changing a temporary copy. Consequently, the original pointer address will not be changed!

Here's a sample program that illustrates this.

```
1  #include <iostream>
2
3  int nFive = 5;
4  int nSix = 6;
5
6  // Function prototype so we can define
7  // SetToSix below main()
8  void SetToSix(int *pTempPtr);
9
10 int main()
11 {
12     using namespace std;
13
14     // First we set pPtr to the address of nFive
15     // Which means *pPtr = 5
16     int *pPtr = &nFive;
17
18     // This will print 5
19     cout << *pPtr;
20
21     // Now we call SetToSix (see function below)
22     // pTempPtr receives a copy of the address of pPtr
23     SetToSix(pPtr);
24
```

```
25      return 0;
26  }
27
28  // pTempPtr copies the value of pPtr!
29  void SetToSix(int *pTempPtr)
30  {
31      using namespace std;
32
33      // This only changes pTempPtr, not pPtr!
34      pTempPtr = &nSix;
35
36      // This will print 6
        cout << *pTempPtr;
    }
```

Because pTempPtr receives a copy of the address of pPtr, even though we change pTempPtr, this does not change the value that pPtr points to. Consequently, this program prints

```
565
```

Even though the address itself is passed by value, you can still dereference that address to permanently change the value at that address! This is what differentiates pass by address (and reference) from pass by value.

The next logical question is, "What if we want to be able to change the address of an argument from within the function?". Turns out, this is surprisingly easy. You just use pass the pointer itself by reference (effectively passing the address by reference). You already learned that values passed by reference reflect any changes made in the function back to the original arguments. So in this case, we're telling the compiler that any changes made to the address of pTempPtr should be reflected back to pPtr! The syntax for doing a reference to a pointer is a little strange (and easy to get backwards): `int *&pPtr`. However, if you do get it backwards, the compiler will give you an error.

The following program illustrates using a reference to a pointer.

```
1   // pTempPtr is now a reference to a pointer to pPtr!
2   // This means if we change pTempPtr, we change pPtr!
3   void SetToSix(int *&pTempPtr)
4   {
5       using namespace std;
6
7       pTempPtr = &nSix;
8
9       // This will print 6
        cout << *pTempPtr;
    }
```

Note that you'll also have to update the function prototype above main to account for the new prototype of SetToSix():

```
1  // Function prototype so we can define
2  // SetToSix below main()
3  void SetToSix(int *&pTempPtr);
```

When we run the program again with this version of the function, we get:

```
566
```

Which shows that calling SetToSix() did indeed change the address of pPtr!

So strangely enough, the conclusion here is that references are pointers, and pointer addresses are passed by value. The value of pass by address (and reference) comes *solely* from the fact that we can dereference addresses to change the original arguments, which we can not do with a normal value parameter.

7.4a — Returning values by value, reference, and address

In the three previous lessons, you learned about passing arguments to functions by value, reference, and address. In this section, we'll consider the issue of returning values back to the caller via all three methods.

As it turns out, returning values from a function to its caller by value, address, or reference works almost exactly the same way as passing parameters to a function does. All of the same upsides and downsides for each method are present. The primary difference between the two is simply that the direction of data flow is reversed. However, there is one more added bit of complexity — because local variables in a function go out of scope when the function returns, we need to consider the effect of this on each return type.

(Author's note: This lesson has a funny lesson number because it was originally omitted from chapter 7)

Return by value

Return by value is the simplest and safest return type to use. When a value is returned by value, a copy of that value is returned to the caller. As with pass by value, you can return by value literals (eg. 5), variables (eg. x), or expressions (eg. x+1), which makes return by value very flexible.

Another advantage of return by value is that you can return variables (or expressions) that involve local variables declared within the function. Because the variables are evaluated before the function goes out of scope, and a copy of the value is returned to the caller, there are no problems when the variable goes out of scope at the end of the function.

```
1  int DoubleValue(int nX)
2  {
3      int nValue = nX * 2;
4      return nValue; // A copy of nValue will be returned here
   } // nValue goes out of scope here
```

Return by value is the most appropriate when returning variables that were declared inside the function, or for returning function arguments that were passed by value. However, like pass by value, return by value is slow for structs and large classes.

Return by reference

Just like with pass by reference, values returned by reference must be variables (you can not return a reference to a literal or an expression). When a variable is returned by reference, a reference to the variable is passed back to the caller. The caller can then use this reference to

continue modifying the variable, which can be useful at times. Return by reference is also fast, which can be useful when returning structs and classes.

However, returning by reference has one additional downside that pass by reference doesn't — you can not return local variables to the function by reference. Consider the following example:

```
1  int& DoubleValue(int nX)
2  {
3      int nValue = nX * 2;
4      return nValue; // return a reference to nValue here
   } // nValue goes out of scope here
```

See the problem here? The function is trying to return a reference to a value that is going to go out of scope when the function returns. This would mean the caller receives a reference to garbage. Fortunately, your compiler will give you an error if you try to do this.

Return by reference is typically used to return arguments passed by reference to the function back to the caller. In the following example, we return (by reference) an element of an array that was passed to our function by reference:

```
1  // This struct holds an array of 25 integers
2  struct FixedArray25
3  {
4      int anValue[25];
   };
5
6  // Returns a reference to the nIndex element of rArray
7  int& Value(FixedArray25 &rArray, int nIndex)
8  {
9      return rArray.anValue[nIndex];
   }
10
11 int main()
12 {
13     FixedArray25 sMyArray;
14
15     // Set the 10th element of sMyArray to the value 5
16     Value(sMyArray, 10) = 5;
17
18     cout << sMyArray.anValue[10] << endl;
19     return 0;
   }
```

This prints:

```
5
```

When we call `Value(sMyArray, 10)`, Value() returns a reference to the 10th element of the array inside sMyArray. main() then uses this reference to assign that element the value 5.

Although this is somewhat of a contrived example (because you could access sMyArray.anValue directly), once you learn about classes you will find a lot more uses for returning values by reference.

Return by address

Returning by address involves returning the address of a variable to the caller. Just like pass by address, return by address can only return the address of a variable, not a literal or an expression. Like return by reference, return by address is fast. However, as with return by reference, return by address can not return local variables:

```
1  int* DoubleValue(int nX)
2  {
3      int nValue = nX * 2;
4      return &nValue; // return nValue by address here
5  } // nValue goes out of scope here
```

As you can see here, nValue goes out of scope just after its address is returned to the caller. The end result is that the caller ends up with the address of non-allocated memory, which will cause lots of problems if used. This is one of the most common programming mistakes that new programmers make. Many newer compilers will give a warning (not an error) if the programmer tries to return a local variable by address — however, there are quite a few ways to trick the compiler into letting you do something illegal without generating a warning, so the burden is on the programmer to ensure the address they are returning will be to a valid variable after the function returns.

Return by address is often used to return newly allocated memory to the caller:

```
1  int* AllocateArray(int nSize)
2  {
3      return new int[nSize];
4  }
5
6  int main()
7  {
8      int *pnArray = AllocateArray(25);
9      // do stuff with pnArray
10
11     delete[] pnArray;
12     return 0;
13 }
```

Conclusion

Most of the time, return by value will be sufficient for your needs. It's also the most flexible and safest way to return information to the caller. However, return by reference or address can also be useful, particularly when working with dynamically allocated classes or structs. When using

return by reference or address, make sure you are not returning a reference to, or the address of, a variable that will go out of scope when the function returns!

7.5 — Inline functions

The use of functions provides many benefits, including:

- The code inside the function can be reused.
- It is much easier to change or update the code in a function (which needs to be done once) than for every in-place instance. Duplicate code is a recipe for disaster.
- It makes your code easier to read and understand, as you do not have to know how a function is implemented to understand what it does.
- The function provides type checking.

However, one major downside of functions is that every time a function is called, there is a certain amount of performance overhead that occurs. This is because the CPU must store the address of the current instruction it is executing (so it knows where to return to later) along with other registers, all the function parameters must be created and assigned values, and the program has to branch to a new location. Code written in-place is significantly faster.

For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This can result in a substantial performance penalty.

C++ offers a way to combine the advantages of functions with the speed of code written in-place: inline functions. The **inline** keyword is used to request that the compiler treat your function as an inline function. When the compiler compiles your code, all inline functions are expanded in-place — that is, the function call is replaced with a copy of the contents of the function itself, which removes the function call overhead! The downside is that because the inline function is expanded in-place for *every* function call, this can make your compiled code quite a bit larger, especially if the inline function is long and/or there are many calls to the inline function.

Consider the following snippet:

```
1  int min(int nX, int nY)
2  {
3      return nX > nY ? nY : nX;
4  }
5
6  int main()
7  {
8      using namespace std;
9      cout << min(5, 6) << endl;
10     cout << min(3, 2) << endl;
11     return 0;
```

```
10 }
```

This program calls function min() twice, incurring the function call overhead penalty twice. Because min() is such a short function, it is the perfect candidate for inlining:

```
1 inline int min(int nX, int nY)
2 {
3     return nX > nY ? nY : nX;
4 }
```

Now when the program compiles main(), it will create machine code as if main() had been written like this:

```
1 int main()
2 {
3     using namespace std;
4     cout << (5 > 6 ? 6 : 5) << endl;
5     cout << (3 > 2 ? 2 : 3) << endl;
6     return 0;
7 }
```

This will execute quite a bit faster, at the cost of the compiled code being slightly larger.

Because of the potential for code bloat, inlining a function is best suited to short functions (eg. no more than a few lines) that are typically called inside loops and do not branch. Also note that the inline key word is only a recommendation — the compiler is free to ignore your request to inline a function. This is likely to be the result if you try to inline a lengthy function!

7.6 — Function overloading

Function overloading is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different parameters. Consider the following function:

```
1  int Add(int nX, int nY)
2  {
3      return nX + nY;
4  }
```

This trivial function adds two integers. However, what if we also need to add two floating point numbers? This function is not at all suitable, as any floating point parameters would be converted to integers, causing the floating point arguments to lose their fractional values.

One way to work around this issue is to define multiple functions with slightly different names:

```
1  int AddI(int nX, int nY)
2  {
3      return nX + nY;
4  }
5
6  double AddD(double dX, double dY)
7  {
8      return dX + dY;
9  }
```

However, for best effect, this requires that you define a consistent naming standard, remember the name of all the different flavors of the function, and call the correct one (calling AddD() with integer parameters may produce the wrong result due to precision issues).

Function overloading provides a better solution. Using function overloading, we can declare another Add() function that takes double parameters:

```
1  double Add(double dX, double dY)
2  {
3      return dX + dY;
4  }
```

We now have two version of Add():

```
1  int Add(int nX, int nY); // integer version
2  double Add(double dX, double dY); // floating point version
```

Which version of Add() gets called depends on the arguments used in the call — if we provide two ints, C++ will know we mean to call Add(int, int). If we provide two floating point numbers, C++ will know we mean to call Add(double, double). In fact, we can define as many overloaded Add() functions as we want, so long as each Add() function has unique parameters.

Consequently, it's also possible to define Add() functions with a differing number of parameters:

```
1  int Add(int nX, int nY, int nZ)
2  {
3      return nX + nY + nZ;
4  }
```

Even though this Add() function has 3 parameters instead of 2, because the parameters are different than any other version of Add(), this is valid.

Note that the function's return type is NOT considered when overloading functions. Consider the case where you want to write a function that returns a random number, but you need a version that will return an int, and another version that will return a double. You might be tempted to do this:

```
1  int GetRandomValue();
2  double GetRandomValue();
```

But the compiler will flag this as an error. These two functions have the same parameters (none), and consequently, the second GetRandomValue() will be treated as an erroneous redeclaration of the first. Consequently, these functions will need to be given different names.

Also keep in mind that declaring a typedef does not introduce a new type — consequently, the following the two declarations of Print() are considered identical:

```
1  typedef char *string;
2  void Print(string szValue);
3  void Print(char *szValue);
```

How function calls are matched with overloaded functions

Making a call to an overloaded function results in one of three possible outcomes:

1) A match is found. The call is resolved to a particular overloaded function.
2) No match is found. The arguments can not be matched to any overloaded function.
3) An ambiguous match is found. The arguments matched more than one overloaded function.

When an overloaded function is called, C++ goes through the following process to determine which version of the function will be called:

1) First, C++ tries to find an exact match. This is the case where the actual argument exactly matches the parameter type of one of the overloaded functions. For example:

```
1 void Print(char *szValue);
2 void Print(int nValue);
3
4 Print(0); // exact match with Print(int)
```

Although 0 could technically match Print(char*), it exactly matches Print(int). Thus Print(int) is the best match available.

2) If no exact match is found, C++ tries to find a match through promotion. In the lesson on type conversion and casting, we covered how certain types can be automatically promoted via internal type conversion to other types. To summarize,

- Char, unsigned char, and short is promoted to an int.
- Unsigned short can be promoted to int or unsigned int, depending on the size of an int
- Float is promoted to double
- Enum is promoted to int

For example:

```
1 void Print(char *szValue);
2 void Print(int nValue);
3
4 Print('a'); // promoted to match Print(int)
```

In this case, because there is no Print(char), the char 'a' is promoted to an integer, which then matches Print(int).

3) If no promotion is found, C++ tries to find a match through standard conversion. Standard conversions include:

- Any numeric type will match any other numeric type, including unsigned (eg. int to float)
- Enum will match the formal type of a numeric type (eg. enum to float)
- Zero will match a pointer type and numeric type (eg. 0 to char*, or 0 to float)
- A pointer will match a void pointer

For example:

```
1 void Print(float fValue);
2 void Print(struct sValue);
3
4 Print('a'); // promoted to match Print(float)
```

In this case, because there is no Print(char), and no Print(int), the ‘a’ is converted to a float and matched with Print(float).

Note that all standard conversions are considered equal. No standard conversion is considered better than any of the others.

4) Finally, C++ tries to find a match through user-defined conversion. Although we have not covered classes yet, classes (which are similar to structs) can define conversions to other types that can be implicitly applied to objects of that class. For example, we might define a class X and a user-defined conversion to int.

```
1  class X; // with user-defined conversion to int
2
3  void Print(float fValue);
   void Print(int nValue);
4
5  X cValue; // declare a variable named cValue of type class X
6  Print(cValue); // cValue will be converted to an int and matched to
7  Print(int)
```

Although cValue is of type class X, because this particular class has a user-defined conversion to int, the function call Print(cValue) will resolve to the Print(int) version of the function.

We will cover the details on how to do user-defined conversions of classes when we cover classes.

Ambiguous matches

If every overloaded function has to have unique parameters, how is it possible that a call could result in more than one match? Because all standard conversions are considered equal, and all user-defined conversions are considered equal, if a function call matches multiple candidates via standard conversion or user-defined conversion, an ambiguous match will result. For example:

```
1  void Print(unsigned int nValue);
2  void Print(float fValue);
3
4  Print('a');
5  Print(0);
6  Print(3.14159);
```

In the case of `Print('a')`, C++ can not find an exact match. It tries promoting ‘a’ to an int, but there is no Print(int) either. Using a standard conversion, it can convert ‘a’ to both an unsigned int and a floating point value. Because all standard conversions are considered equal, this is an ambiguous match.

`Print(0)` is similar. 0 is an int, and there is no Print(int). It matches both calls via standard conversion.

`Print(3.14159)` might be a little surprising, as most programmers would assume it matches Print(float). But remember that all literal floating point values are doubles unless they have the 'f' suffix. 3.14159 is a double, and there is no Print(double). Consequently, it matches both calls via standard conversion.

Ambiguous matches are considered a compile-time error. Consequently, an ambiguous match needs to be disambiguated before your program will compile. There are two ways to resolve ambiguous matches:

1) Often, the best way is simply to define a new overloaded function that takes parameters of exactly the type you are trying to call the function with. Then C++ will be able to find an exact match for the function call.

2) Alternatively, explicitly cast the ambiguous parameter(s) to the type of the function you want to call. For example, to have Print(0) call the Print(unsigned int), you would do this:

```
1 Print(static_cast<unsigned int>(0)); // will call Print(unsigned int)
```

Multiple arguments

If there are multiple arguments, C++ applies the matching rules to each argument in turn. The function chosen is the one for which each argument matches at least as well as all the other functions, with at least one argument matching better than all the other functions. In other words, the function chosen must provide a better match than all the other candidate functions for at least one parameter, and no worse for all of the other parameters.

In the case that such a function is found, it is clearly and unambiguously the best choice. If no such function can be found, the call will be considered ambiguous (or a non-match).

Conclusion

Function overloading can lower a programs complexity significantly while introducing very little additional risk. Although this particular lesson is long and may seem somewhat complex (particularly the matching rules), in reality function overloading typically works transparently and without any issues. The compiler will flag all ambiguous cases, and they can generally be easily resolved.

7.7 — Default parameters

A **default parameter** is a function parameter that has a default value provided to it. If the user does not supply a value for this parameter, the default value will be used. If the user does supply a value for the default parameter, the user-supplied value is used.

Consider the following program:

```
1  void PrintValues(int nValue1, int nValue2=10)
2  {
3      using namespace std;
4      cout << "1st value: " << nValue1 << endl;
5      cout << "2nd value: " << nValue2 << endl;
6  }
7
8  int main()
9  {
10     PrintValues(1); // nValue2 will use default parameter of 10
11     PrintValues(3, 4); // override default value for nValue2
12 }
```

This program produces the following output:

```
1st value: 1
2nd value: 10
1st value: 3
2nd value: 4
```

In the first function call, the caller did not supply an argument for nValue2, so the function used the default value of 10. In the second call, the caller did supply a value for nValue2, so the user-supplied value was used.

Default parameters are an excellent option when the function needs a value that the user may or may not want to override. For example, here are a few function prototypes for which default parameters might be commonly used:

```
1  void OpenLogFile(char *strFilename="default.log");
2  int RollDie(int nSides=6);
3  void PrintString(char *strValue, Color eColor=COLOR_BLACK); // Color is an
   enum
```

A function can have multiple default parameters:

```
1  void PrintValues(int nValue1=10, int nValue2=20, int nValue3=30)
2  {
3      using namespace std;
4      cout << "Values: " << nValue1 << " " << nValue2 << " " << nValue3 <<
```

```
4  endl;
5  }
```

Given the following function calls:

```
1  PrintValues(1, 2, 3);
2  PrintValues(1, 2);
3  PrintValues(1);
4  PrintValues();
```

The following output is produced:

```
Values: 1 2 3
Values: 1 2 30
Values: 1 20 30
Values: 10 20 30
```

Note that it is impossible to supply a user-defined value for nValue3 without also supplying a value for nValue1 and nValue2. This is because C++ does not support a function call such as `PrintValues(,,3)`. This has two major consequences:

1) All default parameters must be the rightmost parameters. The following is not allowed:

```
1  void PrintValue(int nValue1=10, int nValue2); // not allowed
```

2) The leftmost default parameter should be the one most likely to be changed by the user.

Default parameters and function overloading

Functions with default parameters may be overloaded. For example, the following is allowed:

```
1  void Print(char *strString);
2  void Print(char ch=' ');
```

If there user were to call `Print()`, it would resolve to `Print(' ')`, which would print a space.

However, it is important to note that default parameters do NOT count towards the parameters that make the function unique. Consequently, the following is not allowed:

```
1  void PrintValues(int nValue);
2  void PrintValues(int nValue1, int nValue2=20);
```

If the caller were to call `PrintValues(10)`, the compiler would not be able to disambiguate whether the user wanted PrintValues(int) or PrintValues(int, 20) with the default value.

7.8 — Function Pointers

Function pointers are an advanced topic, and this section can be safely skipped or skimmed by those only looking for C++ basics.

In the lesson on pointers, you learned that a pointer is a variable that holds the address of another variable. Function pointers are similar, except that instead of pointing to variables, they point to functions!

Consider the case of an array:

```
1  int nArray[10];
```

As you now know, nArray is actually a constant pointer to a 10 element array. When we dereference the pointer (either by `*nArray` or `nArray[nIndex]`), the appropriate array element is returned.

Now consider the following function:

```
1  int foo();
```

If you guessed that foo is actually a constant pointer to a function, you are correct. When a function is called (via the () operator), the function pointer is dereferenced, and execution branches to the function.

Just like it is possible to declare a non-constant pointer to a variable, it's also possible to declare a non-constant pointer to a function. The syntax for doing so is one of the ugliest things you will ever see:

```
1  // pFoo is a pointer to a function that takes no arguments and returns an
2  integer
   int (*pFoo) ();
```

The parenthesis around *pFoo are necessary for precedence reasons, as `int *pFoo()` would be interpreted as a function named pFoo that takes no parameters and returns a pointer to an integer.

In the above snippet, pFoo is a pointer to a function that has no parameters and returns an integer. pFoo can "point" to any function that matches this signature.

Assigning a function to a function pointer

There are two primary things that you can do with a pointer to a function. First, you can assign a function to it:

```
1  int foo()
2  {
3  }
4
5  int goo()
6  {
7  }
8
9  int main()
10 {
11     int (*pFoo)() = foo; // pFoo points to function foo()
12     pFoo = goo; // pFoo now points to function goo()
13     return 0;
14 }
```

One common mistake is to do this:

```
1  pFoo = goo();
```

This would actually assign the return value from a call to function goo() to pFoo, which isn't what we want. We want pFoo to be assigned to function goo, not the return value from goo(). So no parenthesis are needed.

Note that the signature (parameters and return value) of the function pointer must match the signature of the function. Here is an example of this:

```
1  // function prototypes
2  int foo();
3  double goo();
4  int hoo(int nX);
5
6  // function pointer assignments
7  int (*pFcn1)() = foo; // okay
8  int (*pFcn2)() = goo; // wrong -- return types don't match!
9  double (*pFcn3)() = goo; // okay
10 pFcn1 = hoo; // wrong -- pFcn1 has no parameters, but hoo() does
11 int (*pFcn3)(int) = hoo; // okay
```

Calling a function using a function pointer

The second thing you can do with a function pointer is use it to actually call the function. There are two ways to do this. The first is via explicit dereference:

```
1  int foo(int nX)
2  {
3  }
4
5  int (*pFoo)(int) = foo; // assign pFoo to foo()
6
7  (*pFoo)(nValue); // call function foo(nValue) through pFoo.
```

The second way is via implicit dereference:

```
int foo(int nX)
{
}

int (*pFoo)(int) = foo; // assign pFoo to foo()

pFoo(nValue); // call function foo(nValue) through pFoo.
```

As you can see, the implicit dereference method looks just like a normal function call — which is what you'd expect, since normal function names are pointers to functions anyway! However, some older compilers do not support the implicit dereference method, but all modern compilers should.

Why use pointers to functions?

There are several cases where pointers to function can be of use. One of the most common is the case where you are writing a function to perform a task (such as sorting an array), but you want the user to be able to define how a particular part of that task will be performed (such as whether the array is sorted in ascending or descending order). Let's take a closer look at this problem as applied specifically to sorting, as an example that can be generalized to other similar problems.

All sorting algorithms work on a similar concept: the sorting algorithm walks through a bunch of numbers, does comparisons on pairs of numbers, and reorders the numbers based on the results of those comparisons. Consequently, by varying the comparison (which can be a function), we can change the way the function sorts without affecting the rest of the sorting code.

Here is our selection sort routine from a previous lesson:

```
void SelectionSort(int *anArray, int nSize)
{
    using namespace std;
    for (int nStartIndex= 0; nStartIndex < nSize; nStartIndex++)
    {
        int nBestIndex = nStartIndex;

        // Search through every element starting at nStartIndex+1
        for (int nCurrentIndex = nStartIndex + 1; nCurrentIndex < nSize;
nCurrentIndex++)
        {
            // Note that we are using the user-defined comparison here
            if (anArray[nCurrentIndex] < anArray[nBestIndex]) //
COMPARISON DONE HERE
                nBestIndex = nCurrentIndex;
        }

        // Swap our start element with our best element
        swap(anArray[nStartIndex], anArray[nBestIndex]);
    }
}
```

Now, let's replace that comparison with a function to do the comparison. Because our comparison function is going to compare two integers and return a boolean value, it will look something like this:

```
1  bool Ascending(int nX, int nY)
2  {
3      return nY > nX;
4  }
```

And here's our selection sort routine using the Ascending() function to do the comparison:

```
1  void SelectionSort(int *anArray, int nSize)
2  {
3      using namespace std;
4      for (int nStartIndex= 0; nStartIndex < nSize; nStartIndex++)
5      {
6          int nBestIndex = nStartIndex;
7
8          // Search through every element starting at nStartIndex+1
9          for (int nCurrentIndex = nStartIndex + 1; nCurrentIndex < nSize;
   nCurrentIndex++)
10         {
11             // Note that we are using the user-defined comparison here
12             if (Ascending(anArray[nCurrentIndex], anArray[nBestIndex])) //
   COMPARISON DONE HERE
13                 nBestIndex = nCurrentIndex;
14         }
15
16         // Swap our start element with our best element
17         swap(anArray[nStartIndex], anArray[nBestIndex]);
18     }
   }
```

In order to let the caller decide how the sorting will be done, instead of using our own hard-coded comparison function, we'll allow the caller to provide his own sorting function! This is done via a function pointer.

Because the caller's comparison function is going to compare two integers and return a boolean value, a pointer to such a function would look something like this:

```
1  bool (*pComparison)(int, int);
```

So, we'll allow the caller to pass our sort routine a pointer to their desired comparison function as the third parameter, and then we'll use the caller's function to do the comparison.

Here's a full example of a selection sort that uses a function pointer parameter to do a user-defined comparison, along with an example of how to call it:

```
1  #include <algorithm> // for swap
```

```
2
3   // Note our user-defined comparison is the third parameter
4   void SelectionSort(int *anArray, int nSize, bool (*pComparison)(int, int))
5   {
6       using namespace std;
7       for (int nStartIndex= 0; nStartIndex < nSize; nStartIndex++)
8       {
9           int nBestIndex = nStartIndex;
10
11          // Search through every element starting at nStartIndex+1
12          for (int nCurrentIndex = nStartIndex + 1; nCurrentIndex < nSize;
    nCurrentIndex++)
13          {
14              // Note that we are using the user-defined comparison here
15              if (pComparison(anArray[nCurrentIndex], anArray[nBestIndex]))
    // COMPARISON DONE HERE
16                  nBestIndex = nCurrentIndex;
17          }
18
19          // Swap our start element with our best element
20          swap(anArray[nStartIndex], anArray[nBestIndex]);
21      }
22  }
23
24  // Here is a comparison function that sorts in ascending order
25  // (Note: it's exactly the same as the previous Ascending() function)
26  bool Ascending(int nX, int nY)
27  {
28      return nY > nX;
29  }
30
31  // Here is a comparison function that sorts in descending order
32  bool Descending(int nX, int nY)
33  {
34      return nY < nX;
35  }
36
37  // This function prints out the values in the array
38  void PrintArray(int *pArray, int nSize)
39  {
40      for (int iii=0; iii < nSize; iii++)
41          cout << pArray[iii] << " ";
42      cout << endl;
43  }
44
45  int main()
46  {
47      using namespace std;
48
49      int anArray[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };
50
51      // Sort the array in descending order using the Descending() function
52      SelectionSort(anArray, 9, Descending);
53      PrintArray(anArray, 9);
```

```
48
49      // Sort the array in ascending order using the Ascending() function
50      SelectionSort(anArray, 9, Ascending);
51      PrintArray(anArray, 9);
52
        return 0;
    }
```

This program produces the result:

```
9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9
```

Is that cool or what? We've given the caller the ability to control how our selection sort does it's job.

The caller can even define his own "strange" comparison functions:

```
1   bool EvensFirst(int nX, int nY)
2   {
3           // if nX is not even and nY is, nY goes first
        if ((nX % 2) && !(nY % 2))
4           return false;
5
6           // if nX is even and nY is not, nX goes first
7       if (!(nX % 2) && (nY % 2))
8           return true;
9
10          // otherwise sort in Ascending order
        return Ascending(nX, nY);
11  }
12
13  int main()
14  {
15      using namespace std;
16
17      int anArray[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };
18
19      SelectionSort(anArray, 9, EvensFirst);
20      PrintArray(anArray, 9);
21
22      return 0;
    }
```

The above snippet produces the following result:

```
2 4 6 8 1 3 5 7 9
```

As you can see, using a function pointer in this context provides a nice way to allow a caller to "hook" it's own functionality into something you've previously written and tested, which helps facilitate code reuse! Previously, if you wanted to sort one array in descending order and another

in ascending order, you'd need multiple version of the sort routine. Now you can have one version that can sort any way the caller desires!

Making function pointers pretty with typedef

Let's face it — the syntax for pointers to functions is ugly. However, typedefs can be used to make pointers to functions look more like regular variables:

```
1  typedef bool (*pfcnValidate)(int, int);
```

This defines a typedef called "pfcnValidate" that is a pointer to a function that takes two ints and returns a bool.

Now instead of doing this:

```
1  bool Validate(int nX, int nY, bool (*pfcn)(int, int));
```

You can do this:

```
1  bool Validate(int nX, int nY, pfcnValidate pfcn)
```

Which reads a lot nicer!

7.9 — The stack and the heap

The memory a program uses is typically divided into four different areas:

- The code area, where the compiled program sits in memory.
- The globals area, where global variables are stored.
- The heap, where dynamically allocated variables are allocated from.
- The stack, where parameters and local variables are allocated from.

There isn't really much to say about the first two areas. The heap and the stack are where most of the interesting stuff takes place, and those are the two that will be the focus of this section.

The heap

The heap (also known as the "free store") is a large pool of memory used for dynamic allocation. In C++, when you use the new operator to allocate memory, this memory is assigned from the heap.

```
1  int *pValue = new int; // pValue is assigned 4 bytes from the heap
2  int *pArray = new int[10]; // pArray is assigned 40 bytes from the heap
```

Because the precise location of the memory allocated is not known in advance, the memory allocated has to be accessed indirectly — which is why new returns a pointer. You do not have to worry about the mechanics behind the process of how free memory is located and allocated to the user. However, it is worth knowing that sequential memory requests may not result in sequential memory addresses being allocated!

```
1  int *pValue1 = new int;
2  int *pValue2 = new int;
3  // pValue1 and pValue2 may not have sequential addresses
```

When a dynamically allocated variable is deleted, the memory is "returned" to the heap and can then be reassigned as future allocation requests are received.

The heap has advantages and disadvantages:
1) Allocated memory stays allocated until it is specifically deallocated (beware memory leaks).
2) Dynamically allocated memory must be accessed through a pointer.
3) Because the heap is a big pool of memory, large arrays, structures, or classes should be allocated here.

The stack

The call stack (usually referred to as "the stack") has a much more interesting role to play. Before we talk about the call stack, which refers to a particular portion of memory, let's talk about what a stack is.

Consider a stack of plates in a cafeteria. Because each plate is heavy and they are stacked, you can really only do one of three things:
1) Look at the surface of the top plate
2) Take the top plate off the stack
3) Put a new plate on top of the stack

In computer programming, a stack is a container that holds other variables (much like an array). However, whereas an array lets you access and modify elements in any order you wish, a stack is more limited. The operations that can be performed on a stack are identical to the ones above:

1) Look at the top item on the stack (usually done via a function called top())
2) Take the top item off of the stack (done via a function called pop())
3) Put a new item on top of the stack (done via a function called push())

A stack is a last-in, first-out (LIFO) structure. The last item pushed onto the stack will be the first item popped off. If you put a new plate on top of the stack, anybody who takes a plate from the stack will take the plate you just pushed on first. Last on, first off. As items are pushed onto a stack, the stack grows larger — as items are popped off, the stack grows smaller.

The plate analogy is a pretty good analogy as to how the call stack works, but we can actually make an even better analogy. Consider a bunch of mailboxes, all stacked on top of each other. Each mailbox can only hold one item, and all mailboxes start out empty. Furthermore, each mailbox is nailed to the mailbox below it, so the number of mailboxes can not be changed. If we can't change the number of mailboxes, how do we get a stack-like behavior?

First, we use a marker (like a post-it note) to keep track of where the bottom-most empty mailbox is. In the beginning, this will be the lowest mailbox. When we push an item onto our mailbox stack, we put it in the mailbox that is marked (which is the first empty mailbox), and move the marker up one mailbox. When we pop an item off the stack, we move the marker down one mailbox and remove the item from that mailbox. Anything below the marker is considered "on the stack". Anything at the marker or above the marker is not on the stack.

This is almost exactly analogous to how the call stack works. The call stack is a fixed-size chunk of sequential memory addresses. The mailboxes are memory addresses, and the "items" are pieces of data (typically either variables or addreses). The "marker" is a register (a small piece of memory) in the CPU known as the stack pointer. The stack pointer keeps track of where the top of the stack currently is.

The only difference between our hypothetical mailbox stack and the call stack is that when we pop an item off the call stack, we don't have to erase the memory (the equivalent of emptying the

mailbox). We can just leave it to be overwritten by the next item pushed to that piece of memory. Because the stack pointer will be below that memory location, we know that memory location is not on the stack.

So what do we push onto our call stack? Parameters, local variables, and… function calls.

The stack in action

Because parameters and local variables essentially belong to a function, we really only need to consider what happens on the stack when we call a function. Here is the sequence of steps that takes place when a function is called:

1. The address of the instruction beyond the function call is pushed onto the stack. This is how the CPU remembers where to go after the function returns.
2. Room is made on the stack for the function's return type. This is just a placeholder for now.
3. The CPU jumps to the function's code.
4. The current top of the stack is held in a special pointer called the stack frame. Everything added to the stack after this point is considered "local" to the function.
5. All function arguments are placed on the stack.
6. The instructions inside of the function begin executing.
7. Local variables are pushed onto the stack as they are defined.

When the function terminates, the following steps happen:

1. The function's return value is copied into the placeholder that was put on the stack for this purpose.
2. Everything after the stack frame pointer is popped off. This destroys all local variables and arguments.
3. The return value is popped off the stack and is assigned as the value of the function. If the value of the function isn't assigned to anything, no assignment takes place, and the value is lost.
4. The address of the next instruction to execute is popped off the stack, and the CPU resumes execution at that instruction.

Typically, it is not important to know all the details about how the call stack works. However, understanding that functions are effectively pushed on the stack when they are called and popped off when they return gives you the fundamentals needed to understand recursion, as well as some other concepts that are useful when debugging.

Stack overflow

The stack has a limited size, and consequently can only hold a limited amount of information. If the program tries to put too much information on the stack, stack overflow will result. **Stack overflow** happens when all the memory in the stack has been allocated — in that case, further allocations begin overflowing into other sections of memory.

Stack overflow is generally the result of allocating too many variables on the stack, and/or making too many nested function calls (where function A calls function B calls function C calls function D etc…) Overflowing the stack generally causes the program to crash.

Here is an example program that causes a stack overflow. You can run it on your system and watch it crash:

```
1  int main()
2  {
3      int nStack[100000000];
4      return 0;
   }
```

This program tries to allocate a huge array on the stack. Because the stack is not large enough to handle this array, the array allocation overflows into portions of memory the program is not allowed to use. Consequently, the program crashes.

The stack has advantages and disadvantages:

- Memory allocated on the stack stays in scope as long as it is on the stack. It is destroyed when it is popped off the stack.
- All memory allocated on the stack is known at compile time. Consequently, this memory can be accessed directly through a variable.
- Because the stack is relatively small, it is generally not a good idea to do anything that eats up lots of stack space. This includes allocating large arrays, structures, and classes, as well as heavy recursion.

7.10 — Recursion

A **recursive function** in C++ is a function that calls itself. Here is an example of a poorly-written recursive function:

```
1  void CountDown(int nValue)
2  {
3      using namespace std;
4      cout << nValue << endl;
5      CountDown(nValue-1);
6  }
7
8  int main(void)
9  {
10     CountDown(10);
11     return 0;
12 }
```

When CountDown(10) is called, the number 10 is printed, and CountDown(9) is called. CountDown(9) prints 9 and calls CountDown(8). CountDown(8) prints 8 and calls CountDown(7). The sequence of CountDown(n) calling CountDown(n-1) is continually repeated, effectively forming the recursive equivalent of an infinite loop.

In the lesson on the stack and the heap, you learned that every function call causes data to be placed on the call stack. Because the CountDown() function never returns (it just calls CountDown() again), this information is never being popped off the stack! Consequently, at some point, the computer will run out of stack memory, stack overflow will result, and the program will crash or terminate. On the authors machine, this program counted down to -11732 before terminating!

This program illustrates the most important point about recursive functions: you must include a termination condition, or they will run "forever" (or until the call stack runs out of memory).

Stopping a recursive function generally involves using an if statement. Here is our function redesigned with a termination condition:

```
1  void CountDown(int nValue)
2  {
3      using namespace std;
4      cout << nValue << endl;
5
6      // termination condition
7      if (nValue > 0)
8          CountDown(nValue-1);
9  }
10
11 int main(void)
```

```
{
    CountDown(10);
    return 0;
}
```

Now when we run our program, CountDown() will count down to 0 and then stop!

Let's take a look at another recursive function that is slightly more useful:

```
// return the sum of 1 to nValue
int SumTo(int nValue)
{
    if (nValue <=1)
        return nValue;
    else
        return SumTo(nValue - 1) + nValue;
}
```

Recursive programs can often be hard to figure out just by looking at them. It's often instructive to see what happens when we call a recursive function with a particular value. So let's see what happens when we call this function with nValue = 5.

SumTo(5) called, 5 <= 1 is false, so we return SumTo(4) + 5.
SumTo(4) called, 4 <= 1 is false, so we return SumTo(3) + 4.
SumTo(3) called, 3 <= 1 is false, so we return SumTo(2) + 3.
SumTo(2) called, 2 <= 1 is false, so we return SumTo(1) + 2.
SumTo(1) called, 1 <= 1 is true, so we return 1. This is the termination condition.

Now we unwind the call stack (popping each function off the call stack as it returns):
SumTo(1) returns 1.
SumTo(2) returns SumTo(1) + 2, which is 1 + 2 = 3.
SumTo(3) returns SumTo(2) + 3, which is 3 + 3 = 6.
SumTo(4) returns SumTo(3) + 4, which is 6 + 4 = 10.
SumTo(5) returns SumTo(4) + 5, which is 10 + 5 = 15.

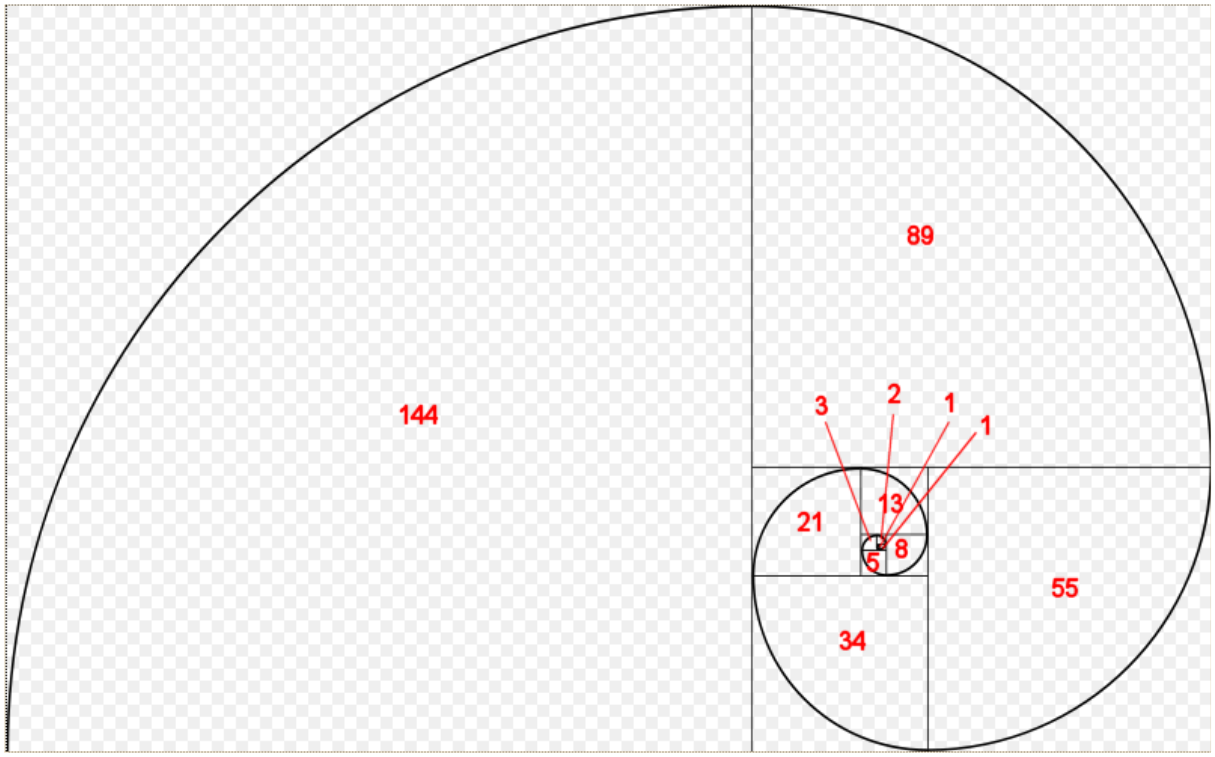
Consequently, SumTo(5) returns 15.

One question that is often asked about recursive functions is, "Why use a recursive function if you can do many of the same tasks iteratively (using a *for loop* or *while loop*)?". It turns out that you can always solve a recursive problem iteratively — however, for non-trivial problems, the recursive version is often much simpler to write (and read).

Fibonacci numbers

One of the most famous mathematical recursive algorithms is the Fibonacci sequence, as Fibonacci sequences appear in many places in nature, such as branching of trees, the spiral of shells, the fruitlets of a pineapple, an uncurling fern frond, and the arrangement of a pine cone.

Here is a picture of a Fibonacci spiral:



Each of the Fibonacci numbers is the length of the side of the square that the number appears in.

Fibonacci numbers are defined mathematically as:

F(n) =	0 if n = 0 1 if n = 1 f(n-1) + f(n-2) if n > 1

Consequently, it's rather simple to write a recursive function to calculate the nth Fibonacci number:

```
1  int Fibonacci(int nNumber)
2  {
3      if (nNumber == 0)
4          return 0;
5      if (nNumber == 1)
6          return 1;
7      return Fibonacci(nNumber-1) + Fibonacci(nNumber-2);
8  }
9
10 // And a main program to display the first 13 Fibonacci numbers
11 int main(void)
   {
       using namespace std;
       for (int iii=0; iii < 13; iii++)
```

```
12          cout << Fibonacci(iii) << " ";
13
14      return 0;
15  }
```

Running the program produces the following result:

```
0 1 1 2 3 5 8 13 21 34 55 89 144
```

Which you will note are exactly the numbers that appear in the Fibonacci spiral diagram.

While it's possible to write the Fibonacci function iteratively, it's much more difficult!

7.11 — Namespaces

Namespaces don't really fit into the functions category, but they are an important concept that we need to cover before we get to classes and object oriented programming, and this is really as good a place as any.

Pretend you are the teacher of a classroom of students. For sake of example, let's say there are two boys named "Alex". If you were to say, "Alex got an A on his test", which Alex are you referring to? Nobody knows, unless you have a way to disambiguate which Alex you mean. Perhaps you point at one, or use their last names. If the two Alex's were in different classrooms, there wouldn't be a problem — the problem is really that there are two things with the same name in the same place. And in fact, as the number of students in the classroom increase, the odds of having two students with the same first name increases exponentially.

A similar issue can arise in programming when two identifiers (variable and/or function names) with the same name are introduced into the same scope. When this happens, a **naming collision** will result, and the compiler will produce an error because it does not have enough information to resolve the ambiguity. As programs get larger and larger, the number of identifiers increases linearly, which in turn causes the probability of naming collisions to increase exponentially.

Let's take a look at an example of a naming collision. In the following example, foo.h and goo.h are the header files that contain functions that do different things but have the same name and parameters.

foo.h:

```
1  // This DoSomething() adds it's parameters
2  int DoSomething(int nX, int nY)
3  {
4      return nX + nY;
5  }
```

goo.h:

```
1  // This DoSomething() subtracts it's parameters
2  int DoSomething(int nX, int nY)
3  {
4      return nX - nY;
5  }
```

main.cpp:

```
1  #include <foo.h>
2  #include <goo.h>
3  #include <iostream>
4
```

```
4
5  int main()
6  {
7      using namespace std;
8      cout << DoSomething(4, 3); // which DoSomething will we get?
       return 0;
   }
```

If foo.h and goo.h are compiled separately, they will each compile without incident. However, by including them in the same program, we have now introduced two different functions with the same name and parameters into the program, which causes a naming collision. As a result, the compiler will issue an error:

```
c:\\VCProjects\\goo.h(4) : error C2084: function 'int __cdecl
DoSomething(int,int)' already has a body
```

In order to help address this type of problem, the concept of namespaces was introduced.

What is a namespace?

A namespace defines an area of code in which all identifiers are guaranteed to be unique. By default, all variables and functions are defined in the **global namespace**. For example, take a look at the following snippet:

```
1  int nX = 5;
2  int foo(int nX)
3  {
4      return -nX;
   }
```

Both nX and foo() are defined in the global namespace.

In the example program above that had the naming collision, when main() #included both foo.h and goo.h, the compiler tried to put both versions of DoSomething() into the global namespace, which is why the naming collision resulted.

In order to help avoid issues where two independent pieces of code have naming collisions with each other when used together, C++ allows us to declare our own namespaces via the **namespace** keyword. Anything declared inside a user-defined namespace belongs to that namespace, not the global namespace.

Here is an example of the headers in the first example rewritten using namespaces:

foo.h:

```
1  namespace Foo
2  {
3      // This DoSomething() belongs to namespace Foo
       int DoSomething(int nX, int nY)
```

```
    {
        return nX + nY;
    }
}
```

goo.h:

```
namespace Goo
{
    // This DoSomething() belongs to namespace Goo
    int DoSomething(int nX, int nY)
    {
        return nX - nY;
    }
}
```

Now the DoSomething() inside of foo.h is inside the Foo namespace, and the DoSomething() inside of goo.h is inside the Goo namespace. Let's see what happens when we recompile main.cpp:

```
int main()
{
    using namespace std;
    cout << DoSomething(4, 3) << endl; // which DoSomething will we get?
    return 0;
}
```

The answer is that we now get another error!

```
C:\\VCProjects\\Test.cpp(15) : error C2065: 'DoSomething' : undeclared
identifier
```

What happened is that when we tried to call the DoSomething() function, the compiler looked in the global namespace to see if it could find a definition of DoSomething(). However, because neither of our DoSomething() functions live in the global namespace any more, it failed to find a definition at all!

There are two different ways to tell the compiler which version of DoSomething to use.

The scope resolution operator (::)

The first way to tell the compiler to look in a particular namespace for an identifier is to use the scope resolution operator (::). This operator allows you to prefix an identifier name with the namespace you wish to use.

Here is an example of using the scope resolution operator to tell the compiler that we explicitly want to use the version of DoSomething that lives in the Foo namespace:

```
1  int main(void)
2  {
3      using namespace std;
4      cout << Foo::DoSomething(4, 3) << endl;
5      return 0;
   }
```

This produces the result:

```
7
```

If we wanted to use the version of DoSomething() that lives in Goo instead:

```
1  int main(void)
2  {
3      using namespace std;
4      cout << Goo::DoSomething(4, 3) << endl;
5      return 0;
   }
```

This produces the result:

```
1
```

The scope resolution operator is very nice because it allows us to specifically pick which namespace we want to look in. It even allows us to do the following:

```
1  int main(void)
2  {
3      using namespace std;
4      cout << Foo::DoSomething(4, 3) << endl;
5      cout << Goo::DoSomething(4, 3) << endl;
       return 0;
   }
```

This produces the result:

```
7
1
```

It is also possible to use the scope resolution operator without any namespace (eg. `::DoSomething`). In that case, it refers to the global namespace.

The using keyword

The second way to tell the compiler to look in a particular namespace for an identifier is to use the **using** keyword. The using keyword tells the compiler that if it can not find the definition for an identifier, it should look in a particular namespace to see if it exists there. For example:

```
1  int main(void)
```

```
{
    using namespace std;
    using namespace Foo; // look in namespace Foo
    cout << DoSomething(4, 3) << endl;
    return 0;
}
```

The `using namespace Foo` line causes DoSomething(4, 3) to resolve to Foo::DoSomething(4, 3). Consequently, this program prints:

```
7
```

Similarly, the following example:

```
int main(void)
{
    using namespace std;
    using namespace Goo; // look in namespace Goo
    cout << DoSomething(4, 3) << endl;
    return 0;
}
```

causes `DoSomething(4, 3)` to resolve to Goo::DoSomething(4, 3). Consequently, this program prints:

```
1
```

One more example:

```
int main(void)
{
    using namespace std;
    using namespace Foo; // look in namespace Foo
    using namespace Goo; // look in namespace Goo
    cout << DoSomething(4, 3) << endl;
    return 0;
}
```

As you might have guessed, this causes an error:

```
C:\\VCProjects\\Test.cpp(15) : error C2668: 'DoSomething' : ambiguous call to
overloaded function
```

In this case, it couldn't find DoSomething() in the global namespace, so it looked in both the Foo namespace and the Goo namespace (and the std namespace). Since DoSomething() was found in more than one namespace, the compiler couldn't figure out which one to use.

The using keyword is scoped just like a normal variable — if it is declared inside a function, it is only in effect during that function. If it is declared outside of a function, it affects the whole file from that point forward. The using keyword can save a lot of typing when you have code that

needs to use many identifiers from a particular namespace… like when you're doing input and output!

Consider the following example:

```
1  int main()
2  {
3      using namespace std;
4      cout << "Enter a number: ";
5      int nX;
6      cin >> nX;
7      cout << "You entered " << nX << endl;
8
9      return 0;
   }
```

Because of the `using namespace std` line, this function is identical to the following:

```
1  int main()
2  {
3      std::cout << "Enter a number: ";
4      int nX;
5      std::cin >> nX;
6      std::cout << "You entered " << nX << std::endl;
7
       return 0;
   }
```

However, the top example is generally considered easier to read!

Using the *using* keyword judiciously can make your code neater and easier to read. Although the using keyword can be used outside of a function to help resolve every identifier in the file, this is not recommended, as it increase the chance of identifiers from multiple namespaces conflicting, which somewhat defeats the point of namespaces in the first place.

7.12 — Handling errors (assert, cerr, exit, and exceptions)

When writing programs, it is almost inevitable that you will make mistakes. In this section, we will talk about the different kinds of errors that are made, and how they are commonly handled.

Errors fall into two categories: syntax and semantic errors.

Syntax errors

A syntax error occurs when you write a statement that is not valid according to the grammar of the C++ language. For example:

```
if 5 > 6 then write "not equal";
```

Although this statement is understandable by humans, it is not valid according to C++ syntax. The correct C++ statement would be:

```
1  if (5 > 6)
2      std::cout << "not equal";
```

Syntax errors are almost always caught by the compiler and are usually easy to fix. Consequently, we typically don't worry about them too much.

Semantic errors

A semantic error occurs when a statement is syntactically valid, but does not do what the programmer intended. For example:

```
1  for (int nCount=0; nCount<=3; nCount++)
2      std::cout << nCount << " ";
```

The programmer may have intended this statement to print `0 1 2`, but it actually prints `0 1 2 3`.

Semantic errors are not caught by the compiler, and can have any number of effects: they may not show up at all, cause the program to produce the wrong output, cause erratic behavior, corrupt data, or cause the program to crash.

It is largely the semantic errors that we are concerned with.

Semantic errors can occur in a number of ways. One of the most common semantic errors is a logic error. A **logic error** occurs when the programmer incorrectly codes the logic of a statement. The above for statement example is a logic error. Here is another example:

```
1  if (x >= 5)
2      std::cout << "x is greater than 5";
```

What happens when x is exactly 5? The conditional expression evaluates to true, and the program prints "x is greater than 5″. Logic errors can be easy or hard to locate, depending on the nature of the problem.

Another common semantic error is the violated assumption. A **violated assumption** occurs when the programmer assumes that something will be either true or false, and it isn't. For example:

```
1  char strHello[] = "Hello, world!";
2  std::cout << "Enter an index: ";
3
4  int nIndex;
5  std::cin >> nIndex;
6
7  std::cout << "Letter #" << nIndex << " is " << strHello[nIndex] <<
   std::endl;
```

See the potential problem here? The programmer has assumed that the user will enter a value between 0 and the length of "Hello, world!". If the user enters a negative number, or a large number, the array index nIndex will be out of bounds. In this case, since we are just reading a value, the program will probably print a garbage letter. But in other cases, the program might corrupt other variables, the stack, or crash.

Defensive programming is a form of program design that involves trying to identify areas where assumptions may be violated, and writing code that detects and handles any violation of those assumptions so that the program reacts in a predictable way when those violations do occur.

Detecting assumption errors

As it turns out, we can catch almost all assumptions that need to be checked in one of three locations:

- When a function has been called, the user may have passed the function parameters that are semantically meaningless.
- When a function returns, the return value may indicate that an error has occured.
- When program receives input (either from the user, or a file), the input may not be in the correct format.

Consequently, the following rules should be used when programming defensively:

- At the top of each function, check to make sure any parameters have appropriate values.
- After a function is called, check it's return value (if any), and any other error reporting mechanisms, to see if an error occured.
- Validate any user input to make sure it meets the expected formatting or range criteria.

Let's take a look at examples of each of these.

Problem: When a function is called, the user may have passed the function parameters that are semantically meaningless.

```
1  void PrintString(char *strString)
2  {
3      std::cout << strString;
   }
```

Can you identify the assumption that may be violated? The answer is that the user might pass in a NULL pointer instead of a valid C-style string. If that happens, the program will crash. Here's the function again with code that checks to make sure the function parameter is non-NULL:

```
1  void PrintString(char *strString)
2  {
3      // Only print if strString is non-null
4      if (strString)
5          std::cout << strString;
   }
```

Problem: When a function returns, the return value may indicate that an error has occured.

```
1  // Declare an array of 10 integers
2  int *panData = new int[10];
3  panData[5] = 3;
```

Can you identify the assumption that may be violated? The answer is that operator new (which actually calls a function to do the allocation) could fail if the user runs out of memory. If that happens, panData will be set to NULL, and when we use the subscript operator on panData, the program will crash. Here's a new version with error checking:

```
1  // Delcare an array of 10 integers
2  int *panData = new int[10];
3  // If something went wrong
4  if (!panData)
5      exit(2); // exit the program with error code 2
   panData[5] = 3;
```

Problem: When program receives input (either from the user, or a file), the input may not be in the correct format. Here's the sample program you saw previously:

```
1  char strHello[] = "Hello, world!";
2  std::cout << "Enter an index: ";
```

```

int nIndex;
std::cin >> nIndex;

std::cout << "Letter #" << nIndex << " is " << strHello[nIndex] <<
std::endl;
```

And here's the version that checks the user input to make sure it is valid:

```
char strHello[] = "Hello, world!";

int nIndex;
do
{
    std::cout << "Enter an index: ";
    std::cin >> nIndex;
} while (nIndex < 0 || nIndex >= strlen(strHello));

std::cout << "Letter #" << nIndex << " is " << strHello[nIndex] <<
std::endl;
```

Handling assumption errors

Now that you know where assumption errors typically occur, let's finish up by talking about different ways to handle them when they do occur. There is no best way to handle an error — it really depends on the nature of the problem.

Here are some typical responses:

1) Quietly skip the code that depends on the assumption being valid:

```
void PrintString(char *strString)
{
    // Only print if strString is non-null
    if (strString)
        std::cout << strString;
}
```

In the above example, if strString is null, we don't print anything. We have skipped the code that depends on strString being non-null.

2) If we are in a function, return an error code back to the caller and let the caller deal with the problem.

```
int g_anArray[10]; // a global array of 10 characters

int GetArrayValue(int nIndex)
{
    // use if statement to detect violated assumption
    if (nIndex < 0 || nIndex > 9)
```

```
        return -1; // return error code to caller

    return g_anArray[nIndex];
}
```

In this case, the function returns -1 if the user passes in an invalid index.

3) If we want to terminate the program immediately, the **exit** function can be used to return an error code to the operating system:

```
int g_anArray[10]; // a global array of 10 characters

int GetArrayValue(int nIndex)
{
    // use if statement to detect violated assumption
    if (nIndex < 0 || nIndex > 9)
       exit(2); // terminate program and return error number 2 to OS

    return g_anArray[nIndex];
}
```

If the user enters an invalid index, this program will terminate immediately (with no error message) and pass error code 2 to the operating system.

4) If the user has entered invalid input, ask the user to enter the input again.

```
char strHello[] = "Hello, world!";

int nIndex;
do
{
    std::cout << "Enter an index: ";
    std::cin >> nIndex;
} while (nIndex < 0 || nIndex >= strlen(strHello));

std::cout << "Letter #" << nIndex << " is " << strHello[nIndex] <<
std::endl;
```

5) cerr is another mechanism that is meant specifically for printing error messages. **cerr** is an output stream (just like cout) that is also defined in iostream.h. Typically, cerr writes the error messages on the screen (just like cout), but it can also be individually redirected to a file.

```
void PrintString(char *strString)
{
    // Only print if strString is non-null
    if (strString)
        std::cout << strString;
    else
        std::cerr << "PrintString received a null parameter";
}
```

6) If working in some kind of graphical environment (eg. MFC or SDL), it is common to pop up a message box with an error code and then terminate the program.

Assert

Using a conditional statement to detect a violated assumption, along with printing an error message and terminating the program is such a common response to problems that C++ provides a shortcut method for doing this. This shortcut is called an assert.

An **assert statement** is a preprocessor macro that evaluates a conditional expression. If the conditional expression is true, the assert statement does nothing. If the conditional expression evaluates to false, an error message is displayed and the program is terminated. This error message contains the conditional expression that failed, along with the name of the code file and the line number of the assert. This makes it very easy to tell not only what the problem was, but where in the code the problem occurred. This can help with debugging efforts immensely.

The assert functionality lives in the cassert header, and is often used both to check that the parameters passed to a function are valid, and to check that the return value of a function call is valid.

```
1  int g_anArray[10]; // a global array of 10 characters
2
3  #include <cassert> // for assert()
4  int GetArrayValue(int nIndex)
5  {
6      // we're asserting that nIndex is between 0 and 9
7      assert(nIndex >= 0 && nIndex <= 9); // this is line 7 in Test.cpp
8
9      return g_anArray[nIndex];
   }
```

If the user calls GetValue(-3), the program prints the following message:

```
Assertion failed: nIndex >= 0 && nIndex <=9, file C:\VCProjects\Test.cpp, line 7
```

We strongly encourage you to use assert statements liberally throughout your code.

Exceptions

C++ provides one more method for detecting and handling errors known as exception handling. The basic idea is that when an error occurs, it is "thrown". If the current function does not "catch" the error, the caller of the function has a chance to catch the error. If the caller does not catch the error, the caller's caller has a chance to catch the error. The error progressively moves up the stack until it is either caught and handled, or until main() fails to handle the error. If nobody handles the error, the program typically terminates with an exception error.

Exception handling is an advanced C++ topic, and we cover it in much detail in chapter 15 of this tutorial.

7.13 — Command line arguments

As you learned in the introduction to development lesson, when you compile and link your program, the compiler produces an executable file. When a program is run, execution starts at the top of the function called main(). Up to this point, we've declared main like this:

```
1  int main()
```

Notice that this version of main() takes no parameters. However, many programs need some kind of input to work with. For example, let's say you were writing a program called WordCount to count the number of words in a text file. What text file should be read and processed? The user has to have some way of telling the program which file to open. To do this, you might take this approach:

```
1  int main()
2  {
3      using namespace std;
4      cout << "Please enter a filename: ";
5      char strFilename[255];
6      cin >> strFilename;
7
8      // open file and process it
9  }
```

However, there is a major problem with this approach. Every time the program is run, the program will wait for the user to enter input. That means execution of the program can not be automated very easily. For example, if you wanted to run this program on 500 files every week, the program would stop and wait for you to enter a new filename each and every time it was run. And you'd have to do so by hand.

Command line arguments

For programs that have minimal and/or optional inputs, command line arguments offer a great way to make programs more modular. **Command line arguments** are optional string arguments that a user can give to a program upon execution. These arguments are passed by the operating system to the program, and the program can use them as input.

Programs are normally run by invoking them by name. On the command line, to run a program, you can simply type in its name. For example, to run the executable file "WordCount" that is located in the root directory of the C:\ drive on a Windows machine, you could type:

```
C:\\>WordCount
```

In order to pass command line arguments to WordCount, we simply list the command line arguments after the executable name:

```
C:\\>WordCount Myfile.txt
```

Now when WordCount is executed, Myfile.txt will be provided as a command line argument. A program can have multiple command line arguments, separated by spaces:

```
C:\\>WordCount Myfile.txt Myotherfile.txt
```

This also works for other command line operating systems, such as Linux (though your prompt and directory structure will undoubtedly vary).

If you are running your program from an IDE, the IDE should provide a way to enter command line arguments. For example, in Microsoft Visual Studio 2005, right click on your project in the solution explorer, then choose properties. Open the "Configuration Properties" tree element, and choose "Debugging". In the right pane, there is a line called "Command Arguments". You can enter your command line arguments there for testing, and they will be automatically passed to your program when you run it.

Now that you know how to provide command line arguments to a program, the next step is to access them from within our C++ program. To do that, we use a different form of main() than we've seen before. This new form of main() takes two arguments (named argc and argv by convention) as follows:

```
1  int main(int argc, char *argv[])
```

You will sometimes also see it written as:

```
1  int main(int argc, char** argv)
```

Even though these are essentially the same, we prefer the first representation because it's intuitively easier to understand.

argc is an integer parameter containing a count of the number of arguments passed to the program (think: argc = **arg**ument **c**ount). argc will always be at least 1, because the first argument is always the name of the program itself! Each command line argument the user provides will cause argc to increase by 1.

argv is where the actual arguments themselves are stored. Although the declaration of argv looks intimidating, argv is really just an array of C-style strings. The length of this array is argc.

Let's write a short program to print the value of all the command line parameters:

```
1  #include <iostream>
2
   int main(int argc, char *argv[])
```

```
{
    using namespace std;

    cout << "There are " << argc << " arguments:" << endl;

    // Loop through each argument and print its number and value
    for (int nArg=0; nArg < argc; nArg++)
        cout << nArg << " " << argv[nArg] << endl;

    return 0;
}
```

Now, when we invoke this program with the command line arguments “Myfile.txt” and “100″, the output will be as follows:

```
There are 3 arguments:
0 C:\\WordCount
1 Myfile.txt
2 100
```

Argument 0 is always the name of the current program being run. Argument 1 and 2 in this case are the two command line parameters we passed in.

Back to our previous example, let’s go ahead and partially write WordCount so it uses command line arguments instead of asking the user for input:

```
int main(int argc, char *argv[])
{
    using namespace std;
    // If the user didn't provide a filename command line argument,
    // print an error and exit.
    if (argc <= 1)
    {
        cout << "Usage: " << argv[0] << " <Filename>" << endl;
        exit(1);
    }

    char *pFilename = argv[1];

    // open file and process it
}
```

Now, when WordCount is run, it will not require any user interaction. This means we can have a batch file, script, or even another program run WordCount many times in a row (with different command line arguments) in an automated manner.

Those of you studying computer science in school (or planning on taking programming classes) may find that your professors ask you to use command line arguments to provide inputs to the program rather than using cin. Running each student’s program (when there are 50 or 100 students) and having to type in the same filenames or inputs to test whether the program works

makes for slow grading and is tedious to boot. By using command line arguments, professors and TAs can automate the process of running the student's programs on a preselected set of inputs, using another program to compare whether the output matches known correct output. This can speed up the overall grading process immensely.

7.14 — Ellipses (and why to avoid them)

In all of the functions we've seen so far, the number of parameters a function will take must be known in advance (even if they have default values). However, there are certain cases where it would be useful to be able to pass a variable number of parameters to a function. C provides a special specifier known as ellipses (aka "…") that allow us to do precisely this.

Because ellipses are rarely used, dangerous, and we strongly recommend avoiding their use, this section can be considered optional reading.

Functions that use ellipses take the form `return_type function_name(argument_list, ...)`. argument_list is one or more fixed parameters, just like normal functions use. The ellipses (which are represented as three periods in a row) must always be the last parameter in the function. Any arguments passed to the function must match the argument_list. The ellipses capture any additional arguments (if there are any). Though it is not quite accurate, it is conceptually useful to think of the ellipses as an array that holds any additional parameters beyond those in the argument_list.

The best way to learn about ellipses is by example. So let's write a simple program that uses ellipses. Let's say we want to write a function that calculates the average of a bunch of integers. We'd do it like this:

```
1  #include <cstdarg> // needed to use ellipses
2  // The ellipses must be the last parameter
3  double FindAverage(int nCount, ...)
4  {
5      long lSum = 0;
6
7      // We access the ellipses through a va_list, so let's declare one
8      va_list list;
9
10     // We initialize the va_list using va_start.  The first parameter is
11     // the list to initialize.  The second parameter is the last non-
12 ellipse
13     // parameter.
14     va_start(list, nCount);
15
16     // Loop nCount times
17     for (int nArg=0; nArg < nCount; nArg++)
18          // We use va_arg to get parameters out of our ellipses
19          // The first parameter is the va_list we're using
20          // The second parameter is the type of the parameter
21          lSum += va_arg(list, int);

     // Cleanup the va_list when we're done.
     va_end(list);
```

```
22      return static_cast<double>(lSum) / nCount;
23  }
24
25  int main()
26  {
27      cout << FindAverage(5, 1, 2, 3, 4, 5) << endl;
28      cout << FindAverage(6, 1, 2, 3, 4, 5, 6) << endl;
    }
```

This code prints

```
3
3.5
```

As you can see, this function takes a variable number of parameters! Now, let's take a look at the components that make up this example.

First, we have to include the cstdarg header. This header defines va_list, va_start, and va_end, which are macros that we need to use to access the parameters that are part of the ellipses.

We then declare our function that uses the ellipse. Remember that the argument list must be one or more fixed parameters. In this case, we're passing in a single integer that tells us how many numbers to average. The ellipses always comes last.

Note that the ellipses parameter has no name! Instead, we access the values in the ellipses through a special type known as va_list. It is conceptually useful to think of va_list as a pointer that points to the ellipses array. First, we declare a va_list, which we've called "list" for simplicity.

The next thing we need to do is make list point to our ellipses parameters. We do this by calling va_start(). va_start() takes two parameters: the va_list itself, and the name of the last non-ellipse parameter in the function. Once va_start() has been called, va_list points to the first parameter in the ellipses.

To get the value of the parameter that va_list currently points to, we use va_arg(). va_arg() also takes two parameters: the va_list itself, and the type of the parameter we're trying to access. Note that va_arg() also moves the va_list to the next parameter in the ellipses!

Finally, to clean up when we are done, we call va_end(), with va_list as the parameter.

Why ellipses are dangerous

Ellipses offer the programmer a lot of flexibility to implement functions that can take a variable number of parameters. However, this flexibility comes with some very dangerous downsides.

With regular function parameters, the compiler uses type checking to ensure the types of the function arguments match the types of the function parameters (or can be implicitly converted so they match). This helps ensure you don't pass a function an integer when it was expecting a

string, or vice versa. However, note that ellipses parameters have no type declarations. When using ellipses, the compiler completely suspends type checking for ellipses parameters. This means it is possible to send arguments of any type to the ellipses! However, the downside is that the compiler will no longer be able to warn you if you call the function with ellipses arguments that do not make sense. When using the ellipses, it is completely up to the caller to ensure the function is called with ellipses arguments that the function can handle. Obviously that leaves quite a bit of room for error (especially if the caller wasn't the one who wrote the function).

Lets look at an example of a mistake that is pretty subtle:

```
1  cout << FindAverage(6, 1.0, 2, 3, 4, 5, 6) << endl;
```

Although this may look harmless enough at first glance, note that the second argument (the first ellipse argument) is a double instead of an integer. This compiles fine, and produces a somewhat surprising result:

```
1.78782e+008
```

which is a REALLY big number. How did this happen?

As you have learned in previous lessons, a computer stores all data as a sequence of bits. A variable's type tells the computer how to translate that sequence of bits into a meaningful value. However, you just learned that the ellipses throw away the variable's type! Consequently, the only way to get a meaningful value back from the ellipses is to manually tell va_arg() what the expected type of the next parameter is. This is what the second parameter of va_arg() does. If the actual parameter type doesn't match the expected parameter type, bad things will usually happen.

In the above FindAverage program, we told va_arg() that our variables are all expected to have a type of int. Consequently, each call to va_arg() will return the next sequence of bits translated as an integer.

In this case, the problem is that the double we passed in as the first ellipse argument is 8 bytes, whereas va_arg(list, int) will only return 4 bytes of data with each call. Consequently, the first call to va_arg will only read the first 4 types of the double (producing a garbage result), and the second call to va_arg will read the second 4 bytes of the double (producing another garbage result). Thus, our overall result is garbage.

Because type checking is suspended, the compiler won't even complain if we do something completely ridiculous, like this:

```
1  int nValue = 7;
2  cout << FindAverage(6, 1.0, 2, "Hello, world!", 'G', &nValue, &FindAverage)
   << endl;
```

Believe it or not, this actually compiles just fine, and produces the following result on the author's machine:

```
1.79766e+008
```

This result epitomizes the phrase, "Garbage in, garbage out", which is a popular computer science phrase "used primarily to call attention to the fact that computers, unlike humans, will unquestioningly process the most nonsensical of input data and produce nonsensical output" (wikipedia).

So, in summary, type checking on the parameters is suspended, and we have to trust the caller to pass in the right type of parameters. If they don't, the compiler won't complain — our program will just produce garbage (or maybe crash).

As if that wasn't dangerous enough, we run into a second potential problem. Not only do the ellipses throw away the *type* of the parameters, it also throws away the *number* of parameters in the ellipses! This means we have to devise our own solution for keeping track of the number of parameters passed into the ellipses. Typically, this is done in one of two ways:

1. One of the fixed parameters is used as a parameter count (this is the solution we use in the FindAverage example above)
2. The ellipse parameters are processed until a sentinel value is reached. A **sentinel** is a special value that is used to terminate a loop when it is encountered. For example, we could pick a sentinel value of 0, and continually process ellipse parameters until we find a 0 (which should be the last value). Sentinel values only work well if you can find a sentinel value that is not a legal data value.

However, even here we run into trouble. For example, consider the following call:

For example:

```
1cout << FindAverage(6, 1, 2, 3, 4, 5) << endl;
```

On the authors machine at the time of writing, this produced the result:

```
699773
```

What happened? We told FindAverage() we were going to give it 6 values, but we only gave it 5. Consequently, the first five values that va_arg() returns were the ones we passed in. The 6th value it returns was a garbage value somewhere in the stack. Consequently, we got a garbage answer.

When using a sentinel value, if the caller forgets to include the sentinel, the loop will run continuously until it runs into garbage that matches the sentinel (or crashes).

Recommendations for safer use of ellipses

First, if possible, do not use ellipses at all! Oftentimes, other reasonable solutions are available, even if they require slightly more work. For example, in our FindAverage() program, we could have passed in a dynamically sized array of integers instead. This would have provided both

strong type checking (to make sure the caller doesn't try to do something nonsensical) while preserving the ability to pass a variable number of integers to be averaged.

Second, if you do use ellipses, do not mix expected argument types within your ellipses if possible. Doing so vastly increases the possibility of the caller inadvertently passing in data of the wrong type and va_arg() producing a garbage result.

Third, using a count parameter as part of the argument list is generally safer than using a sentinel as an ellipses parameter. This forces the user to pick an appropriate value for the count parameter, which ensures the ellipses loop will terminate after a reasonable number of iterations even if it produces a garbage value.

Basic object-oriented programming

8.1 — Welcome to object-oriented programming

All of the previous lessons up to this point have one thing in common — they have been non-object-oriented. Now that you have a basic handle on those concepts, we can proceed into object-oriented programming (OOP), where the real payoff is!

In traditional programming, programs are basically lists of instructions to the computer that define data and then work with that data. Data and the functions that work on that data are separate entities that are combined together to produce the desired result.

So what is object-oriented programming? As with so many things, it is perhaps understood most easily through use of an analogy. Take a look around you — everywhere you look are objects. Most objects have two major components to them: 1) A list of properties (eg. weight, color, size, texture, etc…), and 2) Some number of actions that either they can perform, or that can be performed on them (eg. being opened, having something poured into it, etc…). These two components are inseparable.

With traditional programming, the properties (data) and actions (functions) are separate entities, which means that traditional programming often does not provide a very intuitive representation of reality. We are intuitively used to thinking about things as objects, and expect to be able to perform actions with or on those objects.

Object-oriented programming (OOP) provides us with the ability to design "objects" that have both characteristics (sometimes called attributes, fields, or properties) and behaviors (methods or features), all tied together in one package. This allows programs to be written in a more modular fashion, which makes them easier to write and understand, and also provides a higher degree of code-reusability. Objects provide a more intuitive way to work with our data by allowing us to define how we interact with the objects, and how they interact with other objects. Object-oriented programming also brings several other useful concepts to the table: inheritance, encapsulation, abstraction, and polymorphism (language designers have a philosophy: never use a small word where a big one will do).

We will be covering all of these concepts in the upcoming tutorials over the next few chapters. It's a lot of new material, but once you've been properly familiarized with OOP, you'll never want to go back to traditional programming again.

8.2 — Classes and class members

While C++ provides a number of basic data types (eg. char, int, long, float, double, etc…) that are often sufficient for solving relatively simple problems, it can be difficult to solve complex problems using just these types. One of C++'s more useful features is the ability to define your own data types that better correspond to the problem being worked upon. You have already seen how enumerated types and structs can be used to create your own custom data types.

Here is an example of a struct used to hold a date:

```
1  struct DateStruct
2  {
3      int nMonth;
4      int nDay;
5      int nYear;
   };
```

Enumerated types and structs represent the traditional non-object-oriented programming world, as they can only hold data. If you want to initialize or manipulate this data, you either have to do so directly, or write functions that take a DateStruct as a parameter:

```
1  // Declare a DateStruct variable
2  DateStruct sToday;
3
4  // Initialize it manually
5  sToday.nMonth = 10;
6  sToday.nDay = 14;
7  sToday.nYear = 2020;
8
9  // Here is a function to initialize a date
10 void SetDate(DateStruct &sDate, int nMonth, int nDay, int Year)
11 {
12     sDate.nMonth = nMonth;
13     sDate.nDay = nDay;
14     sDate.nYear = nYear;
15 }
16
17 // Init our date to the same date using the function
   SetDate(sToday, 10, 14, 2020);
```

In the world of object-oriented programming, we often want our types to not only hold data, but provide functions that work with the data as well. In C++, this is done via the **class** keyword. Using the class keyword defines a new user-defined type called a class.

Classes

In C++, classes are very much like structs, except that classes provide much more power and flexibility. In fact, the following struct and class are effectively identical:

```
1  struct DateStruct
2  {
3      int nMonth;
4      int nDay;
5      int nYear;
6  };
7
8  class Date
9  {
10 public:
11     int m_nMonth;
       int m_nDay;
       int m_nYear;
   };
```

Note that the only difference is the *public:* keyword in the class. We will discuss it's function in the next lesson.

Just like a struct definition, a class definition does not declare any memory. It only defines what the class looks like. In order to use a class, a variable of that class type must be declared:

```
1  Date cToday; // declare a variable of class Date
2
3  // Assign values to our members using the member selector operator (.)
4  cToday.m_nMonth = 10;
5  cToday.m_nDay = 14;
   cToday.m_nYear = 2020;
```

In C++, when we declare a variable of a class, we call it **instantiating** the class. The variable itself is called an **instance** of the class. A variable of a class type is also called an **object**.

Member Functions

In addition to holding data, classes can also contain functions! Here is our Date class with a function to set the date:

```
1  class Date
2  {
3  public:
4      int m_nMonth;
5      int m_nDay;
6      int m_nYear;
7
8      void SetDate(int nMonth, int nDay, int nYear)
9      {
10         m_nMonth = nMonth;
           m_nDay = nDay;
           m_nYear = nYear;
```

```
    }
};
```

Just like member variables of a struct or class, member functions of a class are accessed using the member selector operator (.):

```
Date cToday;
cToday.SetDate(10, 14, 2020); // call SetDate() on cToday
```

Note that in the original struct version of SetDate(), we needed to pass the struct itself to the SetDate() function as the first parameter. Otherwise, SetDate() wouldn't know what DateStruct we wanted to work on.

However, in our class version of SetDate(), we do not need to pass cToday to SetDate()! Because SetDate() is being called on cToday, the member variables in SetDate() will refer to the member variables of cToday! Thus, inside function SetDate(), `m_nDay` is actually referring to `cToday.m_nDay`. If we called cTomorrow.SetDate(), `m_nDay` inside of SetDate() would refer to `cTomorrow.m_nDay`.

Using the "m_" prefix for member variables helps distinguish member variables from function parameters or local variables inside member functions. This is useful for several reasons. First, when we see an assignment to a variable with the "m_" prefix, we know that we are changing the state of the class. Second, unlike function parameters or local variables, which are declared within the function, member variables are declared in the class definition. Consequently, if we want to know how a variable with the "m_" prefix is declared, we know that we should look in the class definition instead of within the function.

By convention, class names should begin with an upper case letter.

Here's another example of a class:

```
#include <iostream>
class Employee
{
public:
    char m_strName[25];
    int m_nID;
    double m_dWage;

    // Set the employee information
    void SetInfo(char *strName, int nID, double dWage)
    {
        strncpy(m_strName, strName, 25);
        m_nID = nID;
        m_dWage = dWage;
    }

    // Print employee information to the screen
    void Print()
    {
```

```
16          using namespace std;
17          cout << "Name: " << m_strName << "  Id: " <<
18              m_nID << "  Wage: $" << m_dWage << endl;
19      }
};

int main()
{
    // Declare two employees
    Employee cAlex;
    cAlex.SetInfo("Alex", 1, 25.00);

    Employee cJoe;
    cJoe.SetInfo("Joe", 2, 22.25);

    // Print out the employee information
    cAlex.Print();
    cJoe.Print();

    return 0;
}
```

This produces the output:

```
Name: Alex  Id: 1  Wage: $25
Name: Joe  Id: 2  Wage: $22.25
```

Warning: One of the most common C++ mistakes is to forget to end all class (and struct) declarations with a semicolon. This can cause the compiler to report all sorts of weird, seemingly-unrelated errors!

8.3 — Public vs private access specifiers

Access specifiers

Consider the following struct:

```
1  struct DateStruct
2  {
3      int nMonth;
4      int nDay;
5      int nYear;
6  };
7
8  int main()
9  {
10     DateStruct sDate;
11     sDate.nMonth = 10;
12     sDate.nDay = 14;
13     sDate.nYear = 2020;
14
15     return 0;
16 }
```

In this program, we declare a DateStruct and then we directly access it's members in order to initialize them. This works because all members of a struct are public members. **Public members** are members of a struct or class that can be accessed by any function in the program.

On the other hand, consider the following almost-identical class:

```
1  class Date
2  {
3      int m_nMonth;
4      int m_nDay;
5      int m_nYear;
6  };
7
8  int main()
9  {
10     Date cDate;
11     cDate.m_nMonth = 10;
12     cDate.m_nDay = 14;
13     cDate.m_nYear = 2020;
14
15     return 0;
16 }
```

If you were to compile this program, you would receive an error. This is because by default, all members of a class are private. **Private members** are members of a class that can only be

accessed by other functions within the class. Because main() is not a member of the Date class, it does not have access to Date's private members.

Although class members are private by default, we can make them public by using the public keyword:

```
1  class Date
2  {
3  public:
4      int m_nMonth; // public
5      int m_nDay; // public
6      int m_nYear; // public
7  };
8
9  int main()
10 {
11     Date cDate;
12     cDate.m_nMonth = 10; // okay because m_nMonth is public
13     cDate.m_nDay = 14;  // okay because m_nDay is public
14     cDate.m_nYear = 2020;  // okay because m_nYear is public
15
16     return 0;
17 }
```

Because Date's members are now public, they can be accessed by main().

One of the primary differences between classes and structs is that classes can explicitly use **access specifiers** to restrict who can access members of a class. C++ provides 3 different access specifier keywords: public, private, and protected. We will discuss the protected access specifier when we cover inheritance.

Here is an example of a class that uses all three access specifiers:

```
1  class Access
2  {
3     int m_nA; // private by default
4     int GetA() { return m_nA; } // private by default
5
6  private:
7     int m_nB; // private
8     int GetB() { return m_nB; } // private
9
10 protected:
11    int m_nC; // protected
12    int GetC() { return m_nC; } // protected
13
14 public:
15    int m_nD; // public
16    int GetD() { return m_nD; } // public
17
18 };
```

```
17
18  int main()
19  {
20      Access cAccess;
21      cAccess.m_nD = 5; // okay because m_nD is public
22      std::cout << cAccess.GetD(); // okay because GetD() is public
23
24      cAccess.m_nA = 2; // WRONG because m_nA is private
25      std::cout << cAccess.GetB(); // WRONG because GetB() is private

        return 0;
    }
```

Each of the members “acquires” the access level of the previous access specifier. It is common convention to list private members first.

Why would you want to restrict access to class members? Oftentimes you want to declare members that are for “internal class use only”. For example, when writing a string class, it is common to declare a private member named m_nLength that holds the length of the string. If m_nLength were public, anybody could change the length of the string without changing the actual string! This could cause all sorts of bizarre problems. Consequently, the m_nLength is made private so that only functions within the String class can alter m_nLength.

The group of public members of a class are often referred to as a “public interface”. Because only public members can be accessed outside of the class, the public interface defines how programs using the class will interface with the class.

8.4 — Access functions and encapsulation

Access functions

An **access function** is a short public function whose job is to return the value of a private member variable. For example, in the above mentioned String class, you might see something like this:

```
1  class String
2  {
3  private:
4      char *m_chString; // a dynamically allocated string
5      int m_nLength; // the length of m_chString
6
7  public:
8      int GetLength() { return m_nLength; }
9  };
```

GetLength() is an access function that simply returns the value of m_nLength.

Access functions typically come in two flavors: getters and setters. **Getters** are functions that simply return the value of a private member variable. Setters are functions that simply set the value of a private member variable.

Here's an example class that has some getters and setters:

```
1  class Date
2  {
3  private:
4      int m_nMonth;
5      int m_nDay;
6      int m_nYear;
7
8  public:
9      // Getters
10     int GetMonth() { return m_nMonth; }
11     int GetDay() { return m_nDay; }
12     int GetYear() { return m_nYear; }
13
14     // Setters
15     void SetMonth(int nMonth) { m_nMonth = nMonth; }
16     void SetDay(int nDay) { m_nDay = nDay; }
17     void SetYear(int nYear) { m_nYear = nYear; }
18 };
```

Why bother to make a member variable private if we're going to provide public access functions to it? The answer is: "encapsulation".

Encapsulation

In real life, it is common to use something without knowing how it actually works. For example, your TV remote provides buttons that allow you to do things like turn your TV on and off and adjust the volume. However, the details of how the remote is actually implemented is hidden away. This is useful because it allows you to use the remote without having to worry about the details of why it works or how it was implemented. If gnomes broke into your house in the middle of the night and replaced the internals of your TV remote with a new (but compatible) technology, you'd probably never even notice.

Encapsulation is the idea of hiding the details of how something is implemented and instead exposing an interface to the user. This allows the user to use the item without having to worry about how it is implemented.

In C++, access specifiers allow us to implement encapsulation within our classes. This is typically done by making ALL member variables of a class private, and providing public functions (often access functions) that allow the user to work with the class. Although this may seem more burdensome than providing public access directly, doing so actually provides several very useful benefits that help encourage class reusability and maintainability.

Perhaps most importantly, sometimes it turns out that the initial implementation of a class is too slow or uses too much memory, and a more complex solution is needed. Encapsulating the implementation means that the implementation of a class can be completely changed, and so long as the interface remains the same, the users of the class do not have to worry about the changes at all!

Consider this simple example:

```
1  class Change
2  {
3  public:
4      int m_nValue;
5  };
6
7  int main()
8  {
9      Change cChange;
10     cChange.m_nValue = 5;
11     std::cout << cChange.m_nValue << std::endl;
12 };
```

While this program works fine, what would happen if we decided to rename m_nValue? We'd also break our program! Encapsulation gives us the ability to change our classes without breaking all the code that uses them.

Here is the encapsulated version of this class that uses access functions to access m_nValue:

```
1  class Change
2  {
```

```
2  private:
3      int m_nValue;
4
5  public:
6      void SetValue(int nValue) { m_nValue = nValue; }
7      int GetValue() { return m_nValue; }
8  };
9
10 int main()
11 {
12     Change cChange;
13     cChange.SetValue(5);
14     std::cout << cChange.GetValue() << std::endl;
15 }
```

Now when we decide to rename m_nValue, we only need to change SetValue and GetValue() to reflect the change. Our program does not need to be changed at all!

Second, hiding the details about how a class is implemented means a programmer can use the class without knowing how it was implemented. This lowers the time needed to learn how to use a class, and makes the class much easier to work with.

Third, encapsulation helps prevent accidental changes and misuse. Because the member variables can not be accessed directly, this helps prevent inadvertent changing of values. Furthermore, it is often the case that when a value is modified, other values also need to be updated. For example, in a typical String class, when the string is modified, the length also needs to be updated. If the user has direct access to the string, he/she may forget to update the length when the string is changed. However, an interface function that allows the user to change the string can automatically update the length whenever the string is changed, meaning the user doesn't even have to worry about it!

And finally, encapsulation helps you debug the program when something goes wrong. Often when a program does not work correctly, it is because one of our member variables has an incorrect value. If everyone is able to access the variable directly, tracking down which piece of code modified the variable can be difficult. However, if everybody has to call the same function to modify a variable, you can simply breakpoint that function and watch as each caller changes the value until you see where it goes wrong.

As you can see, encapsulation provides a lot of benefits for just a little bit of effort. In particular, the ability to change the implementation details of the class without affecting any of the programs that use the class is paramount to code maintainability!

8.5 — Constructors

Constructors

A **constructor** is a special kind of class member function that is executed when an object of that class is instantiated. Constructors are typically used to initialize member variables of the class to appropriate default values, or to allow the user to easily initialize those member variables to whatever values are desired.

Unlike normal functions, constructors have specific rules for how they must be named:
1) Constructors should always have the same name as the class (with the same capitalization)
2) Constructors have no return type (not even void)

A constructor that takes no parameters (or has all optional parameters) is called a **default constructor**.

Here is an example of a class that has a default constructor:

```
1  class Fraction
2  {
3  private:
4      int m_nNumerator;
5      int m_nDenominator;
6
7  public:
8      Fraction() // default constructor
9      {
10         m_nNumerator = 0;
11         m_nDenominator = 1;
12     }
13
14     int GetNumerator() { return m_nNumerator; }
15     int GetDenominator() { return m_nDenominator; }
16     double GetFraction() { return static_cast<double>(m_nNumerator) /
   m_nDenominator; }
17 };
```

This class was designed to hold a fractional value as an integer numerator and denominator. We have defined a default constructor named Fraction (the same as the class). When we create an instance of the Fraction class, this default constructor will be called immediately after memory is allocated, and our object will be initialized. For example, the following snippet:

```
1  Fraction cDefault; // calls Fraction() constructor
2  std::cout << cDefault.GetNumerator() << "/" << cDefault.GetDenominator() <<
   std::endl;
```

produces the output:

0/1

Note that our numerator and denominator were initialized with the values we set in our default constructor! This is such a useful feature that almost every class includes a default constructor. Without a default constructor, the numerator and denominator would have garbage values until we explicitly assigned them reasonable values.

Constructors with parameters

While the default constructor is great for ensuring our classes are initialized with reasonable default values, often times we want instances of our class to have specific values. Fortunately, constructors can also be declared with parameters. Here is an example of a constructor that takes two integer parameters that are used to initialize the numerator and denominator:

```
1  #include <cassert>
2  class Fraction
3  {
4  private:
5      int m_nNumerator;
6      int m_nDenominator;
7
8  public:
9      Fraction() // default constructor
10     {
11         m_nNumerator = 0;
12         m_nDenominator = 1;
13     }
14
15     // Constructor with parameters
16     Fraction(int nNumerator, int nDenominator=1)
17     {
18         assert(nDenominator != 0);
19         m_nNumerator = nNumerator;
20         m_nDenominator = nDenominator;
21     }
22
23     int GetNumerator() { return m_nNumerator; }
24     int GetDenominator() { return m_nDenominator; }
25     double GetFraction() { return static_cast<double>(m_nNumerator) /
26 m_nDenominator; }
27 };
```

Note that we now have two constructors: a default constructor that will be called in the default case, and a second constructor that takes two parameters. These two constructors can coexist peacefully in the same class due to function overloading. In fact, you can define as many constructors as you want, so long as each has a unique signature (number and type of parameters).

So how do we use this constructor with parameters? It's simple:

```
1  Fraction cFiveThirds(5, 3); // calls Fraction(int, int) constructor
```

This particular fraction will be initialized to the fraction 5/3!

Note that we have made use of a default value for the second parameter of the constructor with parameters, so the following is also legal:

```
1  Fraction Six(6); // calls Fraction(int, int) constructor
```

In this case, our default constructor is actually somewhat redundant. We could simplify this class as follows:

```
1  #include <cassert>
2  class Fraction
3  {
4  private:
5      int m_nNumerator;
6      int m_nDenominator;
7
8  public:
9      // Default constructor
10     Fraction(int nNumerator=0, int nDenominator=1)
11     {
12         assert(nDenominator != 0);
13         m_nNumerator = nNumerator;
14         m_nDenominator = nDenominator;
15     }
16
17     int GetNumerator() { return m_nNumerator; }
       int GetDenominator() { return m_nDenominator; }
       double GetFraction() { return static_cast<double>(m_nNumerator) /
   m_nDenominator; }
   };
```

This constructor has been defined in a way that allows it to serve as both a default and a non-default constructor!

```
1  Fraction cDefault; // will call Fraction(0, 1)
2  Fraction cSix(6); // will call Fraction(6, 1)
3  Fraction cFiveThirds(5,3); // will call Fraction(5,3)
```

Classes without default constructors

What happens if we do not declare a default constructor and then instantiate our class? The answer is that C++ will allocate space for our class instance, but will not initialize the members of the class (similar to what happens when you declare an int, double, or other basic data type). For example:

```
1  class Date
2  {
3  private:
       int m_nMonth;
```

```
4       int m_nDay;
5       int m_nYear;
6   };
7
8   int main()
9   {
10      Date cDate;
11      // cDate's member variables now contain garbage
12      // Who knows what date we'll get?
13
14      return 0;
15  }
```

In the above example, because we declared a Date object, but there is no default constructor, m_nMonth, m_nDay, and m_nYear were never initialized. Consequently, they will hold garbage values. Generally speaking, this is why providing a default constructor is almost always a good idea:

```
1   class Date
2   {
3   private:
4       int m_nMonth;
5       int m_nDay;
6       int m_nYear;
7
8   public:
9       Date(int nMonth=1, int nDay=1, int nYear=1970)
10      {
11          m_nMonth = nMonth;
12          m_nDay = nDay;
13          m_nYear = nYear;
14      }
15  };
16
17  int main()
18  {
19      Date cDate; // cDate is initialized to Jan 1st, 1970 instead of
    garbage
20
21      Date cToday(9, 5, 2007); // cToday is initialized to Sept 5th, 2007
22
23      return 0;
24  }
```

8.6 — Destructors

A **destructor** is another special kind of class member function that is executed when an object of that class is destroyed. They are the counterpart to constructors. When a variable goes out of scope, or a dynamically allocated variable is explicitly deleted using the delete keyword, the class destructor is called (if it exists) to help clean up the class before it is removed from memory. For simple classes, a destructor is not needed because C++ will automatically clean up the memory for you. However, if you have dynamically allocated memory, or if you need to do some kind of maintenance before the class is destroyed (eg. closing a file), the destructor is the perfect place to do so.

Like constructors, destructors have specific naming rules:
1) The destructor must have the same name as the class, preceded by a tilde (~).
2) The destructor can not take arguments.
3) The destructor has no return type.

Note that rule 2 implies that only one destructor may exist per class, as there is no way to overload destructors since they can not be differentiated from each other based on arguments.

Let's take a look at a simple string class that uses a destructor:

```
1  class MyString
2  {
3  private:
4      char *m_pchString;
5      int m_nLength;
6
7  public:
8      MyString(const char *pchString="")
9      {
10         // Find the length of the string
11         // Plus one character for a terminator
12         m_nLength = strlen(pchString) + 1;
13
14         // Allocate a buffer equal to this length
15         m_pchString = new char[m_nLength];
16
17         // Copy the parameter into our internal buffer
18         strncpy(m_pchString, pchString, m_nLength);
19
20         // Make sure the string is terminated
21         m_pchString[m_nLength-1] = '\0';
22     }

       ~MyString() // destructor
       {
           // We need to deallocate our buffer
```

```
        delete[] m_pchString;

        // Set m_pchString to null just in case
        m_pchString = 0;
    }

    char* GetString() { return m_pchString; }
    int GetLength() { return m_nLength; }
};
```

Let's take a look at how this class is used:

```
int main()
{
    MyString cMyName("Alex");
    std::cout << "My name is: " << cMyName.GetString() << std::endl;
    return 0;
} // cMyName destructor called here!
```

This program produces the result:

```
My name is: Alex
```

On the first line, we instantiate a new MyString class and pass in the C-style string "Alex". This calls the constructor, which dynamically allocates memory to hold the string being passed in. We must use dynamic allocation here because we do not know in advance how long of a string the user is going to pass in.

At the end of main(), cMyName goes out of scope. This causes the ~MyString() destructor to be called, which deletes the buffer that we allocated in the constructor!

Constructor and destructor timing

As mentioned previously, the constructor is called when an object is created, and the destructor is called when an object is destroyed. In the following example, we use cout statements inside the constructor and destructor to show this:

```
class Simple
{
private:
    int m_nID;

public:
    Simple(int nID)
    {
        std::cout << "Constructing Simple " << nID<< std::endl;
        m_nID = nID;
    }

    ~Simple()
    {
```

```
13          std::cout << "Destructing Simple" << m_nID << std::endl;
14      }
15
16      int GetID() { return m_nID; }
17  };
18
19  int main()
20  {
21      // Allocate a Simple on the stack
22      Simple cSimple(1);
23      std::cout << cSimple.GetID() << std::endl;
24
25      // Allocate a Simple dynamically
26      Simple *pSimple = new Simple(2);
27      std::cout << pSimple->GetID() << std::endl;
28      delete pSimple;

        return 0;
    } // cSimple goes out of scope here
```

This program produces the following result:

```
Constructing Simple 1
1
Constructing Simple 2
2
Destructing Simple 2
Destructing Simple 1
```

Note that “Simple 1″ is destroyed after “Simple 2″ because we deleted pSimple before the end of the function, whereas cSimple was not destroyed until the end of main().

As you can see, when constructors and destructors are used together, your classes can initialize and clean up after themselves without the programmer having to do any special work! This reduces the probability of making an error, and makes classes easy to use.

8.7 — The hidden “this” pointer

One of the big questions that new programmers often ask is, “When a member function is called, how does C++ know which object it was called on?”. The answer is that C++ utilizes a hidden pointer named “this”! Let’s take a look at “this” in more detail.

The following is a simple class that holds an integer and provides a constructor and access functions. Note that no destructor is needed because C++ can clean up integers for us.

```
1  class Simple
2  {
3  private:
4      int m_nID;
5
6  public:
7      Simple(int nID)
8      {
9          SetID(nID);
10     }
11
12     void SetID(int nID) { m_nID = nID; }
13     int GetID() { return m_nID; }
14 };
```

Here’s a sample program that uses this class:

```
1  int main()
2  {
3      Simple cSimple(1);
4      cSimple.SetID(2);
5      std::cout << cSimple.GetID() << std::endl;
6  }
```

Let’s take a closer look at the following line: `cSimple.SetID(2);`. Although it looks like this function only has one parameter, it actually has two! When you call `cSimple.SetID(2);`, C++ internally converts this to `SetID(&cSimple, 2);`. Note that this is just a normal function call where C++ has added a parameter, and automatically passed in the address of the class object!

Since C++ converts the function call, it also needs to convert the function itself. It does so like this:

```
1  void SetID(int nID) { m_nID = nID; }
```

becomes:

```
1  void SetID(Simple* const this, int nID) { this->m_nID = nID; }
```

C++ has added a new parameter to the function. The added parameter is a pointer to the class object the class function is working with, and it is always named "this". The **this pointer** is a hidden pointer inside every class member function that points to the class object the member function is working with.

Note that `m_nID` (which is a class member variable) has been converted to `this->m_nID`. Since "this" is currently pointing to cSimple, this actually resolves to cSimple->m_nID, which is exactly what we wanted!

Most of the time, you never need to explicitly reference the "this" pointer. However, there are a few occasions where it can be useful:

1) If you have a constructor (or member function) that has a parameter of the same name as a member variable, you can disambiguate them by using "this":

```
1  class Something
2  {
3  private:
4      int nData;
5
6  public:
7      Something(int nData)
8      {
9          this->nData = nData;
       }
   };
```

Note that our constructor is taking a parameter of the same name as a member variable. In this case, "nData" refers to the parameter, and "this->nData" refers to the member variable. Although this is acceptable coding practice, we find using the "m_" prefix on all member variable names provides a better solution by preventing duplicate names altogether!

2) Occasionally it can be useful to have a function return the object it was working with. Returning *this will return a reference to the object that was implicitly passed to the function by C++.

One use for this feature is that it allows a series of functions to be "chained" together, so that the output of one function becomes the input of another function! The following is somewhat more advanced and can be considered optional material at this point.

Consider the following class:

```
1  class Calc
2  {
3  private:
4      int m_nValue;
5
6  public:
       Calc() { m_nValue = 0; }
```

```
7
8       void Add(int nValue) { m_nValue += nValue; }
9       void Sub(int nValue) { m_nValue -= nValue; }
10      void Mult(int nValue) { m_nValue *= nValue; }
11
12      int GetValue() { return m_nValue; }
};
```

If you wanted to add 5, subtract 3, and multiply by 4, you'd have to do this:

```
1  Calc cCalc;
2  cCalc.Add(5);
3  cCalc.Sub(3);
4  cCalc.Mult(4);
```

However, if we make each function return *this, we can chain the calls together. Here is the new version of Calc with "chainable" functions:

```
1  class Calc
2  {
3  private:
4      int m_nValue;
5
6  public:
7      Calc() { m_nValue = 0; }
8
9      Calc& Add(int nValue) { m_nValue += nValue; return *this; }
10     Calc& Sub(int nValue) { m_nValue -= nValue; return *this; }
11     Calc& Mult(int nValue) { m_nValue *= nValue; return *this; }
12
13     int GetValue() { return m_nValue; }
};
```

Note that Add(), Sub() and Mult() are now returning *this, which is a reference to the class itself. Consequently, this allows us to do the following:

```
1  Calc cCalc;
2  cCalc.Add(5).Sub(3).Mult(4);
```

We have effectively condensed three lines into one expression! Let's take a closer look at how this works.

First, cCalc.Add(5) is called, which adds 5 to our m_nValue. Add() then returns *this, which is a reference to cCalc. Our expression is now cCalc.Sub(3).Mult(4). cCalc.Sub(3) subtracts 3 from m_nValue and returns cCalc. Our expression is now cCalc.Mult(4). cCalc.Mult(4) multiplies m_nValue by 4 and returns cCalc, which is then ignored. However, since each function modified cCalc as it was executed, cCalc now contains the value ((0 + 5) – 3) * 4), which is 8.

Although this is a pretty contrived example, chaining functions in such a manner is common with String classes. For example, it is possible to overload the + operator to do a string append. If the + operator returns *this, then it becomes possible to write expressions like:

```
1 cMyString = "Hello " + strMyName + " welcome to " + strProgramName + ".";
```

And it is pretty easy to see the benefit in being able to do that! We will cover overloading the + operator (and other operators) in a future lesson.

The important point to take away from this lesson is that the “this” pointer is a hidden parameter of any member function. Most of the time, you will not need to access it directly. It’s worth noting that “this” is a const pointer — you can change the value of the object it points to, but you can not make it point to something else!

8.8 — Constructors (Part II)

Private constructors

Occasionally, we do not want users outside of a class to use particular constructors. To enforce this behavior, we can make constructors private. Just like regular private functions, private constructors can only be accessed from within the class. Let's take a look at an example of this:

```
1  class Book
2  {
3  private:
4      int m_nPages;
5
6      // This constructor can only be used by Book's members
7      Book() // private default constructor
8      {
9           m_nPages = 0;
10     }
11
12 public:
13     // This constructor can be used by anybody
14     Book(int nPages) // public non-default constructor
15     {
16         m_nPages = nPages;
17     }
18 };
19
20 int main()
21 {
22     Book cMyBook; // fails because default constructor Book() is private
23     Book cMyOtherBook(242); // okay because Book(int) is public
24
25     return 0;
26 }
```

One problem with public constructors is that they do not provide any way to control how many of a particular class may be created. If a public constructor exists, it can be used to instantiate as many class objects as the user desires. Often it is useful to restrict users to being able to create only one instance of a particular class. Classes that can only be instantiated once are called **singletons**. There are many ways to implement singletons, but most of them involve use of a private (or protected) constructor to prevent users from instantiating as many of the class as they want.

Constructor chaining and initialization issues

When you instantiate a new object, the object's constructor is called implicitly by the C++ compiler. Let's take a look at two related situations that often cause problems for new programmers:

First, sometimes a class has a constructor which needs to do the same work as another constructor, plus something extra. The process of having one constructor call another constructor is called **constructor chaining**. Although some languages such as C# support constructor chaining, C++ does not. If you try to chain constructors, it will usually compile, but it will not work right, and you will likely spend a long time trying to figure out why, even with a debugger. However, constructors *are* allowed to call non-constructor functions in the class. Just be careful that any members the non-constructor function uses have already been initialized.

Although you may be tempted to copy code from the first constructor into the second constructor, having duplicate code makes your class harder to understand and more burdensome to maintain. The best solution to this issue is to create a non-constructor function that does the common initialization, and have both constructors call that function.

For example, the following:

```
1  class Foo
2  {
3  public:
4      Foo()
5      {
6          // code to do A
7      }
8
9      Foo(int nValue)
10     {
11         // code to do A
12         // code to do B
13     }
14 };
```

becomes:

```
1  class Foo
2  {
3  public:
4      Foo()
5      {
6          DoA();
7      }
8
9      Foo(int nValue)
10     {
11         DoA();
12         // code to do B
13     }
14
15     void DoA()
```

```
    {
        // code to do A
    }
};
```

Code duplication is kept to a minimum, and no chained constructor calls are needed.

Second, you may find yourself in the situation where you want to write a member function to re-initialize a class back to default values. Because you probably already have a constructor that does this, you may be tempted to try to call the constructor from your member function. As mentioned, chaining constructor calls are illegal in C++. You could copy the code from the constructor in your function, which would work, but lead to duplicate code. The best solution in this case is to move the code from the constructor to your new function, and have the constructor call your function to do the work of initializing the data:

```
class Foo
{
public:
    Foo()
    {
        Init();
    }

    Foo(int nValue)
    {
        Init();
        // do something with nValue
    }

    void Init()
    {
        // code to init Foo
    }
};
```

It is fairly common to include an Init() function that initializes member variables to their default values, and then have each constructor call that Init() function before doing it's parameter-specific tasks. This minimizes code duplication and allows you to explicitly call Init() from wherever you like.

One small caveat: be careful when using Init() functions and dynamically allocated memory. Because Init() functions can be called by anyone at any time, dynamically allocated memory may or may not have already been allocated when Init() is called. Be careful to handle this situation appropriately — it can be slightly confusing, since a non-null pointer could be either dynamically allocated memory or an uninitialized pointer!

Note: C++11 now supports chaining constructors via the delegating constructors functionality.

8.9 — Class code and header files

Defining member functions outside the class definition

All of the classes that we have written so far have been simple enough that we have been able to implement the functions directly inside the class definition itself. For example, here's our ubiquitous Date class:

```
1  class Date
2  {
3  private:
4      int m_nMonth;
5      int m_nDay;
6      int m_nYear;
7
8      Date() { } // private default constructor
9
10 public:
11     Date(int nMonth, int nDay, int nYear)
12     {
13         SetDate(nMonth, nDay, nYear);
14     }
15
16     void SetDate(int nMonth, int nDay, int nYear)
17     {
18         m_nMonth = nMonth;
19         m_nDay = nDay;
20         m_nYear = nYear;
21     }
22
23     int GetMonth() { return m_nMonth; }
24     int GetDay()   { return m_nDay; }
25     int GetYear() { return m_nYear; }
26 };
```

However, as classes get longer and more complicated, mixing the definition and the implementation details makes the class harder to manage and work with. Typically, when looking at a class definition (for an already written class), you don't care how things are implemented — you want to know how to use the class, which involves only it's definition. In this case, all of the implementation details just get in the way.

Fortunately, C++ provides a way to separate the definition portion of the class from the implementation portion. This is done by defining the class member functions outside of the class definition. To do so, simply define the member functions of the class as if they were normal functions, but prefix the class name to the function using the scope operator (::) (same as for a namespace).

Here is our Date class with the Date constructor and SetDate() function defined outside of the class definition. Note that the prototypes for these functions still exist inside the class definition, but the actual implementation has been moved outside:

```
1  class Date
2  {
3  private:
4      int m_nMonth;
5      int m_nDay;
6      int m_nYear;
7
8      Date() { } // private default constructor
9
10 public:
11     Date(int nMonth, int nDay, int nYear);
12
13     void SetDate(int nMonth, int nDay, int nYear);
14
15     int GetMonth() { return m_nMonth; }
16     int GetDay()   { return m_nDay; }
17     int GetYear() { return m_nYear; }
18 };
19
20 // Date constructor
21 Date::Date(int nMonth, int nDay, int nYear)
22 {
23     SetDate(nMonth, nDay, nYear);
24 }
25
26 // Date member function
27 void Date::SetDate(int nMonth, int nDay, int nYear)
28 {
29     m_nMonth = nMonth;
30     m_nDay = nDay;
31     m_nYear = nYear;
32 }
```

This is pretty straightforward. Because access functions are often only one line, they are typically left in the class definition, even though they could be moved outside.

Here is another example:

```
1  class Calc
2  {
3  private:
4      int m_nValue;
5
6  public:
7      Calc() { m_nValue = 0; }
8
9      void Add(int nValue) { m_nValue += nValue; }
10     void Sub(int nValue) { m_nValue -= nValue; }
11     void Mult(int nValue) { m_nValue *= nValue; }
```

```
10
11      int GetValue() { return m_nValue; }
};
```

becomes:

```
1  class Calc
2  {
3  private:
4      int m_nValue;
5
6  public:
7      Calc() { m_nValue = 0; }
8
9      void Add(int nValue);
10     void Sub(int nValue);
11     void Mult(int nValue);
12
13     int GetValue() { return m_nValue; }
14 };
15
16 void Calc::Add(int nValue)
17 {
18     m_nValue += nValue;
19 }
20
21 void Calc::Sub(int nValue)
22 {
23     m_nValue -= nValue;
24 }
25
26 void Calc::Mult(int nValue)
27 {
28     m_nValue *= nValue;
29 }
```

In this case, we left the default constructor in the class definition because it was so short.

Putting class definitions in a header file

In the lesson on header files, you learned that you can put functions inside header files in order to reuse them in multiple files or even multiple projects. Classes are no different. Class definitions can be put in header files in order to facilitate reuse in multiple files or multiple projects. Traditionally, the class definition is put in a header file of the same name as the class, and the member functions defined outside of the class are put in a .cpp file of the same name as the class. You may sometimes hear the term "one file, one class", which alludes to the principle of putting classes in their own separate header/code files.

Here's our Date class again, broken into a .cpp and .h file:

Date.h:

```
1  #ifndef DATE_H
2  #define DATE_H
3
4  class Date
5  {
6  private:
7      int m_nMonth;
8      int m_nDay;
9      int m_nYear;
10
11     Date() { } // private default constructor
12
13 public:
14     Date(int nMonth, int nDay, int nYear);
15
16     void SetDate(int nMonth, int nDay, int nYear);
17
18     int GetMonth() { return m_nMonth; }
19     int GetDay()  { return m_nDay; }
20     int GetYear() { return m_nYear; }
21 };
22
23 #endif
```

Date.cpp:

```
1  #include "Date.h"
2
3  // Date constructor
4  Date::Date(int nMonth, int nDay, int nYear)
5  {
6      SetDate(nMonth, nDay, nYear);
7  }
8
9  // Date member function
10 void Date::SetDate(int nMonth, int nDay, int nYear)
11 {
12     m_nMonth = nMonth;
13     m_nDay = nDay;
14     m_nYear = nYear;
15 }
```

Now any other header or code file that wants to use the date class can simply `#include "Date.h"`. Note that Date.cpp also needs to be compiled into any project that uses Date.h so the linker knows how Date is implemented. Don't forget the header guards on the .h file!

In future lessons, most of our classes will be defined in the .cpp file, with all the functions implemented directly in the class definition. This is just for convenience and to keep the examples short. In real projects, it is much more common for classes to be put in their own code and header files.

8.10 — Const class objects and member functions

In the lesson on passing arguments by reference, we covered the merits of passing function parameters as const variables. To recap, making variables const ensures their values are not accidentally changed. This is particularly important when passing variables by reference, as callers generally will not expect the values they pass to a function to be changed.

Just like the built-in data types (int, double, char, etc…), class objects can be made const by using the const keyword. All const variables must be initialized at time of creation. In the case of built-in data types, initilization is done through explicit or implicit assignment:

```
const int nValue = 5; // initialize explicitly
const int nValue2(7); // initialize implictly
```

In the case of classes, this initialization is done via constructors:

```
const Date cDate; // initialize using default constructor
const Date cDate2(10, 16, 2020); // initialize using parameterized constructor
```

If a class is not initialized using a parameterized constructor, a public default constructor *must* be provided — if no public default constructor is provided in this case, a compiler error will occur.

Once a const class object has been initialized via constructor, any attempt to modify the member variables of the object is disallowed, as it would violate the constness of the object. This includes both changing member variables directly (if they are public), or calling member functions that sets the value of member variables:

```
class Something
{
public:
    int m_nValue;

    Something() { m_nValue = 0; }

    void ResetValue() { m_nValue = 0; }
    void SetValue(int nValue) { m_nValue = nValue; }

    int GetValue() { return m_nValue; }
};

int main()
{
    const Something cSomething; // calls default constructor
```

```
15
16      cSomething.m_nValue = 5; // violates const
17      cSomething.ResetValue(); // violates const
18      cSomething.SetValue(5); // violates const
19
20      return 0;
    }
```

All three of the above lines involving cSomething are illegal because they violate the constness of cSomething by attempting to change a member variable or calling a member function that attempts to change a member variable.

Now, consider the following call:

```
1std::cout << cSomething.GetValue();
```

Surprisingly, this will cause a compile error! This is because const class objects can only call const member functions, GetValue() has not been marked as a const member function. A **const member function** is a member function that guarantees it will not change any class variables or call any non-const member functions.

To make GetValue() a const member function, we simply append the const keyword to the function prototype:

```
1   class Something
2   {
3   public:
4       int m_nValue;
5
6       Something() { m_nValue = 0; }
7
8       void ResetValue() { m_nValue = 0; }
9       void SetValue(int nValue) { m_nValue = nValue; }
10
        int GetValue() const { return m_nValue; }
    };
```

Now GetValue() has been made a const member function, which means we can call it on any const objects.

Const member functions declared outside the class definition must specify the const keyword on both the function prototype in the class definition and on the function prototype in the code file:

```
1   class Something
2   {
3   public:
4       int m_nValue;
5
6       Something() { m_nValue = 0; }
```

```
7       void ResetValue() { m_nValue = 0; }
8       void SetValue(int nValue) { m_nValue = nValue; }
9
10      int GetValue() const;
11  };
12
13  int Something::GetValue() const
14  {
        return m_nValue;
    }
```

Any const member function that attempts to change a member variable or call a non-const member function will cause a compiler error to occur. For example:

```
1   class Something
2   {
3   public:
4       int m_nValue;
5
6       void ResetValue() const { m_nValue = 0; }
    };
```

In this example, ResetValue() has been marked as a const member function, but it attempts to change m_nValue. This will cause a compiler error.

Note that constructors should not be marked as const. This is because const objects should initialize their member variables, and a const constructor would not be able to do so.

Finally, although it is not done very often, it is possible to overload a function in such a way to have a const and non-const version of the same function:

```
1   class Something
2   {
3   public:
4       int m_nValue;
5
6       const int& GetValue() const { return m_nValue; }
7       int& GetValue() { return m_nValue; }
    };
```

The const version of the function will be called on any const objects, and the non-const version will be called on any non-const objects:

```
1   Something cSomething;
2   cSomething.GetValue(); // calls non-const GetValue();
3
4   const Something cSomething2;
    cSomething2.GetValue(); // calls const GetValue();
```

Overloading a function with a const and non-const version is typically done when the return value needs to differ in constness. In the example above, the const version of GetValue() returns a const reference, whereas the non-const version returns a non-const reference.

Let's take a look at making our date class member functions const so they can be used with const Date objects:

```
1  class Date
2  {
3  private:
4      int m_nMonth;
5      int m_nDay;
6      int m_nYear;
7
8      Date() { } // private default constructor
9
10 public:
11     Date(int nMonth, int nDay, int nYear)
12     {
13         SetDate(nMonth, nDay, nYear);
14     }
15
16     void SetDate(int nMonth, int nDay, int nYear)
17     {
18         m_nMonth = nMonth;
19         m_nDay = nDay;
20         m_nYear = nYear;
21     }
22
23     int GetMonth() { return m_nMonth; }
24     int GetDay()   { return m_nDay; }
25     int GetYear() { return m_nYear; }
26 };
```

The only non-constructor member functions that do not modify member variables (or call functions that modify member variables) are the access functions. Consequently, they should be made const. Here is the const version of our Date class:

```
1  class Date
2  {
3  private:
4      int m_nMonth;
5      int m_nDay;
6      int m_nYear;
7
8      Date() { } // private default constructor
9
10 public:
11     Date(int nMonth, int nDay, int nYear)
12     {
13         SetDate(nMonth, nDay, nYear);
14     }
```

```
13
14      void SetDate(int nMonth, int nDay, int nYear)
15      {
16          m_nMonth = nMonth;
17          m_nDay = nDay;
18          m_nYear = nYear;
19      }
20
21      int GetMonth() const { return m_nMonth; }
        int GetDay() const { return m_nDay; }
        int GetYear() const { return m_nYear; }
    };
```

The following is now valid code:

```
1   void PrintDate(const Date &cDate)
2   {
3       // although cDate is const, we can call const member functions
4       std::cout << cDate.GetMonth() << "/" <<
5           cDate.GetDay() << "/" <<
6           cDate.GetYear() << std::endl;
7   }
8
9   int main()
10  {
11      const Date cDate(10, 16, 2020);
12      PrintDate(cDate);
13
        return 0;
    }
```

8.11 — Static member variables

Static keyword in C

In the lesson on file scope and the static keyword, you learned that static variables keep their values and are not destroyed even after they go out of scope. For example:

```
1  int GenerateID()
2  {
3      static int s_nID = 0;
4      return s_nID++;
5  }
6
7  int main()
8  {
9      std::cout << GenerateID() << std::endl;
10     std::cout << GenerateID() << std::endl;
11     std::cout << GenerateID() << std::endl;
12     return 0;
13 }
```

This program prints:

```
0
1
2
```

Note that s_nID has kept it's value across multiple function calls.

The static keyword has another meaning when applied to global variables — it changes them from global scope to file scope. Because global variables are typically avoided by competent programmers, and file scope variables are just global variables limited to a single file, the static keyword is typically not used in this capacity.

Static member variables

C++ introduces two new uses for the static keyword when applied to classes: static member variables, and static member classes. Before we go into the static keyword as applied to member variables, first consider the following class:

```
1  class Something
2  {
3  private:
4      int m_nValue;
5  public:
6      Something() { m_nValue = 0; }
7  };
```

```
7
8  int main()
9  {
10     Something cFirst;
11     Something cSecond;
       return 0;
   }
```

When we instantiate a class object, each object gets it's own copy of all normal member variables. In this case, because we have declared two Something class objects, we end up with two copies of m_nValue — one inside cFirst, and one inside cSecond. cFirst->m_nValue is different than cSecond->m_nValue.

Member variables of a class can be made static by using the static keyword. Static member variables only exist once in a program regardless of how many class objects are defined! One way to think about it is that all objects of a class share the static variables. Consider the following program:

```
1  class Something
2  {
3  public:
4      static int s_nValue;
5  };
6
7  int Something::s_nValue = 1;
8
9  int main()
10 {
11     Something cFirst;
12     cFirst.s_nValue = 2;
13
14     Something cSecond;
15     std::cout << cSecond.s_nValue;

       return 0;
   }
```

This program produces the following output:

```
2
```

Because s_nValue is a static member variable, s_nValue is shared between all objects of the class. Consequently, cFirst.s_nValue is the same as cSecond.s_nValue. The above program shows that the value we set using cFirst can be accessed using cSecond!

Although you can access static members through objects of the class type, this is somewhat misleading. `cFirst.s_nValue` implies that s_nValue belongs to cFirst, and this is really not the case. s_nValue does not belong to any object. In fact, s_nValue exists even if there are no objects of the class have been instantiated!

Consequently, it is better to think of static members as belonging to the class itself, not the objects of the class. Because s_nValue exists independently of any class objects, it can be accessed directly using the class name and the scope operator:

```
1  class Something
2  {
3  public:
4      static int s_nValue;
5  };
6
7  int Something::s_nValue = 1;
8
9  int main()
10 {
11     Something::s_nValue = 2;
12     std::cout << Something::s_nValue;
13     return 0;
14 }
```

In the above snippet, s_nValue is referenced by class name rather than through an object. Note that we have not even instantiated an object of type Something, but we are still able to access and use Something::s_nValue. This is the preferred method for accessing static members.

Initializing static member variables

Because static member variables are not part of the individual objects, you must explicitly define the static member if you want to initialize it to a non-zero value. The following line in the above example initializes the static member to 1:

```
1 int Something::s_nValue = 1;
```

This initializer should be placed in the code file for the class (eg. Something.cpp). In the absense of an initializing line, C++ will initialize the value to 0.

An example of static member variables

Why use static variables inside classes? One great example is to assign a unique ID to every instance of the class. Here's an example of that:

```
1  class Something
2  {
3  private:
4      static int s_nIDGenerator;
5      int m_nID;
6  public:
7      Something() { m_nID = s_nIDGenerator++; }
8
9      int GetID() const { return m_nID; }
10 };
11
```

```
10 int Something::s_nIDGenerator = 1;
11
12 int main()
13 {
14     Something cFirst;
15     Something cSecond;
16     Something cThird;
17
18     using namespace std;
19     cout << cFirst.GetID() << endl;
20     cout << cSecond.GetID() << endl;
21     cout << cThird.GetID() << endl;
22     return 0;
23 }
```

This program prints:

```
1
2
3
```

Because s_nIDGenerator is shared by all Something objects, when a new Something object is created, it's constructor grabs the current value out of s_nIDGenerator and then increments the value for the next object. This guarantees that each Something object receives a unique id (incremented in the order of creation). This can really help when debugging multiple items in an array, as it provides a way to tell multiple objects of the same class type apart!

Static member variables can also be useful when the class needs to utilize an internal lookup table (eg. to look up the name of something, or to find a pre-calculated value). By making the lookup table static, only one copy exists for all objects, rather than a copy for each object instantiated. This can save substantial amounts of memory.

8.12 — Static member functions

In the previous lesson on static member variables, you learned that classes can have member variables that are shared across all objects of that class type. However, what if our static member variables are private? Consider the following example:

```
1  class Something
2  {
3  private:
4      static int s_nValue;
5
6  };
7
8  int Something::s_nValue = 1; // initializer
9
10 int main()
11 {
12     // how do we access Something::s_nValue?
13 }
```

In this case, we can't access Something::s_nValue directly from main(), because it is private. Normally we access private members through public member functions. While we could create a normal public member function to access s_nValue, we'd then need to instantiate an object of the class type to use the function! We can do better. In this case, the answer to the problem is that we can also make member functions static.

Like static member variables, static member functions are not attached to any particular object. Here is the above example with a static member function accessor:

```
1  class Something
2  {
3  private:
4      static int s_nValue;
5  public:
6      static int GetValue() { return s_nValue; }
7  };
8
9  int Something::s_nValue = 1; // initializer
10
11 int main()
12 {
13     std::cout << Something::GetValue() << std::endl;
14 }
```

Because static member functions are not attached to a particular object, they can be called directly by using the class name and the scope operator. Like static member variables, they can also be called through objects of the class type, though this is not recommended.

Static member functions have two interesting quirks worth noting. First, because static member functions are not attached to an object, they have no *this* pointer! This makes sense when you think about it — the *this* pointer always points to the object that the member function is working on. Static member functions do not work on an object, so the *this* pointer is not needed.

Second, static member functions can only access static member variables. They can not access non-static member variables. This is because non-static member variables must belong to a class object, and static member functions have no class object to work with!

Here's another example using static member variables and functions:

```
1  class IDGenerator
2  {
3  private:
4      static int s_nNextID;
5
6  public:
7       static int GetNextID() { return s_nNextID++; }
8  };
9
10 // We'll start generating IDs at 1
11 int IDGenerator::s_nNextID = 1;
12
13 int main()
14 {
15     for (int i=0; i < 5; i++)
16         cout << "The next ID is: " << IDGenerator::GetNextID() << endl;
17
18     return 0;
19 }
```

This program prints:

```
The next ID is: 1
The next ID is: 2
The next ID is: 3
The next ID is: 4
The next ID is: 5
```

Note that because all the data and functions in this class are static, we don't need to instantiate an object of the class to make use of it's functionality! This class utilizes a static member variable to hold the value of the next ID to be assigned, and provides a static member function to return that ID and increment it.

Be careful when writing classes with all static members like this. Although such "pure static classes" can be useful, they also come with some potential downsides. First, because all of the members belong to the class, and the class is accessible from anywhere in the program, it's essentially the equivalent of declaring a global variable of the class type. In the section on global variables, you learned that global variables are dangerous because one piece of code can change the value of the global variable and end up breaking another piece of seemingly unrelated code.

The same holds true for pure static classes. Second, because all static members are instantiated only once, there is no way to have multiple copies of a pure static class (without cloning the class and renaming it). For example, if you needed two independent IDGenerators, this would not be possible.

8.13 — Friend functions and classes

For much of this chapter, we've been preaching the virtues of keeping your data private. However, you may occasionally find situations where you will find you have classes and functions that need to work very closely together. For example, you might have a class that stores data, and a function (or another class) that displays the data on the screen. Although the storage class and display code have been separated for easier maintenance, the display code is really intimately tied to the details of the storage class. Consequently, there isn't much to gain by hiding the storage classes details from the display code.

In situations like this, there are two options:
1) Have the display code use the publicly exposed functions of the storage class. However, this has several potential downsides. First, these public member functions have to be defined, which takes time, and can clutter up the interface of the storage class. Second, the storage class may have to expose functions for the display code that it doesn't really want accessible to anybody else. There is no way to say "this function is meant to be used by the display class only".

2) Alternatively, using friend classes and friend functions, you can give your display code access to the private details of the storage class. This lets the display code directly access all the private members and functions of the storage class! In this lesson, we'll take a closer look at how this is done.

Friend functions

A **friend function** is a function that can access the private members of a class as though it were a member of that class. In all other regards, the friend function is just like a normal function. A friend function may or may not be a member of another class. To declare a friend function, simply use the *friend* keyword in front of the prototype of the function you wish to be a friend of the class. It does not matter whether you declare the friend function in the private or public section of the class.

Here's an example of using a friend function:

```
1  class Accumulator
2  {
3  private:
4      int m_nValue;
5  public:
6      Accumulator() { m_nValue = 0; }
7      void Add(int nValue) { m_nValue += nValue; }
8
9      // Make the Reset() function a friend of this class
10     friend void Reset(Accumulator &cAccumulator);
   };
```

```
// Reset() is now a friend of the Accumulator class
void Reset(Accumulator &cAccumulator)
{
    // And can access the private data of Accumulator objects
    cAccumulator.m_nValue = 0;
}
```

In this example, we've declared a function named Reset() that takes an object of class Accumulator, and sets the value of m_nValue to 0. Because Reset() is not a member of the Accumulator class, normally Reset() would not be able to access the private members of Accumulator. However, because Accumulator has specifically declared this Reset() function to be a friend of the class, the Reset() function is given access to the private members of Accumulator.

Note that we have to pass an Accumulator object to Reset(). This is because Reset() is not a member function. It does not have a *this pointer, nor does it have an Accumulator object to work with, unless given one.

While this example is pretty contrived, here's another example that's a lot closer to something you'll see again in the near future, when we talk about operator overloading:

```
class Value
{
private:
    int m_nValue;
public:
    Value(int nValue) { m_nValue = nValue; }
    friend bool IsEqual(const Value &cValue1, const Value &cValue2);
};

bool IsEqual(const Value &cValue1, const Value &cValue2)
{
    return (cValue1.m_nValue == cValue2.m_nValue);
}
```

In this example, we declare the IsEqual() function to be a friend of the Value class. IsEqual() takes two Value objects as parameters. Because IsEqual() is a friend of the Value class, it can access the private members of all Value objects. In this case, it uses that access to do a comparison on the two objects, and returns true if they are equal.

A function can be a friend of more than one class at the same time. For example, consider the following example:

```
class Humidity;

class Temperature
{
private:
    int m_nTemp;
public:
```

```
    Temperature(int nTemp) { m_nTemp = nTemp; }

    friend void PrintWeather(Temperature &cTemperature, Humidity &cHumidity);
};

class Humidity
{
private:
    int m_nHumidity;
public:
    Humidity(int nHumidity) { m_nHumidity = nHumidity; }

    friend void PrintWeather(Temperature &cTemperature, Humidity &cHumidity);
};

void PrintWeather(Temperature &cTemperature, Humidity &cHumidity)
{
    std::cout << "The temperature is " << cTemperature.m_nTemp <<
        " and the humidity is " << cHumidity.m_nHumidity << std::endl;
}
```

There are two things worth noting about this example. First, because PrintWeather is a friend of both classes, it can access the private data from objects of both classes. Second, note the following line at the top of the example:

```
class Humidity;
```

This is a class prototype that tells the compiler that we are going to define a class called Humidity in the future. Without this line, the compiler would tell us it doesn't know what a Humidity is when parsing the prototype for PrintWeather() inside the Temperature class. Class prototypes serve the same role as function prototypes — they tell the compiler what something looks like so it can be used now and defined later. However, unlike functions, classes have no return types or parameters, so class prototypes are always simply `class ClassName`, where ClassName is the name of the class.

Friend classes

It is also possible to make an entire class a friend of another class. This gives all of the members of the friend class access to the private members of the other class. Here is an example:

```
class Storage
{
private:
    int m_nValue;
    double m_dValue;
public:
    Storage(int nValue, double dValue)
    {
        m_nValue = nValue;
```

```
        m_dValue = dValue;
    }

    // Make the Display class a friend of Storage
    friend class Display;
};

class Display
{
private:
    bool m_bDisplayIntFirst;

public:
    Display(bool bDisplayIntFirst) { m_bDisplayIntFirst =
bDisplayIntFirst; }

    void DisplayItem(Storage &cStorage)
    {
        if (m_bDisplayIntFirst)
            std::cout << cStorage.m_nValue << " " << cStorage.m_dValue <<
std::endl;
        else // display double first
            std::cout << cStorage.m_dValue << " " << cStorage.m_nValue <<
std::endl;
    }
};
```

Because the Display class is a friend of Storage, any of Display's members that use a Storage class object can access the private members of Storage directly. Here's a simple program that shows the above classes in use:

```
int main()
{
    Storage cStorage(5, 6.7);
    Display cDisplay(false);

    cDisplay.DisplayItem(cStorage);

    return 0;
}
```

This program produces the following result:

```
6.7 5
```

A few additional notes on friend classes. First, even though Display is a friend of Storage, Display has no direct access to the *this pointer of Storage objects. Second, just because Display is a friend of Storage, that does not mean Storage is also a friend of Display. If you want two classes to be friends of each other, both must declare the other as a friend. Finally, if class A is a friend of B, and B is a friend of C, that does not mean A is a friend of C.

Be careful when using friend functions and classes, because it allows the friend function or class to violate encapsulation. If the details of the class change, the details of the friend will also be forced to change. Consequently, limit your use of friend functions and classes to a minimum.

8.14 — Anonymous variables and objects

In some cases, we need a variable only temporarily. For example, consider the following situation:

```
1  int Add(int nX, int nY)
2  {
3      int nSum = nX + nY;
4      return nSum;
5  }
6
7  int main()
8  {
9      using namespace std;
10     cout << Add(5, 3);
11
12     return 0;
13 }
```

In the Add() function, note that the nSum variable is really only used as a temporary placeholder variable. It doesn't contribute much — rather, it's only function is to transfer the result of the expression to the return value.

There is actually an easier way to write the Add() function using an anonymous variable. An **anonymous variable** is a variable that is given no name. Anonymous variables in C++ have "expression scope", meaning they are destroyed at the end of the expression in which they are created. Consequently, they must be used immediately!

Here is the Add() function rewritten using an anonymous variable:

```
1  int Add(int nX, int nY)
2  {
3      return nX + nY;
4  }
```

When the expression `nX + nY` is evaluated, the result is placed in an anonymous, unnamed variable. A copy of the anonymous variable is returned to the caller by value.

This not only works with return values, but also with function parameters. For example, instead of this:

```
1  void PrintValue(int nValue)
2  {
3      using namespace std;
4      cout << nValue;
5  }
```

```
5
6   int main()
7   {
8       int nSum = 5 + 3;
9       PrintValue(nSum);
10      return 0;
11  }
12
```

We can write this:

```
1   int main()
2   {
3       PrintValue(5 + 3);
4       return 0;
5   }
```

In this case, the expression 5 + 3 is evaluated to produce the result 8, which is placed in an anonymous variable. A copy of this anonymous variable is then passed to the PrintValue() function, which prints the value 8.

Note how much cleaner this keeps our code — we don't have to litter the code with temporary variables that are only used once.

Anonymous class objects

Although our prior examples have been with built-in data types, it is possible to construct anonymous objects of our own class types as well. This is done by creating objects like normal, but omitting the variable name.

```
1   Cents cCents(5); // normal variable
2   Cents(7); // anonymous variable
```

In the above code, `Cents(7)` will create an anonymous Cents object, initialize it with the value 7, and then destroy it. In this context, that isn't going to do us much good. So let's take a look at an example where it can be put to good use:

```
1   class Cents
2   {
3   private:
4       int m_nCents;
5
6   public:
7       Cents(int nCents) { m_nCents = nCents; }
8
9       int GetCents() { return m_nCents; }
10  };
```

```
10
11  Cents Add(Cents &c1, Cents &c2)
12  {
13      Cents cTemp(c1.GetCents() + c2.GetCents());
14      return cTemp;
15  }
16
17  int main()
18  {
19      Cents cCents1(6);
20      Cents cCents2(8);
21      Cents cCentsSum = Add(cCents1, cCents2);
22      std::cout << "I have " << cCentsSum.GetCents() << " cents." <<
23  std::endl;

        return 0;
    }
```

Note that this example is very similar to the prior one using integers. In this case, our Add() function is constructing a short-lived cTemp variable that only serves as a placeholder. We are also using a cCentsSum variable in main().

We can simplify this program by using anonymous variables:

```
1   class Cents
2   {
3   private:
4       int m_nCents;
5
6   public:
7       Cents(int nCents) { m_nCents = nCents; }
8
9       int GetCents() { return m_nCents; }
10  };
11
12  Cents Add(Cents &c1, Cents &c2)
13  {
14      return Cents(c1.GetCents() + c2.GetCents());
15  }
16
17  int main()
18  {
19      Cents cCents1(6);
20      Cents cCents2(8);
21      std::cout << "I have " << Add(cCents1, cCents2).GetCents() << "
    cents." << std::endl;

        return 0;
    }
```

This version of Add() functions identically to the one above, except it uses an anonymous Cents value instead of a named variable. Also note that in main(), we no longer use a named

cCentsSum variable as temporary storage. Instead, we use the return value of Add() anonymously!

As a result, our program is shorter, cleaner, and generally easier to follow (once you understand the concept).

In C++, anonymous variables are primarily used either to pass or return values without having to create lots of temporary variables to do so. However, it is worth noting that anonymous objects can only be passed or returned by value! If a variable is passed or returned by reference or address, a named variable must be used instead. It is also worth noting that because anonymous variables have expression scope, if you need to reference a value in multiple expressions, you will have to use a named variable.

Operator overloading

9.1 — Introduction to operator overloading

In the lesson on function overloading, you learned that you can create multiple functions of the same name that work differently depending on parameter type. **Operator overloading** allows the programmer to define how operators (such as +, -, ==, =, and !) should interact with various data types. Because operators in C++ are implemented as functions, operator overloading works very analogously to function overloading.

Consider the following example:

```
1 int nX = 2;
2 int nY = 3;
3 cout << nX + nY << endl;
```

C++ already knows how the plus operator (+) should be applied to integer operands — the compiler adds nX and nY together and returns the result. Now consider this case:

```
1 Mystring cString1 = "Hello, ";
2 Mystring cString2 = "World!";
3 cout << cString1 + cString2 << endl;
```

What would you expect to happen in this case? The intuitive expected result is that the string "Hello, World!" is printed on the screen. However, because Mystring is a user-defined class, C++ does not know what operator + should do. We need to tell it how the + operator should work with two objects of type Mystring. Once an operator has been overloaded, C++ will call the appropriate overloaded version of the operator based on parameter type. If you add two integers, the integer version of operator plus will be called. If you add two Mystrings, the Mystring version of operator plus will be called.

Almost any operator in C++ can be overloaded. The exceptions are: arithmetic if (?:), sizeof, scope (::), member selector (.), and member pointer selector (.*). You can overload the + operator to concatenate your user-defined string class, or add two Fraction class objects together. You can overload the << operator to make it easy to print your class to the screen (or a file). You can overload the equality operator (==) to compare two objects. This makes operator overloading one of the most useful features in C++ -- simply because it allows you to work with your classes in a more intuitive way.

Before we go into more details, there are a few things to keep in mind going forward.

First, at least one of the operands in any overloaded operator must be a user-defined type. This means you can not overload the plus operator to work with one integer and one double. However, you could overload the plus operator to work with an integer and a Mystring.

Second, you can only overload the operators that exist. You can not create new operators. For example, you could not create an operator ** to do exponents.

Third, all operators keep their current precedence and associativity, regardless of what they're used for. For example, the bitwise XOR operator (^) could be overloaded to do exponents, except it has the wrong precedence and associativity and there is no way to change this.

Within those confines, you will still find plenty of useful functionality to overload for your custom classes!

Operators as functions

When you see the expression `nX + nY`, you can translate this in your head to `operator+(nX, nY)` (where operator+ is the name of the function). Similarly `dX + dY` becomes `operator+(dX, dY)`. Even though both expressions call a function named operator+(), function overloading is used to resolve the function calls to different versions of the function based on parameter type(s). For example, In the lesson on arithmetic operators, you learned that C++ does integer and floating point division differently. This works because the operator/() function has two flavors — one that is called for integer operands, and one for floating point operands.

More generally, when evaluating an expression with operators, C++ looks at the operands around the operator to see what type they are. If *all* operands are built-in types, C++ calls a built-in routine. If *any* of the operands are user data types (eg. one of your classes), it looks to see whether the class has an overloaded operator function that it can call. If the compiler finds an overloaded operator whose parameters match the types of the operands, it calls that function. Otherwise, it produces a compiler error.

9.2 — Overloading the arithmetic operators

Some of the most commonly used operators in C++ are the arithmetic operators — that is, the plus operator (+), minus operator (-), multiplication operator (*), and division operator (/). Note that all of the arithmetic operators are binary operators — meaning they take two operands — one on each side of the operator. All four of these operators are overloaded in the exact same way.

Overloading operators using friend functions

When the operator does not modify its operands, the best way to overload the operator is via friend function. None of the arithmetic operators modify their operands (they just produce and return a result), so we will utilize the friend function overloaded operator method here.

The following example shows how to overload operator plus (+) in order to add two "Cents" objects together:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5
6  public:
7      Cents(int nCents) { m_nCents = nCents; }
8
9      // Add Cents + Cents
10     friend Cents operator+(const Cents &c1, const Cents &c2);
11
12     int GetCents() { return m_nCents; }
13 };
14
15 // note: this function is not a member function!
16 Cents operator+(const Cents &c1, const Cents &c2)
17 {
18     // use the Cents constructor and operator+(int, int)
19     return Cents(c1.m_nCents + c2.m_nCents);
20 }
21
22 int main()
23 {
24     Cents cCents1(6);
25     Cents cCents2(8);
26     Cents cCentsSum = cCents1 + cCents2;
27     std::cout << "I have " << cCentsSum .GetCents() << " cents." <<
28 std::endl;
29
30     return 0;
31 }
```

This produces the result:

```
I have 14 cents.
```

Overloading the plus operator (+) is as simple as declaring a function named operator+, giving it two parameters of the type of the operands we want to add, picking an appropriate return type, and then writing the function.

In the case of our Cents object, implementing our operator+() function is very simple. First, the parameter types: in this version of operator+, we are going to add two Cents objects together, so our function will take two objects of type Cents. Second, the return type: our operator+ is going to return a result of type Cents, so that's our return type.

Finally, implementation: to add two Cents objects together, we really need to add the m_nCents member from each Cents object. Because our overloaded operator+() function is a friend of the class, we can access the m_nCents member of our parameters directly. Also, because m_nCents is an integer, and C++ knows how to add integers together using the built-in version of the plus operator that works with integer operands, we can simply use the + operator to do the adding.

Overloading the subtraction operator (-) is simple as well:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5
6  public:
7      Cents(int nCents) { m_nCents = nCents; }
8
9      // overload Cents + Cents
10     friend Cents operator+(const Cents &c1, const Cents &c2);
11
12     // overload Cents - Cents
13     friend Cents operator-(const Cents &c1, const Cents &c2);
14
15     int GetCents() { return m_nCents; }
16 };
17
18 // note: this function is not a member function!
19 Cents operator+(const Cents &c1, const Cents &c2)
20 {
21     // use the Cents constructor and operator+(int, int)
22     return Cents(c1.m_nCents + c2.m_nCents);
23 }
24
25 // note: this function is not a member function!
   Cents operator-(const Cents &c1, const Cents &c2)
   {
       // use the Cents constructor and operator-(int, int)
       return Cents(c1.m_nCents - c2.m_nCents);
   }
```

Overloading the multiplication operator (*) and division operator (/) are as easy as defining functions for operator* and operator/.

Overloading operators for operands of different types

Often it is the case that you want your overloaded operators to work with operands that are different types. For example, if we have Cents(4), we may want to add the integer 6 to this to produce the result Cents(10).

When C++ evaluates the expression `x + y`, x becomes the first parameter, and y becomes the second parameter. When x and y have the same type, it does not matter if you add x + y or y + x — either way, the same version of operator+ gets called. However, when the operands have different types, x + y is not the same as y + x.

For example, `Cents(4) + 6` would call operator+(Cents, int), and `6 + Cents(4)` would call operator+(int, Cents). Consequently, whenever we overload binary operators for operands of different types, we actually need to write two functions — one for each case. Here is an example of that:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5
6  public:
7      Cents(int nCents) { m_nCents = nCents; }
8
9      // Overload cCents + int
10     friend Cents operator+(const Cents &cCents, int nCents);
11
12     // Overload int + cCents
13     friend Cents operator+(int nCents, const Cents &cCents);
14
15     int GetCents() { return m_nCents; }
16 };
17
18 // note: this function is not a member function!
19 Cents operator+(const Cents &cCents, int nCents)
20 {
21     return Cents(cCents.m_nCents + nCents);
22 }
23
24 // note: this function is not a member function!
25 Cents operator+(int nCents, const Cents &cCents)
26 {
27     return Cents(cCents.m_nCents + nCents);
28 }
29
30 int main()
31 {
32     Cents c1 = Cents(4) + 6;
```

```
    Cents c2 = 6 + Cents(4);
    std::cout << "I have " << c1.GetCents() << " cents." << std::endl;
    std::cout << "I have " << c2.GetCents() << " cents." << std::endl;

    return 0;
}
```

Note that both overloaded functions have the same implementation — that's because they do the same thing, they just take their parameters in a different order.

Another example

Let's take a look at another example:

```
class MinMax
{
private:
    int m_nMin; // The min value seen so far
    int m_nMax; // The max value seen so far

public:
    MinMax(int nMin, int nMax)
    {
        m_nMin = nMin;
        m_nMax = nMax;
    }

    int GetMin() { return m_nMin; }
    int GetMax() { return m_nMax; }

    friend MinMax operator+(const MinMax &cM1, const MinMax &cM2);
    friend MinMax operator+(const MinMax &cM, int nValue);
    friend MinMax operator+(int nValue, const MinMax &cM);
};

MinMax operator+(const MinMax &cM1, const MinMax &cM2)
{
    // Get the minimum value seen in cM1 and cM2
    int nMin = cM1.m_nMin < cM2.m_nMin ? cM1.m_nMin : cM2.m_nMin;

    // Get the maximum value seen in cM1 and cM2
    int nMax = cM1.m_nMax > cM2.m_nMax ? cM1.m_nMax : cM2.m_nMax;

    return MinMax(nMin, nMax);
}

MinMax operator+(const MinMax &cM, int nValue)
{
    // Get the minimum value seen in cM and nValue
    int nMin = cM.m_nMin < nValue ? cM.m_nMin : nValue;

    // Get the maximum value seen in cM and nValue
    int nMax = cM.m_nMax > nValue ? cM.m_nMax : nValue;
```

```
34
35      return MinMax(nMin, nMax);
36  }
37
38  MinMax operator+(int nValue, const MinMax &cM)
39  {
40      // call operator+(MinMax, nValue)
41      return (cM + nValue);
42  }
43
44  int main()
45  {
46      MinMax cM1(10, 15);
47      MinMax cM2(8, 11);
48      MinMax cM3(3, 12);
49
50      MinMax cMFinal = cM1 + cM2 + 5 + 8 + cM3 + 16;
51
52      std::cout << "Result: (" << cMFinal.GetMin() << ", " <<
53          cMFinal.GetMax() << ")" << std::endl;

        return 0;
    }
```

The MinMax class keeps track of the minimum and maximum values that it has seen so far. We have overloaded the + operator 3 times, so that we can add two MinMax objects together, or add integers to MinMax objects.

This example produces the result:

```
Result: (3, 16)
```

which you will note is the minimum and maximum values that we added to cMFinal.

One other interesting thing to note is that we defined operator+(int, MinMax) by calling operator+(MinMax, int). This is slightly less efficient than implementing it directly (due to the extra function call), but keeps our code shorter and easier to maintain (because it reduces duplicate code). It is often possible to define overloaded operators by calling other overloaded operators — when possible, do so!

9.3 — Overloading the I/O operators

Overloading

For classes that have multiple member variables, printing each of the individual variables on the screen can get tiresome fast. For example, consider the following class:

```
1  class Point
2  {
3  private:
4      double m_dX, m_dY, m_dZ;
5
6  public:
7      Point(double dX=0.0, double dY=0.0, double dZ=0.0)
8      {
9      m_dX = dX;
10     m_dY = dY;
11     m_dZ = dZ;
12     }
13
14     double GetX() { return m_dX; }
15     double GetY() { return m_dY; }
16     double GetZ() { return m_dZ; }
17 };
```

If you wanted to print an instance of this class to the screen, you'd have to do something like this:

```
1  Point cPoint(5.0, 6.0, 7.0);
2  cout << "(" << cPoint.GetX() << ", " <<
3      cPoint.GetY() << ", " <<
4      cPoint.GetZ() << ")";
```

And that's just for one instance! It would be much easier if you could simply type:

```
1  Point cPoint(5.0, 6.0, 7.0);
2  cout << cPoint;
```

and get the same result. By overloading the << operator, you can! Overloading operator<< is similar to overloading operator+ (they are both binary operators), except that the parameter types are different.

Consider the expression `cout << cPoint`. If the operator is <<, what are the operands? The left operand is the cout object, and the right operand is your Point class object. cout is actually an object of type ostream. Therefore, our overloaded function will look like this:

```
1  friend ostream& operator<< (ostream &out, Point &cPoint);
```

Implementation of operator<< is fairly straightforward -- because C++ already knows how to output doubles using operator<<, and our members are all doubles, we can simply use operator<< to output the member variables of our Point. Here is the above Point class with the overloaded operator<<.

```
1  class Point
2  {
3  private:
4      double m_dX, m_dY, m_dZ;
5
6  public:
7      Point(double dX=0.0, double dY=0.0, double dZ=0.0)
8      {
9      m_dX = dX;
10     m_dY = dY;
11     m_dZ = dZ;
12     }
13
14     friend ostream& operator<< (ostream &out, Point &cPoint);
15
16     double GetX() { return m_dX; }
17     double GetY() { return m_dY; }
18     double GetZ() { return m_dZ; }
19 };
20
21 ostream& operator<< (ostream &out, Point &cPoint)
22 {
23     // Since operator<< is a friend of the Point class, we can access
24     // Point's members directly.
25     out << "(" << cPoint.m_dX << ", " <<
26         cPoint.m_dY << ", " <<
27         cPoint.m_dZ << ")";
28     return out;
29 }
```

This is pretty straightforward -- note how similar our output line is to the line we wrote when we were outputting our members manually. They are almost identical, except cout has become parameter out!

The only tricky part here is the return type. Why are we returning an object of type ostream? The answer is that we do this so we can "chain" output commands together, such as `cout << cPoint << endl;`

Consider what would happen if our operator<< returned void. When the compiler evaluates `cout << cPoint << endl;`, due to the precedence/associativity rules, it evaluates this expression as `(cout << cPoint) << endl;`. `cout << cPoint` calls our void-returning overloaded operator<< function, which returns void. Then the partially evaluated expression becomes: `void << endl;`, which makes no sense!

By returning the out parameter as the return type instead, `(cout << cPoint)` returns cout. Then our partially evaluated expression becomes: `cout << endl;`, which then gets evaluated itself!

Any time we want our overloaded binary operators to be chainable in such a manner, the left operand should be returned.

Just to prove it works, consider the following example, which uses the Point class with the overloaded operator<< we wrote above:

```
1  int main()
2  {
3      Point cPoint1(2.0, 3.0, 4.0);
4      Point cPoint2(6.0, 7.0, 8.0);
5
6      using namespace std;
7      cout << cPoint1 << " " << cPoint2 << endl;
8
       return 0;
   }
```

This produces the following result:

```
(2.0, 3.0, 4.0) (6.0, 7.0, 8.0)
```

Overloading >>

It is also possible to overload the input operator. This is done in a manner very analogous to overloading the output operator. The key thing you need to know is that cin is an object of type istream. Here's our Point class with an overloaded operator>>:

```
1  class Point
2  {
3  private:
4      double m_dX, m_dY, m_dZ;
5
6  public:
7      Point(double dX=0.0, double dY=0.0, double dZ=0.0)
8      {
9      m_dX = dX;
10     m_dY = dY;
11     m_dZ = dZ;
       }
12
13     friend ostream& operator<< (ostream &out, Point &cPoint);
14     friend istream& operator>> (istream &in, Point &cPoint);
15
16     double GetX() { return m_dX; }
17     double GetY() { return m_dY; }
18     double GetZ() { return m_dZ; }
   };
```

```
ostream& operator<< (ostream &out, Point &cPoint)
{
    // Since operator<< is a friend of the Point class, we can access
    // Point's members directly.
    out << "(" << cPoint.m_dX << ", " <<
        cPoint.m_dY << ", " <<
        cPoint.m_dZ << ")";
    return out;
}

istream& operator>> (istream &in, Point &cPoint)
{
    in >> cPoint.m_dX;
    in >> cPoint.m_dY;
    in >> cPoint.m_dZ;
    return in;
}
```

Here's a sample program using both the overloaded operator<< and operator>>:

```
int main()
{
    using namespace std;
    cout << "Enter a point: " << endl;

    Point cPoint;
    cin >> cPoint;

    cout << "You entered: " << cPoint << endl;

    return 0;
}
```

Assuming the user enters `3.0 4.5 7.26` as input, the program produces the following result:

```
You entered: (3, 4.5, 7.26)
```

Conclusion

Overloading operator<< and operator>> make it extremely easy to output your class to screen and accept user input.

Before we finish this lesson, there is one additional point that is important to make. The overloaded output operator<<

```
friend ostream& operator<< (ostream &out, Point &cPoint);
```

is actually better written as

```
friend ostream& operator<< (ostream &out, const Point &cPoint);
```

This way, you will be able to output both const and non-const objects.

However, for the overloaded input operator>>, you will have to leave cPoint as non-const because the overloaded operator>> modifies cPoints members.

By now, you should be starting to become comfortable with classes and passing parameters by reference. In future lessons, we will start making more of our parameters const references (which we should have been doing all along, but have abstained for purposes of simplicity). It is a good habit to get into early.

9.4 — Overloading the comparison operators

Overloading the comparison operators is simple once you've learned how to overload the arithmetic operators.

Because the comparison operators are all binary operators that do not modify their operands, we will make our overloaded comparison operators friend functions.

Here's an example Point class from the previous lesson with an overloaded operator== and operator!=.

```
1  class Point
2  {
3  private:
4      double m_dX, m_dY, m_dZ;
5
6  public:
7      Point(double dX=0.0, double dY=0.0, double dZ=0.0)
8      {
9      m_dX = dX;
10     m_dY = dY;
11     m_dZ = dZ;
12     }
13
14     friend bool operator== (Point &cP1, Point &cP2);
15     friend bool operator!= (Point &cP1, Point &cP2);
16
17     double GetX() { return m_dX; }
18     double GetY() { return m_dY; }
19     double GetZ() { return m_dZ; }
20 };
21
22 bool operator== (Point &cP1, Point &cP2)
23 {
24     return (cP1.m_dX == cP2.m_dX &&
25             cP1.m_dY == cP2.m_dY &&
26             cP1.m_dZ == cP2.m_dZ);
27 }
28
29 bool operator!= (Point &cP1, Point &cP2)
30 {
31     return !(cP1 == cP2);
32 }
```

The code here should be straightforward. Because the result of operator!= is the opposite of operator==, we define operator!= in terms of operator==, which helps keep things simpler, more error free, and reduces the amount of code we have to write.

It doesn't really make sense to overload operator> or operator< for the Point class. What does it mean for a 3d point to be greater or less than another point? Greater than or less than isn't a concept we normally apply to 3d points, so it's better not to include those operators in the Point class, because the results of the operators (whatever you define them to be) would not be intuitive.

Here's a different example with an overloaded operator>, operator<, operator>=, and operator<=:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5
6  public:
7      Cents(int nCents) { m_nCents = nCents; }
8
9      friend bool operator> (Cents &cC1, Cents &cC2);
10     friend bool operator<= (Cents &cC1, Cents &cC2);
11
12     friend bool operator< (Cents &cC1, Cents &cC2);
13     friend bool operator>= (Cents &cC1, Cents &cC2);
14 };
15
16 bool operator> (Cents &cC1, Cents &cC2)
17 {
18     return cC1.m_nCents > cC2.m_nCents;
19 }
20
21 bool operator<= (Cents &cC1, Cents &cC2)
22 {
23     return cC1.m_nCents <= cC2.m_nCents;
24 }
25
26 bool operator< (Cents &cC1, Cents &cC2)
27 {
28     return cC1.m_nCents < cC2.m_nCents;
29 }
30
31 bool operator>= (Cents &cC1, Cents &cC2)
32 {
33     return cC1.m_nCents >= cC2.m_nCents;
34 }
```

This is also pretty straightforward.

Note that there is some redundancy here as well. operator> and operator<= are logical opposites, so one could be defined in terms of the other. operator< and operator>= are also logical opposites, and one could be defined in terms of the other. In this case, I chose not to do so because the function definitions are so simple, and the comparison operator in the function name line up nicely with the comparison operator in the return statement.

9.5 — Overloading unary operators +, -, and !

Overloading unary operators

Unlike the operators you've seen so far, the positive (+), negative (-) and logical not (!) operators all are unary operators, which means they only operate on one operand. Because none of these operators change their operands, we will be implementing them as friend functions. All three operands are implemented in an identical manner.

Let's take a look at how we'd implement operator- on the Cents class we used in a previous example:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5
6  public:
7      Cents(int nCents) { m_nCents = nCents; }
8
9      // Overload -cCents
10     friend Cents operator-(const Cents &cCents);
11 };
12
13 // note: this function is not a member function!
14 Cents operator-(const Cents &cCents)
   {
       return Cents(-cCents.m_nCents);
   }
```

This should be extremely straightforward. Our overloaded negative operator (-) takes one parameter of type Cents, and returns a value of type Cents.

Here's another example. The ! operator is often used as a shorthand method to test if something is set to the value zero. For example, the following example would only execute if nX were zero:

```
1 if (!nX)
2     // do something
```

Similarly, we can overload the ! operator to work similarly for a user-defined class:

```
1  class Point
2  {
3  private:
```

```
    double m_dX, m_dY, m_dZ;

public:
    Point(double dX=0.0, double dY=0.0, double dZ=0.0)
    {
    m_dX = dX;
    m_dY = dY;
    m_dZ = dZ;
    }

    // Convert a Point into it's negative equivalent
    friend Point operator- (const Point &cPoint);

    // Return true if the point is set at the origin
    friend bool operator! (const Point &cPoint);

    double GetX() { return m_dX; }
    double GetY() { return m_dY; }
    double GetZ() { return m_dZ; }
};

// Convert a Point into it's negative equivalent
Point operator- (const Point &cPoint)
{
    return Point(-cPoint.m_dX, -cPoint.m_dY, -cPoint.m_dZ);
}

// Return true if the point is set at the origin
bool operator! (const Point &cPoint)
{
    return (cPoint.m_dX == 0.0 &&
        cPoint.m_dY == 0.0 &&
        cPoint.m_dZ == 0.0);
}
```

In this case, if our point has coordinates (0.0, 0.0, 0.0), the logical not operator will return true. Otherwise, it will return false. Thus, we can say:

```
Point cPoint; // use default contructor to set to (0.0, 0.0, 0.0)

if (!cPoint)
    cout << "cPoint was set at the origin." << endl;
else
    cout << "cPoint was not set at the origin." << endl;

```

which produces the result:

```
cPoint was set at the origin.
```

9.6 — Overloading operators using member functions

In the lesson on overloading the arithmetic operators, you learned that when the operator does not modify it's operands, it's best to implement the overloaded operator as a friend function of the class. For operators that do modify their operands, we typically overload the operator using a member function of the class.

Overloading operators using a member function is very similar to overloading operators using a friend function. When overloading an operator using a member function:

- The leftmost operand of the overloaded operator must be an object of the class type.
- The leftmost operand becomes the implicit *this parameter. All other operands become function parameters.

Most operators can actually be overloaded either way, however there are a few exception cases:

- If the leftmost operand is not a member of the class type, such as when overloading operator+(int, YourClass), or operator<<(ostream&, YourClass), the operator must be overloaded as a friend.
- The assignment (=), subscript ([]), call (()), and member selection (->) operators must be overloaded as member functions.

Overloading the unary negative (-) operator

The negative operator is a unary operator that can be implemented using either method. Before we show you how to overload the operator using a member function, here's a reminder of how we overloaded it using a friend function:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5
6  public:
7      Cents(int nCents) { m_nCents = nCents; }
8
9      // Overload -cCents
10     friend Cents operator-(const Cents &cCents);
11 };
12
13 // note: this function is not a member function!
   Cents operator-(const Cents &cCents)
   {
       return Cents(-cCents.m_nCents);
```

```
14  }
```

Now let's overload the same operator using a member function instead:

```
1   class Cents
2   {
3   private:
4       int m_nCents;
5
6   public:
7       Cents(int nCents) { m_nCents = nCents; }
8
9       // Overload -cCents
10      Cents operator-();
11  };
12
13  // note: this function is a member function!
14  Cents Cents::operator-()
15  {
16      return Cents(-m_nCents);
17  }
```

You'll note that this method is pretty similar. However, the member function version of operator- doesn't take any parameters! Where did the parameter go? In the lesson on the hidden this pointer, you learned that a member function has an implicit *this pointer which always points to the class object the member function is working on. The parameter we had to list explicitly in the friend function version (which doesn't have a *this pointer) becomes the implicit *this parameter in the member function version.

Remember that when C++ sees the function prototype `Cents Cents::operator-();`, the compiler internally converts this to `Cents operator-(const Cents *this)`, which you will note is almost identical to our friend version `Cents operator-(const Cents &cCents)`!

Overloading the binary addition (+) operator

Let's take a look at an example of a binary operator overloaded both ways. First, overloading operator+ using the friend function:

```
1   class Cents
2   {
3   private:
4       int m_nCents;
5
6   public:
7       Cents(int nCents) { m_nCents = nCents; }
8
9       // Overload cCents + int
10      friend Cents operator+(Cents &cCents, int nCents);
11
12      int GetCents() { return m_nCents; }
13
```

```
};

// note: this function is not a member function!
Cents operator+(Cents &cCents, int nCents)
{
    return Cents(cCents.m_nCents + nCents);
}
```

Now, the same operator overloaded using the member function method:

```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload cCents + int
    Cents operator+(int nCents);

    int GetCents() { return m_nCents; }
};

// note: this function is a member function!
Cents Cents::operator+(int nCents)
{
    return Cents(m_nCents + nCents);
}
```

Our two-parameter friend function becomes a one-parameter member function, because the leftmost parameter (cCents) becomes the implicit *this parameter in the member function version.

Most programmers find the friend function version easier to read than the member function version, because the parameters are listed explicitly. Furthermore, the friend function version can be used to overload some things the member function version can not. For example, `friend operator+(int, cCents)` can not be converted into a member function because the leftmost parameter is not a class object.

However, when dealing with operands that modify the class itself (eg. operators =, +=, -=, ++, --, etc…) the member function method is typically used because C++ programmers are used to writing member functions (such as access functions) to modify private member variables. Writing friend functions that modify private member variables of a class is generally not considered good coding style, as it violates encapsulation.

Furthermore, as mentioned, some specific operators must be implemented as member functions. We'll be covering most of these in upcoming lessons.

9.7 — Overloading the increment and decrement operators

Overloading the increment (`++`) and decrement (`--`) operators are pretty straightforward, with one small exception. There are actually two versions of the increment and decrement operators: a prefix increment and decrement (eg. `++nX; --nY;`) and a postfix increment and decrement (eg. `nX++; nY--;`).

Because the increment and decrement operators modify their operands, they're best overloaded as member functions. We'll tackle the prefix versions first because they're the most straightforward.

Overloading prefix increment and decrement

Prefix increment and decrement is overloaded exactly the same as any normal unary operator. We'll do this one by example:

```
1  class Digit
2  {
3  private:
4      int m_nDigit;
5  public:
6      Digit(int nDigit=0)
7      {
8          m_nDigit = nDigit;
9      }
10
11     Digit& operator++();
12     Digit& operator--();
13
14     int GetDigit() const { return m_nDigit; }
15 };
16
17 Digit& Digit::operator++()
18 {
19     // If our number is already at 9, wrap around to 0
20     if (m_nDigit == 9)
21         m_nDigit = 0;
22     // otherwise just increment to next number
23     else
24         ++m_nDigit;
25
26     return *this;
27 }
28
29 Digit& Digit::operator--()
30 {
```

```
    // If our number is already at 0, wrap around to 9
    if (m_nDigit == 0)
        m_nDigit = 9;
    // otherwise just decrement to next number
    else
        --m_nDigit;

    return *this;
}
```

Our Digit class holds a number between 0 and 9. We've overloaded increment and decrement so they increment/decrement the digit, wrapping around if the digit is incremented/decremented out range.

Note that we return *this. The overloaded increment and decrement operators return a Digit so multiple operators can be "chained" together. Consequently, we need to return an item of type Digit. Since these operators are implemented as member functions, we can just return *this, which is an item of type Digit!

Overloading postfix increment and decrement

Normally, functions can be overloaded when they have the same name but a different number and/or different type of parameters. However, consider the case of the prefix and postfix increment and decrement operators. Both have the same name (eg. operator++), are unary, and take one parameter of the same type. So how it is possible to differentiate the two when overloading?

The answer is that C++ uses a "dummy variable" or "dummy argument" for the postfix operators. This argument is a fake integer parameter that only serves to distinguish the postfix version of increment/decrement from the prefix version. Here is the above Digit class with both prefix and postfix overloads:

```
class Digit
{
private:
    int m_nDigit;
public:
    Digit(int nDigit=0)
    {
        m_nDigit = nDigit;
    }

    Digit& operator++(); // prefix
    Digit& operator--(); // prefix

    Digit operator++(int); // postfix
    Digit operator--(int); // postfix

    int GetDigit() const { return m_nDigit; }
};
```

```
16
17  Digit& Digit::operator++()
18  {
19      // If our number is already at 9, wrap around to 0
20      if (m_nDigit == 9)
21          m_nDigit = 0;
22      // otherwise just increment to next number
23      else
24          ++m_nDigit;
25
26      return *this;
27  }
28
29  Digit& Digit::operator--()
30  {
31      // If our number is already at 0, wrap around to 9
32      if (m_nDigit == 0)
33          m_nDigit = 9;
34      // otherwise just decrement to next number
35      else
36          --m_nDigit;
37
38      return *this;
39  }
40
41  Digit Digit::operator++(int)
42  {
43      // Create a temporary variable with our current digit
44      Digit cResult(m_nDigit);
45
46      // Use prefix operator to increment this digit
47      ++(*this);              // apply operator
48
49      // return temporary result
50      return cResult;       // return saved state
51  }
52
53  Digit Digit::operator--(int)
54  {
55      // Create a temporary variable with our current digit
56      Digit cResult(m_nDigit);
57
58      // Use prefix operator to increment this digit
59      --(*this);              // apply operator
60
61      // return temporary result
    return cResult;       // return saved state
}

int main()
{
    Digit cDigit(5);
    ++cDigit; // calls Digit::operator++();
    cDigit++; // calls Digit::operator++(int);
```

```
62
63      return 0;
    }
```

There are a few interesting things going on here. First, note that we've distinguished the prefix from the postfix operators by providing an integer dummy parameter on the postfix version. Second, because the dummy parameter is not used in the function implementation, we have not even given it a name. This tells the compiler to treat this variable as a placeholder, which means it won't warn us that we declared a variable but never used it.

Third, note that the prefix and postfix operators do the same job — they both increment or decrement the class. The difference between the two is in the value they return. The overloaded prefix operators return the class after it has been incremented or decremented. Consequently, overloading these is fairly straightforward. We simply increment or decrement our member variables, and then return *this.

The postfix operators, on the other hand, need to return the state of the class before it is incremented or decremented. This leads to a bit of a conundrum — if we increment or decrement the class, we won't be able to return the state of the class before it was incremented or decremented. On the other hand, if we return the state of the class before we increment or decrement it, the increment or decrement will never be called.

The typical way this problem is solved is to use a temporary variable that holds the value of the class before it is incremented or decremented. Then the class itself can be incremented or decremented. And finally, the temporary variable is returned to the caller. In this way, the caller receives a copy of the class before it was incremented or decremented, but the class itself is incremented or decremented. Note that this means the return value of the overloaded operator much be a non-reference, because we can't return a reference to a local variable that will be destroyed when the function exits. Also note that this means the postfix operators are typically less efficient than the prefix operators because of the added overhead of instantiating a temporary variable and returning by value instead of reference.

Finally, note that we've written the post-increment and post-decrement in such a way that it calls the pre-increment and pre-decrement to do most of the work. This cuts down on duplicate code, and makes our class easier to modify in the future.

9.8 — Overloading the subscript operator

When working with arrays, we typically use the subscript operator ([]) to index specific elements of an array:

```
1  anArray[0] = 7; // put the value 7 in the first element of the array
```

However, consider the following IntList class, which has a member variable that is an array:

```
1  class IntList
2  {
3  private:
4      int m_anList[10];
5  };
6
7  int main()
8  {
9      IntList cMyList;
10     return 0;
11 }
```

Because the m_anList member variable is private, we can not access it directly from cMyList. This means we have no way to directly get or set values in the m_anList array. So how do we get or put elements into our list?

Without operator overloading, the typical method would be to create access functions:

```
1  class IntList
2  {
3  private:
4      int m_anList[10];
5
6  public:
7      void SetItem(int nIndex, int nData) { m_anList[nIndex] = nData; }
8      int GetItem(int nIndex) { return m_anList[nIndex]; }
9  };
```

While this works, it's not particularly user friendly. Consider the following example:

```
1  int main()
2  {
3      IntList cMyList;
4      cMyList.SetItem(2, 3);
5
6      return 0;
7  }
```

Are we setting element 2 to the value 3, or element 3 to the value 2? Without seeing the definition of SetItem(), it's simply not clear.

A better solution in this case is to overload the subscript operator ([]) to allow access to the elements of m_anList. The subscript operator is one of the operators that must be overloaded as a member function. In this case, our overloaded subscript will take one parameter, an integer value that is the index of the element to access, and it will return an integer.

```
1  class IntList
2  {
3  private:
4      int m_anList[10];
5
6  public:
7      int& operator[] (const int nIndex);
8  };
9
10 int& IntList::operator[] (const int nIndex)
11 {
12     return m_anList[nIndex];
13 }
```

Now, whenever we use the subscript operator ([]) on an object of our class, the compiler will return the corresponding element from the m_anList member variable! This allows us to both get and set values of m_anList directly:

```
1  IntList cMyList;
2  cMyList[2] = 3; // set a value
3  cout << cMyList[2]; // get a value
4
5  return 0;
```

In this case, it's much more obvious that `cMyList[2] = 3` is setting element 2 to the value of 3!

Why operator[] returns a reference

Let's take a closer look at how `cMyList[2] = 3` evaluates. Because the subscript operator has a higher precedence than the assignment operator, `cMyList[2]` evaluates first. `cMyList[2]` calls operator[], which we've defined to return a reference to `cMyList.m_anList[2]`. Because operator[] is returning a reference, it returns the actual `cMyList.m_anList[2]` array element. Our partially evaluated expression becomes `cMyList.m_anList[2] = 3`, which is a straightforward integer assignment.

In the lesson a first look at variables, you learned that any value on the left hand side of an assignment statement must be an l-value (which is a variable that has an actual memory address). Because the result of operator[] can be used on the left hand side of an assignment (eg. `cMyList[2] = 3`), the return value of operator[] must be an l-value. As it turns out, references

are always l-values, because you can only take a reference of variables that have memory addresses. So by returning a reference, the compiler is satisfied that we are returning an l-value.

Consider what would happen if operator[] returned an integer by value instead of by reference. `cMyList[2]` would call operator[], which would return the *value of* cMyList.m_anList[2]. For example, if m_anList[2] had the value of 6, operator[] would return the value 6. `cMyList[2] = 3` would partially evaluate to `6 = 3`, which makes no sense! If you try to do this, the C++ compiler will complain:

```
C:\VCProjects\Test.cpp(386) : error C2106: '=' : left operand must be l-value
```

Rule: Values returned by reference or pointer can be l-values or r-values. Values returned by value can only be r-values.

Conclusion

The subscript operator is typically overloaded to provide access to 1-dimensional array elements contained within a class. Because strings are typically implemented as arrays of characters, operator[] is often implemented in string classes to allow the user to access a single character of the string.

One other advantage of overloading the subscript operator is that we can make it safer than accessing arrays directly. Normally, when accessing arrays, the subscript operator does not check whether the index is valid. For example, the compiler will not complain about the following code:

```
1  int anArray[5];
2  anArray[7] = 3; // index 7 is out of bounds!
```

However, if we know the size of our array, we can make our overloaded subscript operator check to ensure the index is within bounds:

```
1  #include <cassert> // for assert()
2
3  class IntList
4  {
5  private:
6      int m_anList[10];
7
8  public:
9      int& operator[] (const int nIndex);
10 };
11
12 int& IntList::operator[] (const int nIndex)
13 {
14     assert(nIndex >= 0 && nIndex < 10);

       return m_anList[nIndex];
   }
```

In the above example, we have used the assert() function (included in the cassert header) to make sure our index is valid. If the expression inside the assert evaluates to false (which means the user passed in an invalid index), the program will terminate with an error message, which is much better than the alternative (corrupting memory). This is probably the most common method of doing error checking of this sort.

9.9 — Overloading the parenthesis operator

All of the overloaded operators you have seen so far let you define the type of the operator's parameters, but the number of parameters is fixed based on the type of the operator. For example, the == operator always takes two parameters, whereas the logical NOT operator always takes one. The parenthesis operator (()) is a particularly interesting operator in that it allows you to vary both the type AND number of parameters it takes!

There are two things to keep in mind: first, the parenthesis operator must be implemented as a member function. Second, in non-class C++, the () operator is used to call functions or write subexpressions that evaluate with higher precedence. In the case of operator overloading, the () operator does neither — rather, it is just a normal operator that calls a function (named operator()) like any other overloaded operator.

Let's take a look at a common example that lends itself to overloading this operator:

```
1  class Matrix
2  {
3  private:
4      double adData[4][4];
5  public:
6      Matrix()
7      {
8          // Set all elements of the matrix to 0.0
9          for (int nCol=0; nCol<4; nCol++)
10             for (int nRow=0; nRow<4; nRow++)
                   adData[nRow][nCol] = 0.0;
       }
   };
```

Matrices are a key component of linear algebra, and are often used to do geometric modeling and 3d computer graphics work. In this case, all you need to recognize is that the Matrix class is a 4 by 4 two-dimensional array of doubles.

In the lesson on overloading the subscript operator, you learned that we could overload operator[] to provide direct access to a private one-dimensional array. However, in this case, we want access to a private two-dimensional array. Because operator[] only has one parameter, it is not sufficient to let us index a two-dimensional array.

However, because the () operator can take as many parameters as we want it to have, we can declare a version of operator() that takes two integers and use it to access our two-dimensional array. Here is an example of this:

```
1  #include <cassert> // for assert()
2  class Matrix
3  {
```

```
private:
    double adData[4][4];
public:
    Matrix()
    {
        // Set all elements of the matrix to 0.0
        for (int nCol=0; nCol<4; nCol++)
            for (int nRow=0; nRow<4; nRow++)
                adData[nRow][nCol] = 0.0;
    }

    double& operator()(const int nCol, const int nRow);
};

double& Matrix::operator()(const int nCol, const int nRow)
{
    assert(nCol >= 0 && nCol < 4);
    assert(nRow >= 0 && nRow < 4);

    return adData[nRow][nCol];
}
```

Now we can declare a Matrix and access it's elements like this:

```
Matrix cMatrix;
cMatrix(1, 2) = 4.5;
std::cout << cMatrix(1, 2);
```

which produces the result:

```
4.5
```

Now, let's overload the () operator again, this time in a way that takes no parameters at all:

```
#include <cassert> // for assert()
class Matrix
{
private:
    double adData[4][4];
public:
    Matrix()
    {
        // Set all elements of the matrix to 0.0
        for (int nCol=0; nCol<4; nCol++)
            for (int nRow=0; nRow<4; nRow++)
                adData[nRow][nCol] = 0.0;
    }

    double& operator()(const int nCol, const int nRow);
    void operator()();
};

double& Matrix::operator()(const int nCol, const int nRow)
```

```
{
    assert(nCol >= 0 && nCol < 4);
    assert(nRow >= 0 && nRow < 4);

    return adData[nRow][nCol];
}

void Matrix::operator()()
{
    // reset all elements of the matrix to 0.0
    for (int nCol=0; nCol<4; nCol++)
        for (int nRow=0; nRow<4; nRow++)
            adData[nRow][nCol] = 0.0;
}
```

And here's our new example:

```
Matrix cMatrix;
cMatrix(1, 2) = 4.5;
cMatrix(); // erase cMatrix
std::cout << cMatrix(1, 2);

```

which produces the result:

```
0
```

Because the () operator is so flexible, it can be tempting to use it for many different purposes. However, this is strongly discouraged, since the () symbol does not really give any indication of what the operator is doing. In our example above, it would be better to have written the erase functionality as a function called Clear() or Erase(), as `cMatrix.Erase()` is easier to understand than `cMatrix()` (which could do anything!).

Operator () is commonly overloaded with two parameters to index multidimensional arrays, or to retrieve a subset of a one dimensional array (returning all the elements from parameter 1 to parameter 2). Anything else is probably better written as a member function with a more descriptive name.

9.10 — Overloading typecasts

In the lesson on type conversion and casting, you learned that C++ allows you to convert one data type to another. The following example shows an int being converted into a double:

```
int nValue = 5;
double dValue = nValue; // int implicitly cast to a double
```

C++ already knows how to convert between the built-in data types. However, it does not know how to convert any of our user-defined classes. That's where overloading the typecast operators comes into play.

Overloading the typecast operators allows us to convert our class into another data type. Take a look at the following class:

```
class Cents
{
private:
    int m_nCents;
public:
    Cents(int nCents=0)
    {
        m_nCents = nCents;
    }

    int GetCents() { return m_nCents; }
    void SetCents(int nCents) { m_nCents = nCents; }
};
```

This class is pretty simple: it holds some number of cents as an integer, and provides access functions to get and set the number of cents. It also provides a constructor for converting an int into a Cents.

If we can convert an int into a Cents, then doesn't it also make sense for us to be able to convert a Cents back into an int? In the following example, we have to use GetCents() to convert our Cents variable back into an integer so we can print it using PrintInt():

```
void PrintInt(int nValue)
{
    cout << nValue;
}

int main()
{
    Cents cCents(7);
    PrintInt(cCents.GetCents()); // print 7
```

```
    return 0;
}
```

If we have already written a lot of functions that take integers as parameters, our code will be littered with calls to GetCents(), which makes it more messy than it needs to be.

To make things easier, we'll overload the int cast, which will allow us to cast our Cents class into an int. The following example shows how this is done:

```
class Cents
{
private:
    int m_nCents;
public:
    Cents(int nCents=0)
    {
        m_nCents = nCents;
    }

    // Overloaded int cast
    operator int() { return m_nCents; }

    int GetCents() { return m_nCents; }
    void SetCents(int nCents) { m_nCents = nCents; }
};
```

There are two things to note:
1) To overload the function that casts our class to an int, we write a new function in our class called operator int(). Note that there is a space between the word operator and the type we are casting to.
2) Casting operators do not have a return type. C++ assumes you will be returning the correct type.

Now in our example, we call PrintInt() like this:

```
int main()
{
    Cents cCents(7);
    PrintInt(cCents); // print 7

    return 0;
}
```

The compiler will first note that PrintInt takes an integer parameter. Then it will note that cCents is not an int. Finally, it will look to see if we've provided a way to convert a Cents into an int. Since we have, it will call our operator int() function, which returns an int, and the returned int will be passed to PrintInt().

We can now also explicitly cast our Cents variable to an int:

```
Cents cCents(7);
int nCents = static_cast<int>(cCents);
```

You can overload cast operators for any data type you wish, including your own user-defined data types!

Here's a new class called Dollars that provides an overloaded Cents cast operator:

```
class Dollars
{
private:
    int m_nDollars;
public:
    Dollars(int nDollars=0)
    {
        m_nDollars = nDollars;
    }

     // Allow us to convert Dollars into Cents
     operator Cents() { return Cents(m_nDollars * 100); }
};
```

This allows us to convert a Dollars object directly into a Cents object! This allows you to do something like this:

```
void PrintCents(Cents cCents)
{
    cout << cCents.GetCents();
}

int main()
{
    Dollars cDollars(9);
    PrintCents(cDollars); // cDollars will be cast to a Cents

    return 0;
}
```

Consequently, this program will print the value:

```
900
```

which makes sense, since 9 dollars is 900 cents!

9.11 — The copy constructor and overloading the assignment operator

Although using the assignment operator is fairly straightforward, correctly implementing an overloaded assignment operator can be a little more tricky than you might anticipate. There are two primary reasons for this. First, there are some cases where the assignment operator isn't called when you might expect it to be. Second, there are some issues in dealing with dynamically allocated memory (which we will cover in the next lesson).

The **assignment operator** is used to copy the values from one object to another *already existing object*. The key words here are "already existing". Consider the following example:

```
1 Cents cMark(5); // calls Cents constructor
2 Cents cNancy; // calls Cents default constructor
3 cNancy = cMark; // calls Cents assignment operator
```

In this case, cNancy has already been created by the time the assignment is executed. Consequently, the Cents assignment operator is called. The assignment operator must be overloaded as a member function.

What happens if the object being copied into does not already exist? To understand what happens in that case, we need to talk about the copy constructor.

The copy constructor

Consider the following example:

```
1 Cents cMark(5); // calls Cents constructor
2 Cents cNancy = cMark; // calls Cents copy constructor!
```

Because the second statement uses an equals symbol in it, you might expect that it calls the assignment operator. However, it doesn't! It actually calls a special type of constructor called a copy constructor. A **copy constructor** is a special constructor that initializes a *new object* from an existing object.

The purpose of the copy constructor and the assignment operator are almost equivalent — both copy one object to another. However, the assignment operator copies to existing objects, and the copy constructor copies to newly created objects.

The difference between the copy constructor and the assignment operator causes a lot of confusion for new programmers, but it's really not all that difficult. Summarizing:

- If a new object has to be created before the copying can occur, the copy constructor is used.
- If a new object does not have to be created before the copying can occur, the assignment operator is used.

There are three general cases where the copy constructor is called instead of the assignment operator:

1. When instantiating one object and initializing it with values from another object (as in the example above).
2. When passing an object by value.
3. When an object is returned from a function by value.

In each of these cases, a new variable needs to be created before the values can be copied — hence the use of the copy constructor.

Because the copy constructor and assignment operator essentially do the same job (they are just called in different cases), the code needed to implement them is almost identical.

An overloaded assignment operator and copy constructor example

Now that you understand the difference between the copy constructor and assignment operator, let's see how they are implemented. For simple classes such as our Cents class, it is very straightforward.

Here is a simplified version of our Cents class:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5  public:
6      Cents(int nCents=0)
7      {
8          m_nCents = nCents;
       }
   };
```

First, let's add the copy constructor. Thinking about this logically, because it is a constructor, it needs to be named Cents. Because it needs to copy an existing object, it needs to take a Cents object as a parameter. And finally, because it is a constructor, it doesn't have a return type. Putting all of these things together, here is our Cents class with a copy constructor.

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5  public:
       Cents(int nCents=0)
```

```
    {
        m_nCents = nCents;
    }

    // Copy constructor
    Cents(const Cents &cSource)
    {
        m_nCents = cSource.m_nCents;
    }
};
```

A copy constructor looks just like a normal constructor that takes a parameter of the class type. However, there are two things which are worth explicitly mentioning. First, because our copy constructor is a member of Cents, and our parameter is a Cents, we can directly access the internal private data of our parameter. Second, the parameter MUST be passed by reference, and not by value. Can you figure out why?

The answer lies above in the list that shows the cases where a copy constructor is called. A copy constructor is called when a parameter is passed by value. If we pass our cSource parameter by value, it would need to call the copy constructor to do so. But calling the copy constructor again would mean the parameter is passed by value again, requiring another call to the copy constructor. This would result in an infinite recursion (well, until the stack memory ran out and the the program crashed). Fortunately, modern C++ compilers will produce an error if you try to do this:

```
C:\\Test.cpp(431) : error C2652: 'Cents' : illegal copy constructor: first parameter must not be a 'Cents'
```

The first parameter in this case must be a reference to a Cents!

Now let's overload the assignment operator. Following the same logic, the prototype and implementation are fairly straightforward:

```
class Cents
{
private:
    int m_nCents;
public:
    Cents(int nCents=0)
    {
        m_nCents = nCents;
    }

    // Copy constructor
    Cents(const Cents &cSource)
    {
        m_nCents = cSource.m_nCents;
    }

    Cents& operator= (const Cents &cSource);

```

```
};

Cents& Cents::operator= (const Cents &cSource)
{
    // do the copy
    m_nCents = cSource.m_nCents;

    // return the existing object
    return *this;
}
```

A couple of things to note here: First, the line that does the copying is exactly identical to the one in the copy constructor. This is typical. In order to reduce duplicate code, the portion of the code that does the actual copying could be moved to a private member function that the copy constructor and overloaded assignment operator both call. Second, we're returning *this so we can chain multiple assigments together:

```
cMark = cNancy = cFred = cJoe; // assign cJoe to everyone
```

If you need a refresher on chaining, we cover that in the section on overloading the I/O operators.

Finally, note that it is possible in C++ to do a self-assignment:

```
cMark = cMark; // valid assignment
```

In these cases, the assignment operator doesn't need to do anything (and if the class uses dynamic memory, it can be dangerous if it does). It is a good idea to do a check for self-assignment at the top of an overloaded assignment operator. Here is an example of how to do that:

```
Cents& Cents::operator= (const Cents &cSource)
{
    // check for self-assignment by comparing the address of the
    // implicit object and the parameter
    if (this == &cSource)
        return *this;

    // do the copy
    m_nCents = cSource.m_nCents;

    // return the existing object
    return *this;
}
```

Note that there is no need to check for self-assignment in a copy-constructor. This is because the copy constructor is only called when new objects are being constructed, and there is no way to assign a newly created object to itself in a way that calls to copy constructor.

Default memberwise copying

Just like other constructors, C++ will provide a **default copy constructor** if you do not provide one yourself. However, unlike other operators, C++ will provide a **default assignment operator** if you do not provide one yourself!

Because C++ does not know much about your class, the default copy constructor and default assignment operators it provides are very simple. They use a copying method known as a memberwise copy (also known as a shallow copy). We will talk more about shallow and deep copying in the next lesson.

9.12 — Shallow vs. deep copying

In the previous lesson on The copy constructor and overloading the assignment operator, you learned about the differences and similarities of the copy constructor and the assignment operator. This lesson is a follow-up to that one.

Shallow copying

Because C++ does not know much about your class, the default copy constructor and default assignment operators it provides use a copying method known as a shallow copy (also known as a memberwise copy). A **shallow copy** means that C++ copies each member of the class individually using the assignment operator. When classes are simple (eg. do not contain any dynamically allocated memory), this works very well.

For example, let's take a look at our Cents class:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5  public:
6      Cents(int nCents=0)
7      {
8          m_nCents = nCents;
       }
};
```

When C++ does a shallow copy of this class, it will copy m_nCents using the standard integer assignment operator. Since this is exactly what we'd be doing anyway if we wrote our own copy constructor or overloaded assignment operator, there's really no reason to write our own version of these functions!

However, when designing classes that handle dynamically allocated memory, memberwise (shallow) copying can get us in a lot of trouble! This is because the standard pointer assignment operator just copies the address of the pointer — it does not allocate any memory or copy the contents being pointed to!

Let's take a look at an example of this:

```
1  class MyString
2  {
3  private:
4      char *m_pchString;
5      int m_nLength;
6
7  public:
```

```
7       MyString(char *pchString="")
8       {
9           // Find the length of the string
10          // Plus one character for a terminator
11          m_nLength = strlen(pchString) + 1;
12
13          // Allocate a buffer equal to this length
14          m_pchString= new char[m_nLength];
15
16          // Copy the parameter into our internal buffer
17          strncpy(m_pchString, pchString, m_nLength);
18
19          // Make sure the string is terminated
20          m_pchString[m_nLength-1] = '\\0';
21      }
22
23      ~MyString() // destructor
24      {
25          // We need to deallocate our buffer
26          delete[] m_pchString;
27
28          // Set m_pchString to null just in case
29          m_pchString = 0;
30      }

        char* GetString() { return m_pchString; }
        int GetLength() { return m_nLength; }
};
```

The above is a simple string class that allocates memory to hold a string that we pass in. Note that we have not defined a copy constructor or overloaded assignment operator. Consequently, C++ will provide a default copy constructor and default assignment operator that do a shallow copy.

Now, consider the following snippet of code:

```
1   MyString cHello("Hello, world!");
2
3   {
4       MyString cCopy = cHello; // use default copy constructor
5   } // cCopy goes out of scope here
6
7   std::cout << cHello.GetString() << std::endl; // this will crash
```

While this code looks harmless enough, it contains an insidious problem that will cause the program to crash! Can you spot it? Don't worry if you can't, it's rather subtle.

Let's break down this example line by line:

```
1   MyString cHello("Hello, world!");
```

This line is harmless enough. This calls the MyString constructor, which allocates some memory, sets cHello.m_pchString to point to it, and then copies the string "Hello, world!" into it.

```
1 MyString cCopy = cHello; // use default copy constructor
```

This line seems harmless enough as well, but it's actually the source of our problem! When this line is evaluated, C++ will use the default copy constructor (because we haven't provided our own), which does a shallow pointer copy on cHello.m_pchString. Because a shallow pointer copy just copies the address of the pointer, the address of cHello.m_pchString is copied into cCopy.m_pchString. As a result, cCopy.m_pchString and cHello.m_pchString are now both pointing to the same piece of memory!

```
1 } // cCopy goes out of scope here
```

When cCopy goes out of scope, the MyString destructor is called on cCopy. The destructor deletes the dynamically allocated memory that both cCopy.m_pchString and cHello.m_pchString are pointing to! Consequently, by deleting cCopy, we've also (inadvertently) affected cHello. Note that the destructor will set cCopy.m_pchString to 0, but cHello.m_pchString will be left pointing to the deleted (invalid) memory!

```
1 std::cout << cHello.GetString() << std::endl; // this will crash
```

Now you can see why this crashes. We deleted the string that cHello was pointing to, and now we are trying to print the value of memory that is no longer allocated.

The root of this problem is the shallow copy done by the copy constructor — doing a shallow copy on pointer values in a copy constructor or overloaded assignment operator is almost always asking for trouble.

Deep copying

The answer to this problem is to do a deep copy on any non-null pointers being copied. A **deep copy** duplicates the object or variable being pointed to so that the destination (the object being assigned to) receives it's own local copy. This way, the destination can do whatever it wants to it's local copy and the object that was copied from will not be affected. Doing deep copies requires that we write our own copy constructors and overloaded assignment operators.

Let's go ahead and show how this is done for our MyString class:

```
1 // Copy constructor
2 MyString::MyString(const MyString& cSource)
3 {
4     // because m_nLength is not a pointer, we can shallow copy it
5     m_nLength = cSource.m_nLength;
6
7     // m_pchString is a pointer, so we need to deep copy it if it is non-
  null
```

```
8       if (cSource.m_pchString)
9       {
10          // allocate memory for our copy
11          m_pchString = new char[m_nLength];
12
13          // Copy the string into our newly allocated memory
14          strncpy(m_pchString, cSource.m_pchString, m_nLength);
15      }
16      else
            m_pchString = 0;
    }
```

As you can see, this is quite a bit more involved than a simple shallow copy! First, we have to check to make sure cSource even has a string (line 8). If it does, then we allocate enough memory to hold a copy of that string (line 11). Finally, we have to manually copy the string using strncpy() (line 14).

Now let's do the overloaded assignment operator. The overloaded assignment operator is a tad bit trickier:

```
1   // Assignment operator
2   MyString& MyString::operator=(const MyString& cSource)
3   {
4       // check for self-assignment
5       if (this == &cSource)
6           return *this;
7
8       // first we need to deallocate any value that this string is holding!
9       delete[] m_pchString;
10
11      // because m_nLength is not a pointer, we can shallow copy it
12      m_nLength = cSource.m_nLength;
13
14      // now we need to deep copy m_pchString
15      if (cSource.m_pchString)
16      {
17          // allocate memory for our copy
18          m_pchString = new char[m_nLength];
19
20          // Copy the parameter the newly allocated memory
21          strncpy(m_pchString, cSource.m_pchString, m_nLength);
22      }
23      else
            m_pchString = 0;

        return *this;
    }
```

Note that our assignment operator is very similar to our copy constructor, but there are three major differences:

- We added a self-assignment check (line 5).

- We return *this so we can chain the assignment operator (line 26).
- We need to explicitly deallocate any value that the string is already holding (line 9).

When the overloaded assignment operator is called, the item being assigned to may already contain a previous value, which we need to make sure we clean up before we assign memory for new values. For non-dynamically allocated variables (which are a fixed size), we don't have to bother because the new value just overwrite the old one. However, for dynamically allocated variables, we need to explicitly deallocate any old memory before we allocate any new memory. If we don't, the code will not crash, but we will have a memory leak that will eat away our free memory every time we do an assignment!

Checking for self-assignment

In our overloaded assignment operators, the first thing we do is check for self assignment. There are two reasons for this. One is simple efficiency: if we don't need to make a copy, why make one? The second reason is because not checking for self-assignment when doing a deep copy will cause problems if the class uses dynamically allocated memory. Let's take a look at an example of this.

Consider the following overloaded assignment operator that does not do a self-assignment check:

```
1  // Problematic assignment operator
2  MyString& MyString::operator=(const MyString& cSource)
3  {
4      // Note: No check for self-assignment!
5
6      // first we need to deallocate any value that this string is holding!
7      delete[] m_pchString;
8
9      // because m_nLength is not a pointer, we can shallow copy it
10     m_nLength = cSource.m_nLength;
11
12     // now we need to deep copy m_pchString
13     if (cSource.m_pchString)
14     {
15         // allocate memory for our copy
16         m_pchString = new char[m_nLength];
17
18         // Copy the parameter the newly allocated memory
19         strncpy(m_pchString, cSource.m_pchString, m_nLength);
20     }
21     else
22         m_pchString = 0;
23
24     return *this;
25 }
```

What happens when we do the following?

```
1  cHello = cHello;
```

This statement will call our overloaded assignment operator. The *this* pointer will point to the address of cHello (because it's the left operand), and cSource will be a reference to cHello (because it's the right operand). Consequently, m_pchString is the same as cSource.m_pchString.

Now look at the first line of code that would be executed: `delete[] m_pchString;`.

This line is meant to deallocate any previously allocated memory in cHello so we can copy the new string from the source without a memory leak. However, in this case, when we delete m_pchString, we also delete cSource.m_pchString! We've now destroyed our source string, and have lost the information we wanted to copy in the first place. The rest of the code will allocate a new string, then copy the uninitialized garbage in that string to itself. As a final result, you will end up with a new string of the correct length that contain garbage characters.

The self-assignment check prevents this from happening.

Preventing copying

Sometimes we simply don't want our classes to be copied at all. The best way to do this is to add the prototypes for the copy constructor and overloaded operator= to the private section of your class.

```
1  class MyString
2  {
3  private:
4      char *m_pchString;
5      int m_nLength;
6
7      MyString(const MyString& cSource);
8      MyString& operator=(const MyString& cSource);
9  public:
       // Rest of code here
   };
```

In this case, C++ will not automatically create a default copy constructor and default assignment operator, because we've told the compiler we're defining our own functions. Furthermore, any code located outside the class will not be able to access these functions because they're private.

Summary

- The default copy constructor and default assignment operators do shallow copies, which is fine for classes that contain no dynamically allocated variables.
- Classes with dynamically allocated variables need to have a copy constructor and assignment operator that do a deep copy.
- The assignment operator is usually implemented using the same code as the copy constructor, but it checks for self-assignment, returns *this, and deallocates any previously allocated memory before deep copying.

- If you don’t want a class to be copyable, use a private copy constructor and assignment operator prototype in the class header.

Composition

10.1 — Constructor initialization lists

Up until now, we've been initializing our class member data in the constructor using the assignment operator. For example:

```
1  class Something
2  {
3  private:
4      int m_nValue;
5      double m_dValue;
6      int *m_pnValue;
7
8  public:
9      Something()
10     {
11         m_nValue = 0;
12         m_dValue = 0.0;
           m_pnValue = 0;
       }
   };
```

When the class's constructor is executed, m_nValue, m_dValue, and m_chValue are created. Then the body of the constructor is run, where the member data variables are assigned values. This is similar to the flow of the following code in non-object-oriented C++:

```
1  int nValue;
2  double dValue;
3  int *pnValue;
4
5  nValue = 0;
6  dValue = 0.0;
   pnValue = 0;
```

While this does not exhibit good style, it is valid within the syntax of the C++ language.

So far, the classes that we have written have only included non-const or pointer member variables. However, what happens when we want to use const or reference variables as member variables? As you have learned in previous lessons, const and reference variables must be initialized on the line they are declared. Consider the following example:

```
1  class Something
2  {
3  private:
4      const int m_nValue;
5  public:
6      Something()
7      {
           m_nValue = 5;
```

```
8       }
};
```

This produces code similar to the following:

```
1  const int nValue; // error, const vars must be assigned values immediately
2  nValue = 5;
```

Consequently, assigning const or reference member variables values in the body of the constructor is not sufficient.

Initialization lists

C++ provides another way of initializing member variables that allows us to initialize member variables when they are created rather than afterwards. This is done through use of an initialization list.

In the lesson on basic addressing and variable declaration, you learned that you could assign values to variables in two ways: explicitly and implicitly:

```
1  int nValue = 5; // explicit assignment
2  double dValue(4.7); // implicit assignment
```

Using an initialization list is very similar to doing implicit assignments.

Let's take a look at our top example again. Here's the code that does explicit assignments in the constructor body:

```
1  class Something
2  {
3  private:
4      int m_nValue;
5      double m_dValue;
6      int *m_pnValue;
7
8  public:
9      Something()
10     {
11         m_nValue = 0;
12         m_dValue = 0.0;
           m_pnValue = 0;
       }
};
```

Now let's write the same code using an initialization list:

```
1  class Something
2  {
3  private:
       int m_nValue;
```

```
    double m_dValue;
    int *m_pnValue;

public:
    Something() : m_nValue(0), m_dValue(0.0), m_pnValue(0)
    {
    }
};
```

The initialization list is inserted after the constructor parameters, begins with a colon (:), and then lists each variable to initialize along with the value for that variable separated by a comma. Note that we no longer need to do the explicit assignments in the constructor body, since the initialization list replaces that functionality. Also note that the initialization list does not end in a semicolon.

Here's an example of a class that has a const member variable:

```
class Something
{
private:
    const int m_nValue;
public:
    Something(): m_nValue(5)
    {
    }
};
```

We strongly encourage you to begin using this new syntax (even if you aren't using const or reference member variables) as initialization lists are required when doing composition and inheritance (subjects we will be covering shortly).

10.2 — Composition

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc… Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called **composition** (also known as object composition).

More specifically, composition is used for objects that have a *has-a* relationship to each other. A car *has-a* metal frame, *has-an* engine, and *has-a* transmission. A personal computer *has-a* CPU, a motherboard, and other components. You *have-a* head, a body, some limbs.

So far, all of the classes we have used in our examples have had member variables that are built-in data types (eg. int, double). While this is generally sufficient for designing and implementing small, simple classes, it quickly becomes burdensome for more complex classes, especially those built from many sub-parts. In order to facilitate the building of complex classes from simpler ones, C++ allows us to do object composition in a very simple way — by using classes as member variables in other classes.

Lets take a look at some examples of how this is done. If we were designing a personal computer class, we might do it like this (assuming we'd already written a CPU, Motherboard, and RAM class):

```
1  #include "CPU.h"
2  #include "Motherboard.h"
3  #include "RAM.h"
4
5  class PersonalComputer
6  {
7  private:
8     CPU m_cCPU;
9     Motherboard m_cMotherboard;
10    RAM m_cRAM;
11 };
```

Initializing class member variables

In the previous lesson on initializer lists, you learned that the preferred way to initialize class members is through initializer lists rather than assignment. So let's write a constructor for our PersonalComputer class that uses an initialization list to initialize the member variables. This constructor will take 3 parameters: a CPU speed, a motherboard model, and a RAM size, which it will then pass to the respective member variables when they are constructed.

```
1  PersonalComputer::PersonalComputer(int nCPUSpeed, char *strMotherboardModel,
```

```
int nRAMSize)
    :        m_cCPU(nCPUSpeed),        m_cMotherboard(strMotherboardModel),
m_cRAM(nRAMSize)
{
}
```

Now, when a PersonalComputer object is instantiated using this constructor, that PersonalComputer object will contain a CPU object initialized with nCPUSpeed, a Motherboard object initialized with strMotherboardModel, and a RAM object initialized with nRAMSize.

It is worth explicitly noting that composition implies ownership between the complex class and any subclasses. When the complex class is created, the subclasses are created. When the complex class is destroyed, the subclasses are similarly destroyed.

A full example

While the above example is useful in giving the general idea of how composition works, let's do a full example that you can compile yourself. Many games and simulations have creatures or objects that move around a board, map, or screen. The one thing that all of these creatures/objects have in common is that they all *have-a* location. In this example, we are going to create a creature class that uses a point class to hold the creature's location.

First, let's design the point class. Our creature is going to live in a 2d world, so our point class will have 2 dimensions, X and Y. We will assume the world is made up of discrete squares, so these dimensions will always be integers.

Point2D.h:

```
#ifndef POINT2D_H
#define POINT2D_H

#include <iostream>

class Point2D
{
private:
    int m_nX;
    int m_nY;

public:
    // A default constructor
    Point2D()
        : m_nX(0), m_nY(0)
    {
    }

    // A specific constructor
    Point2D(int nX, int nY)
        : m_nX(nX), m_nY(nY)
    {
    }
```

```
20
21      // An overloaded output operator
22      friend std::ostream& operator<<(std::ostream& out, const Point2D
    &cPoint)
23      {
24          out << "(" << cPoint.GetX() << ", " << cPoint.GetY() << ")";
25          return out;
26      }
27
28      // Access functions
29      void SetPoint(int nX, int nY)
30      {
31          m_nX = nX;
32          m_nY = nY;
33      }
34
35      int GetX() const { return m_nX; }
36      int GetY() const { return m_nY; }
37  };

    #endif
```

Note that because we've implemented all of our functions in the header file (for the sake of keeping the example concise), there is no Point2D.cpp.

Now let's design our Creature. Our Creature is going to have a few properties. It's going to have a name, which will be a string, and a location, which will be our Point2D class.

Creature.h:

```
1   #ifndef CREATURE_H
2   #define CREATURE_H
3
4   #include <iostream>
5   #include <string>
6   #include "Point2D.h"
7
8   class Creature
9   {
10  private:
11      std::string m_strName;
12      Point2D m_cLocation;
13
14      // We don't want people to create Creatures with no name or location
15      // so our default constructor is private
16      Creature() { }
17
18  public:
19      Creature(std::string strName, const Point2D &cLocation)
20          : m_strName(strName), m_cLocation(cLocation)
21      {
22      }
```

```
20
21      friend std::ostream& operator<<(std::ostream& out, const Creature
22  &cCreature)
23      {
24          out << cCreature.m_strName.c_str() << " is at " <<
25  cCreature.m_cLocation;
26          return out;
27      }
28
29      void MoveTo(int nX, int nY)
30      {
31          m_cLocation.SetPoint(nX, nY);
32      }
33  };
34  #endif
```

And finally, Main.cpp:

```
1   #include <string>
2   #include <iostream>
3   #include "Creature.h"
4
5   int main()
6   {
7       using namespace std;
8       cout << "Enter a name for your creature: ";
9       std::string cName;
10      cin >> cName;
11      Creature cCreature(cName, Point2D(4, 7));
12
13      while (1)
14      {
15          cout << cCreature << endl;
16          cout << "Enter new X location for creature (-1 to quit): ";
17          int nX=0;
18          cin >> nX;
19          if (nX == -1)
20              break;
21
22          cout << "Enter new Y location for creature (-1 to quit): ";
23          int nY=0;
24          cin >> nY;
25          if (nY == -1)
26              break;
27
28          cCreature.MoveTo(nX, nY);
29          }
30
31      return 0;
32  }
```

Here's a transcript of this code being run:

```
Enter a name for your creature: Marvin
Marvin is at (4, 7)
Enter new X location for creature (-1 to quit): 6
Enter new Y location for creature (-1 to quit): 12
Marvin is at (6, 12)
Enter new X location for creature (-1 to quit): 3
Enter new Y location for creature (-1 to quit): 2
Marvin is at (3, 2)
Enter new X location for creature (-1 to quit): -1
```

Why use composition?

Instead of using the Point2D class to implement the Creature's location, we could have instead just added 2 integers to the Creature class and written code in the Creature class to handle the positioning. However, using composition provides a number of useful benefits:

1. Each individual class can be kept relatively simple and straightforward, focused on performing one task. This makes those classes easier to write and much easier to understand. For example, Point2D only worries about point-related stuff, which helps keep it simple.
2. Each subobject can be self-contained, which makes them reusable. For example, we could reuse our Point2D class in a completely different application. Or if our creature ever needed another point (for example, a destination it was trying to get to), we can simply add another Point2D member variable.
3. The complex class can have the simple subclasses do most of the hard work, and instead focus on coordinating the data flow between the subclasses. This helps lower the overall complexity of the complex object, because it can delegate tasks to the sub-objects, who already know how to do them. For example, when we move our Creature, it delegates that task to the Point class, which already understands how to set a point. Thus, the Creature class does not have to worry about how such things would be implemented.

One question that new programmers often ask is "When should I use composition instead of direct implementation of a feature?". There's no 100% answer to that question. However, a good rule of thumb is that each class should be built to accomplish a single task. That task should either be the storage and manipulation of some kind of data (eg. Point2D), OR the coordination of subclasses (eg. Creature). Not both.

In this case of our example, it makes sense that Creature shouldn't have to worry about how Points are implemented, or how the name is being stored. Creature's job isn't to know those intimate details. Creature's job is to worry about how to coordinate the data flow and ensure that each of the subclasses knows *what* it is supposed to do. It's up to the individual subclasses to worry about *how* they will do it.

10.3 — Aggregation

In the previous lesson on composition, you learned that compositions are complex classes that contain other subclasses as member variables. In addition, in a composition, the complex object "owns" all of the subobjects it is composed of. When a composition is destroyed, all of the subobjects are destroyed as well. For example, if you destroy a car, it's frame, engine, and other parts should be destroyed as well. If you destroy a PC, you would expect it's RAM and CPU to be destroyed as well.

Aggregration

An **aggregation** is a specific type of composition where no ownership between the complex object and the subobjects is implied. When an aggregate is destroyed, the subobjects are not destroyed.

For example, consider the math department of a school, which is made up of one or more teachers. Because the department does not own the teachers (they merely work there), the department should be an aggregate. When the department is destroyed, the teachers should still exist independently (they can go get jobs in other departments).

Because aggregations are just a special type of compositions, they are implemented almost identically, and the difference between them is mostly semantic. In a composition, we typically add our subclasses to the composition using either normal variables or pointers where the allocation and deallocation process is handled by the composition class.

In an aggregation, we also add other subclasses to our complex aggregate class as member variables. However, these member variables are typically either references or pointers that are used to point at objects that have been created outside the scope of the class. Consequently, an aggregate class usually either takes the objects it is going to point to as constructor parameters, or it begins empty and the subobjects are added later via access functions or operators.

Because these subclass objects live outside of the scope of the class, when the class is destroyed, the pointer or reference member variable will be destroyed, but the subclass objects themselves will still exist.

Let's take a look at our Teacher and Department example in more detail.

```
1  #include <string>
2  using namespace std;
3
4  class Teacher
5  {
6  private:
       string m_strName;
```

```
public:
    Teacher(string strName)
        : m_strName(strName)
    {
    }

    string GetName() { return m_strName; }
};

class Department
{
private:
    Teacher *m_pcTeacher; // This dept holds only one teacher

public:
    Department(Teacher *pcTeacher=NULL)
        : m_pcTeacher(pcTeacher)
    {
    }
};

int main()
{
    // Create a teacher outside the scope of the Department
    Teacher *pTeacher = new Teacher("Bob"); // create a teacher
    {
        // Create a department and use the constructor parameter to pass
        // the teacher to it.
        Department cDept(pTeacher);

    } // cDept goes out of scope here and is destroyed

    // pTeacher still exists here because cDept did not destroy it
    delete pTeacher;
}
```

In this case, pTeacher is created independetly of cDept, and then passed into cDept's constructor. Note that the department class uses an initialization list to set the value of m_pcTeacher to the pTeacher value we passed in. When cDept is destroyed, the m_pcTeacher pointer destroyed, but pTeacher is not deallocated, so it still exists until it is independently destroyed.

To summarize the differences between composition and aggregation:

Compositions:

- Typically use normal member variables
- Can use pointer values if the composition class automatically handles allocation/deallocation
- Responsible for creation/destruction of subclasses

Aggregations:

- Typically use pointer variables that point to an object that lives outside the scope of the aggregate class
- Can use reference values that point to an object that lives outside the scope of the aggregate class
- Not responsible for creating/destroying subclasses

It is worth noting that the concepts of composition and aggregation are not mutually exclusive, and can be mixed freely within the same class. It is entirely possible to write a class that is responsible for the creation/destruction of some subclasses but not others. For example, our Department class could have a name and a teacher. The name would probably be added to the department by composition, and would be created and destroyed with the department. On the other hand, the teacher would be added to the department by aggregate, and created/destroyed independently.

It is also possible to create other hybrid aggregate/composition schemes, such as where a class holds independent subobjects like an aggregate, but will destroy them when the class goes out of scope like a composition.

While aggregates can be extremely useful (which we will see more of in the next lesson on container classes), they are also potentially dangerous. As noted several times, aggregates are not responsible for deallocating their subobjects when they are destroyed. Consequently, if there are no other pointers or references to those subobjects when the aggregate is destroyed, those subobjects will cause a memory leak. It is up to the programmer to ensure that this does not happen. This is generally handled by ensuring other pointers or references to those subobjects exist when the aggregate is destroyed.

10.4 — Container classes

In real life, we use containers all the time. Your breakfast cereal comes in a box, the pages in your book come inside a cover and binding, and you might store any number of items in containers in your garage. Without containers, it would be extremely inconvenient to work with many of these objects. Imagine trying to read a book that didn't have any sort of binding, or eat cereal that didn't come in a box without using a bowl. It would be a mess. The value the container provides is largely in it's ability to help organize and store items that are put inside it.

Similarly, a **container class** is a class designed to hold and organize multiple instances of another class. There are many different kinds of container classes, each of which has various advantages, disadvantages, and restrictions in their use. By far the most commonly used container in programming is the array, which you have already seen many examples of. Although C++ has built-in array functionality, programmers will often use an array container class instead because of the additional benefits it provides. Unlike built-in arrays, array container classes generally provide dynamically resizing (when elements are added or removed) and do bounds-checking. This not only makes array container classes more convenient than normal arrays, but safer too.

Container classes typically implement a fairly standardized minimal set of functionality. Most well-defined containers will include functions that:

- Create an empty container (via a constructor)
- Insert a new object into the container
- Remove an object from the container
- Report the number of objects currently in the container
- Empty the container of all objects
- Provide access to the stored objects
- Sort the elements (optional)

Sometimes certain container classes will omit some of this functionality. For example, arrays container classes often omit the insert and delete functions because they are slow and the class designer does not want to encourage their use.

Container classes generally come in two different varieties. **Value containers** are compositions that store copies of the objects that they are holding (and thus are responsible for creating and destroying those copies). **Reference containers** are aggregations that store pointers or references to other objects (and thus are not responsible for creation or destruction of those objects).

Unlike in real life, where containers can hold whatever you put in them, in C++, containers typically only hold one type of data. For example, if you have an array of integers, it will only hold integers. Unlike some other languages, C++ generally does not allow you to mix types inside a container. If you want one container class that holds integers and another that holds

doubles, you will have to write two separate containers to do this (or use templates, which is an advanced C++ feature). Despite the restrictions on their use, containers are immensely useful, and they make programming easier, safer, and faster.

An array container class

In this example, we are going to write an integer array class that implements most of the common functionality that containers should have. This array class is going to be a value container, which will hold copies of the elements its organizing.

First, let's create the IntArray.h file:

```
1  #ifndef INTARRAY_H
2  #define INTARRAY_H
3
4  class IntArray
5  {
6  };
7
   #endif
```

Our IntArray is going to need to keep track of two values: the data itself, and the size of the array. Because we want our array to be able to change in size, we'll have to do some dynamic allocation, which means we'll have to use a pointer to store the data.

```
1  #ifndef INTARRAY_H
2  #define INTARRAY_H
3
4  class IntArray
5  {
6  private:
7      int m_nLength;
8      int *m_pnData;
9  };

   #endif
```

Now we need to add some constructors that will allow us to create IntArrays. We are going to add two constructors: one that constructs an empty array, and one that will allow us to construct an array of a predetermined size.

```
1  #ifndef INTARRAY_H
2  #define INTARRAY_H
3
4  class IntArray
5  {
6  private:
7      int m_nLength;
8      int *m_pnData;
```

```
public:
    IntArray()
    {
        m_nLength = 0;
        m_pnData = 0;
    }

    IntArray(int nLength)
    {
        m_pnData = new int[nLength];
        m_nLength = nLength;
    }
};

#endif
```

We'll also need some functions to help us clean up IntArrays. First, we'll write a destructor, which simply deallocates any dynamically allocated data. Second, we'll write a function called Erase(), which will erase the array and set the length to 0.

```
~IntArray()
{
    delete[] m_pnData;
}

void Erase()
{
    delete[] m_pnData;
    // We need to make sure we set m_pnData to 0 here, otherwise it will
    // be left pointing at deallocated memory!
    m_pnData = 0;
    m_nLength = 0;
}
```

Now let's overload the [] operator so we can access the elements of the array. We should bounds check the index to make sure it's valid, which is best done using the assert() function. We'll also add an access function to return the length of the array.

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <assert.h> // for assert()

class IntArray
{
private:
    int m_nLength;
    int *m_pnData;

public:
    IntArray()
    {
        m_nLength = 0;
```

```
14          m_pnData = 0;
15      }
16
17      IntArray(int nLength)
18      {
19          m_pnData = new int[nLength];
20          m_nLength = nLength;
21      }
22
23      ~IntArray()
24      {
25          delete[] m_pnData;
26      }
27
28      void Erase()
29      {
30          delete[] m_pnData;
31          // We need to make sure we set m_pnData to 0 here, otherwise it
32  will
33          // be left pointing at deallocated memory!
34          m_pnData = 0;
35          m_nLength = 0;
36      }
37
38      int& operator[](int nIndex)
39      {
40          assert(nIndex >= 0 && nIndex < m_nLength);
41          return m_pnData[nIndex];
42      }
43
44      int GetLength() { return m_nLength; }
45  };
46
47  #endif
```

At this point, we already have an IntArray class that we can use. We can allocate IntArrays of a given size, and we can use the [] operator to retrieve or change the value of the elements.

However, there are still a few thing we can't do with our IntArray. We still can't change it's size, still can't insert or delete elements, and we still can't sort it.

First, let's write some code that will allow us to resize an array. We are going to write two different functions to do this. The first function, Reallocate(), will destroy any existing elements in the array when it is resized, but it will be fast. The second function, Resize(), will keep any existing elements in the array when it is resized, but it will be slow.

```
1   // Reallocate resizes the array.  Any existing elements will be destroyed.
2   // This function operates quickly.
3   void Reallocate(int nNewLength)
4   {
5       // First we delete any existing elements
6       Erase();
```

```
6
7       // If our array is going to be empty now, return here
8       if (nNewLength<= 0)
9           return;
10
11      // Then we have to allocate new elements
12      m_pnData = new int[nNewLength];
13      m_nLength = nNewLength;
14  }
15
16  // Resize resizes the array.  Any existing elements will be kept.
17  // This function operates slowly.
18  void Resize(int nNewLength)
19  {
20      // If we are resizing to an empty array, do that and return
21      if (nNewLength <= 0)
22      {
23          Erase();
24          return;
25      }
26
27      // Now we can assume nNewLength is at least 1 element.  This algorithm
28      // works as follows: First we are going to allocate a new array.  Then
29  we
30      // are going to copy elements from the existing array to the new
31  array.
32      // Once that is done, we can destroy the old array, and make m_pnData
33      // point to the new array.
34
35      // First we have to allocate a new array
36      int *pnData = new int[nNewLength];
37
38      // Then we have to figure out how many elements to copy from the
39  existing
40      // array to the new array.  We want to copy as many elements as there
41  are
42      // in the smaller of the two arrays.
43      if (m_nLength > 0)
44      {
45          int nElementsToCopy = (nNewLength > m_nLength) ? m_nLength :
46  nNewLength;
47
48          // Now copy the elements one by one
49          for (int nIndex=0; nIndex < nElementsToCopy; nIndex++)
50              pnData[nIndex] = m_pnData[nIndex];
51      }

    // Now we can delete the old array because we don't need it any more
    delete[] m_pnData;

    // And use the new array instead!  Note that this simply makes
m_pnData point
    // to the same address as the new array we dynamically
allocated.  Because
    // pnData was dynamically allocated, it won't be destroyed when it
```

```
52  goes out of scope.
53      m_pnData = pnData;
54      m_nLength = nNewLength;
    }
```

Whew! That was a little tricky!

Many array container classes would stop here. However, just in case you want to see how insert and delete functionality would be implemented we'll go ahead and write those too. Both of these algorithms are very similar to Resize().

```
1   void InsertBefore(int nValue, int nIndex)
2   {
3       // Sanity check our nIndex value
4       assert(nIndex >= 0 && nIndex <= m_nLength);
5
6       // First create a new array one element larger than the old array
7       int *pnData = new int[m_nLength+1];
8
9       // Copy all of the elements up to the index
10      for (int nBefore=0; nBefore < nIndex; nBefore++)
11          pnData[nBefore] = m_pnData[nBefore];
12
13      // Insert our new element into the new array
14      pnData[nIndex] = nValue;
15
16      // Copy all of the values after the inserted element
17      for (int nAfter=nIndex; nAfter < m_nLength; nAfter++)
18          pnData[nAfter+1] = m_pnData[nAfter];
19
20      // Finally, delete the old array, and use the new array instead
21      delete[] m_pnData;
22      m_pnData = pnData;
23      m_nLength += 1;
24  }
25
26  void Remove(int nIndex)
27  {
28      // Sanity check our nIndex value
29      assert(nIndex >= 0 && nIndex < m_nLength);
30
31      // First create a new array one element smaller than the old array
32      int *pnData = new int[m_nLength-1];
33
34      // Copy all of the elements up to the index
35      for (int nBefore=0; nBefore < nIndex; nBefore++)
36          pnData[nBefore] = m_pnData[nBefore];

        // Copy all of the values after the inserted element
        for (int nAfter=nIndex+1; nAfter < m_nLength; nAfter++)
            pnData[nAfter-1] = m_pnData[nAfter];

        // Finally, delete the old array, and use the new array instead
```

```
    delete[] m_pnData;
    m_pnData = pnData;
    m_nLength -= 1;
}

// A couple of additional functions just for convenience
void InsertAtBeginning(int nValue) { InsertBefore(nValue, 0); }
void InsertAtEnd(int nValue) { InsertBefore(nValue, m_nLength); }
```

Here is our IntArray container class in it's entirety:

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <assert.h> // for assert()

class IntArray
{
private:
    int m_nLength;
    int *m_pnData;

public:
    IntArray()
    {
        m_nLength = 0;
        m_pnData = 0;
    }

    IntArray(int nLength)
    {
        m_pnData = new int[nLength];
        m_nLength = nLength;
    }

    ~IntArray()
    {
        delete[] m_pnData;
    }

    void Erase()
    {
        delete[] m_pnData;
        // We need to make sure we set m_pnData to 0 here, otherwise it
will
        // be left pointing at deallocated memory!
        m_pnData = 0;
        m_nLength = 0;
    }

    int& operator[](int nIndex)
    {
        assert(nIndex >= 0 && nIndex < m_nLength);
        return m_pnData[nIndex];
```

```
37      }
38
39      // Reallocate resizes the array.  Any existing elements will be
destroyed.
40      // This function operates quickly.
41      void Reallocate(int nNewLength)
42      {
43          // First we delete any existing elements
44          Erase();
45
46          // If our array is going to be empty now, return here
47          if (nNewLength<= 0)
48              return;
49
50          // Then we have to allocate new elements
51          m_pnData = new int[nNewLength];
52          m_nLength = nNewLength;
53      }
54
55      // Resize resizes the array.  Any existing elements will be kept.
56      // This function operates slowly.
57      void Resize(int nNewLength)
58      {
59          // If we are resizing to an empty array, do that and return
60          if (nNewLength <= 0)
61          {
62              Erase();
63              return;
64          }
65
66          // Now we can assume nNewLength is at least 1 element.  This
algorithm
67          // works as follows: First we are going to allocate a new
array.  Then we
68          // are going to copy elements from the existing array to the new
array.
69          // Once that is done, we can destroy the old array, and make
m_pnData
70          // point to the new array.
71
72          // First we have to allocate a new array
73          int *pnData = new int[nNewLength];
74
75          // Then we have to figure out how many elements to copy from the
existing
76          // array to the new array.  We want to copy as many elements as
there are
77          // in the smaller of the two arrays.
78          if (m_nLength > 0)
79          {
80              int nElementsToCopy = (nNewLength > m_nLength) ? m_nLength :
nNewLength;
81
82              // Now copy the elements one by one
              for (int nIndex=0; nIndex < nElementsToCopy; nIndex++)
```

```
                pnData[nIndex] = m_pnData[nIndex];
        }

        // Now we can delete the old array because we don't need it any more
        delete[] m_pnData;

        // And use the new array instead!  Note that this simply makes m_pnData point
        //  to the same address as the new array we dynamically allocated.  Because
        // pnData was dynamically allocated, it won't be destroyed when it goes out of scope.
        m_pnData = pnData;
        m_nLength = nNewLength;
    }

    void InsertBefore(int nValue, int nIndex)
    {
        // Sanity check our nIndex value
        assert(nIndex >= 0 && nIndex <= m_nLength);

        // First create a new array one element larger than the old array
        int *pnData = new int[m_nLength+1];

        // Copy all of the elements up to the index
        for (int nBefore=0; nBefore < nIndex; nBefore++)
            pnData[nBefore] = m_pnData[nBefore];

        // insert our new element into the new array
        pnData[nIndex] = nValue;

        // Copy all of the values after the inserted element
        for (int nAfter=nIndex; nAfter < m_nLength; nAfter++)
            pnData[nAfter+1] = m_pnData[nAfter];

        // Finally, delete the old array, and use the new array instead
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength += 1;
    }

    void Remove(int nIndex)
    {
        // Sanity check our nIndex value
        assert(nIndex >= 0 && nIndex < m_nLength);

        // First create a new array one element smaller than the old array
        int *pnData = new int[m_nLength-1];

        // Copy all of the elements up to the index
        for (int nBefore=0; nBefore < nIndex; nBefore++)
            pnData[nBefore] = m_pnData[nBefore];
```

```
129
130         // Copy all of the values after the inserted element
131         for (int nAfter=nIndex+1; nAfter < m_nLength; nAfter++)
132             pnData[nAfter-1] = m_pnData[nAfter];
133
134         // Finally, delete the old array, and use the new array instead
135         delete[] m_pnData;
136         m_pnData = pnData;
137         m_nLength -= 1;
138     }
139
140     // A couple of additional functions just for convenience
141     void InsertAtBeginning(int nValue) { InsertBefore(nValue, 0); }
142     void InsertAtEnd(int nValue) { InsertBefore(nValue, m_nLength); }
143
144     int GetLength() { return m_nLength; }
};

#endif
```

Now, let's test it just to prove it works:

```
1  #include <iostream>
2  #include "IntArray.h"
3
4  using namespace std;
5
6  int main()
7  {
8      // Declare an array with 10 elements
9      IntArray cArray(10);
10
11     // Fill the array with numbers 1 through 10
12     for (int i=0; i<10; i++)
13         cArray[i] = i+1;
14
15     // Resize the array to 8 elements
16     cArray.Resize(8);
17
18     // Insert the number 20 before the 5th element
19     cArray.InsertBefore(20, 5);
20
21     // Remove the 3rd element
22     cArray.Remove(3);
23
24     // Add 30 and 40 to the end and beginning
25     cArray.InsertAtEnd(30);
26     cArray.InsertAtBeginning(40);
27
28     // Print out all the numbers
29     for (int j=0; j<cArray.GetLength(); j++)
30         cout << cArray[j] << " ";
31
```

```
    return 0;
}
```

This produces the result:

```
40 1 2 3 5 20 6 7 8 30
```

Although writing container classes can be pretty complex, the good news is that you only have to write them once. Once the container class is working, you can use and reuse it as often as you like without any additional programming effort required.

It is also worth explicitly mentioning that even though our sample IntArray container class holds a built-in data type (int), we could have just as easily used a user-defined type (eg. a point class).

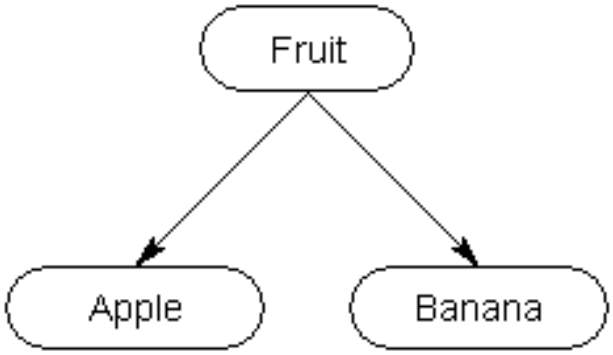
Inheritance

11.1 — Introduction to inheritance

In the lesson on composition, you learned how to construct complex classes by combining simpler classes. Composition is perfect for building new objects that have a *has-a* relationship with their subobjects. However, composition (and aggregation) is just one of the two major ways that C++ lets you construct complex classes. The second way is through inheritance.

Unlike composition, which involves creating new objects by combining and connecting other objects, **inheritance** involves creating new objects by directly acquiring the attributes and behaviors of other objects and then extending or specializing them. Like composition, inheritance is everywhere in real life. You inherited your parents genes, and acquired physical attributes from both of them. Technological products (computers, cell phones, etc…) often inherit features from their predecessors. C++ inherited many features from C, the language upon which it is based, and C itself inherited many of its features from the programming languages that came before it.

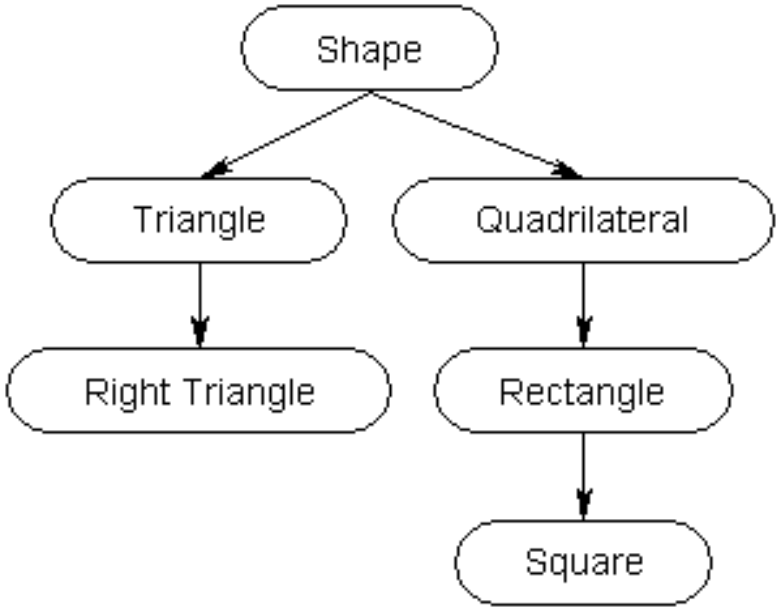
Consider an apple and a banana. Although an apple and a banana are different fruits, both have in common that they *are* fruits. Because apples and bananas *are* fruits, anything that is true of fruits is also true of apples and bananas. For example, all fruits have a name, a flavor, and are tangible objects. Thus, apples and bananas also have a name, a flavor, and are tangible objects. Apples and bananas inherit these properties from the concept of fruit because they *are* fruit. Apples and bananas then define some of these properties in different ways (apples and bananas have different flavors), which is what makes them distinct from each other.



The object being inherited from is called the **parent** or **base**, and the object doing the inheriting is called the **child** or **derived** object. In the above picture, "fruit" is the parent, and both "apple" and "banana" are children. Unlike in composition, where each object has a *has-a* relationship with it's subobjects, in inheritance, each child has an *is-a* relationship with it's parent. An apple *is-a* fruit. A triangle *is-a* shape. Red *is-a* color.

By default, the children receive all of the properties of the parents. However, the children are then free to define or redefine inherited properties (bananas have that unique banana flavor), add new properties (eg. bananas add the property of being "starchy", which is not true of many other fruits), or even hide properties.

It is possible to define entire hierarchies of objects via inheritance. For example, a square is a rectangle, which is a quadrilateral, which is a shape. A right triangle is a triangle, which is also a shape.



Why the need for inheritance in C++?

One of the fundamental ideas behind object-oriented programming is that code should be reusable. However, existing code often does not do EXACTLY what you need it to. For example, what if you have a triangle and you need a square? In this case, we are presented with a number of choices on how to proceed, all of which have various benefits and downsides.

Perhaps the most obvious way to proceed is to change the existing code to do what you want. However, if we do this, we will no longer be able to use it for it's original purpose, so this is rarely a good idea.

A slightly better idea is to make a copy of some or all of the existing code and change it to do what we want. However, this has several major downsides. First, although copy-and-paste seems simple enough, it's actually quite dangerous. A single omitted or misplaced line can cause the program to work incorrectly and can take days to find in a complex program. Renaming a class via search-and-replace can also be dangerous if you inadvertently replace something you didn't mean to. Second, to rewrite the code to make it do what you want, you need to have an intimate understanding what it does. This can be difficult when the code is complex and not adequately documented. Third, and perhaps most relevant, this generally involves duplicating of existing

functionality, which causes a maintenance problem. Improvements or bug fixes have to be added to multiples copies of functions that do essentially the same thing, which wastes programmer time. And that's assuming the programmer realizes multiple copies even exist! If not, some copies may not get the improvements or bug fixes.

Inheritance solves most of these problems in an efficient way. Instead of manually copying and modifying every bit of code your program needs, inheritance allows you directly reuse existing code that meets your needs. You only need to add new features, redefine existing features that do not meet your needs, or hide features you do not want. This is typically much less work (as you are only defining what has changed compared to the base, rather than redefining everything), and safer too. Furthermore, any changes made to the base code automatically get propagated to the inherited code. This means it is possible to change one piece of code (eg. to apply a bug fix) and all derived objects will automatically be updated.

Inheritance does have a couple of potential downsides, but we will cover those in future lessons.

11.2 — Basic inheritance in C++

Now that we've talked about what inheritance is in an abstract sense, let's talk about how it's used within C++.

Inheritance in C++ takes place between classes. When one class inherits from another, the derived class inherits the variables and functions of the base class. These variables and functions become part of the derived class.

A Person base class

Here's a simple base class:

```
#include <string>
class Person
{
public:
    std::string m_strName;
    int m_nAge;
    bool m_bIsMale;

    std::string GetName() { return m_strName; }
    int GetAge() { return m_nAge; }
    bool IsMale() { return m_bIsMale; }

    Person(std::string strName = "", int nAge = 0, bool bIsMale = false)
        : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
    {
    }
};
```

This base class is meant to hold information about a person — in this case, the name, age, and sex. There are two things to note here. First, we have only defined fields that are common to ALL people. This is a generic person class meant to be reused with anybody who is a person. Thus, it's appropriate to only include information used for all people.

Second, note that we've made all of our variables and functions public. This is purely for the sake of keeping these examples simple right now. Normally we would make the variables private. We will cover those cases in future lessons.

A BaseballPlayer derived class

Let's say we wanted to write a program that keeps track of information about some baseball players. Baseball players have information that only people who are baseball players — for example, we might want to store a player's batting average, and the number of home runs they've hit. Here's our incomplete Baseball player class:

```
1  class BaseballPlayer
2  {
3  public:
4      double m_dBattingAverage;
       int m_nHomeRuns;
   };
```

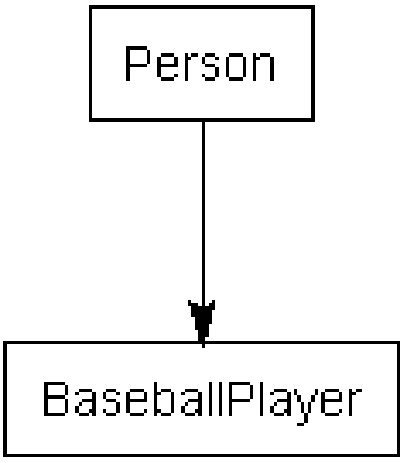
Note that we have not included the baseball player's name, age, or sex in this class, even though we want that information. While we could add member variables to hold this information directly to BaseballPlayer, we've already written a generic Person class that we can simply reuse to handle those details.

Logically, we know that BaseballPlayer and Person have some sort of relationship. Which makes more sense: a baseball player "has a" person, or a baseball player "is a" person? A baseball player "is a" person, therefore, our baseball player class will use inheritance rather than composition.

To inherit our Person class, the syntax is fairly simple. After the `class BaseballPlayer` declaration, we use a colon, the word "public", and the name of the class we wish to inherit. This is called *public inheritance*. We'll talk more about what public inheritance means in a future section.

```
1  // BaseballPlayer publicly inheriting Person
2  class BaseballPlayer : public Person
3  {
4  public:
5      double m_dBattingAverage;
       int m_nHomeRuns;
6
7      BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0)
8          : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
9      {
       }
10 };
```

Using a derivation diagram, our inheritance looks like this:



When BaseballPlayer inherits from Person, BaseballPlayer automatically receives the functions and variables from Person. Thus, BaseballPlayer objects will have 5 member variables (m_dBattingAverage and m_nHomeRuns from BaseballPlayer, and m_strName, m_nAge, and m_bIsMale from Person).

This is easy to prove:

```
#include <iostream>
int main()
{
    // Create a new BaseballPlayer object
    BaseballPlayer cJoe;
    // Assign it a name (we can do this directly because m_strName is
public)
    cJoe.m_strName = "Joe";
    // Print out the name
    std::cout << cJoe.GetName() << std::endl;

    return 0;
}
```

Which prints the value:

```
Joe
```

This compiles and runs because cJoe is a BaseballPlayer, and all BaseballPlayer objects have a m_strName member variable that they inherit from the Person class.

An Employee derived class

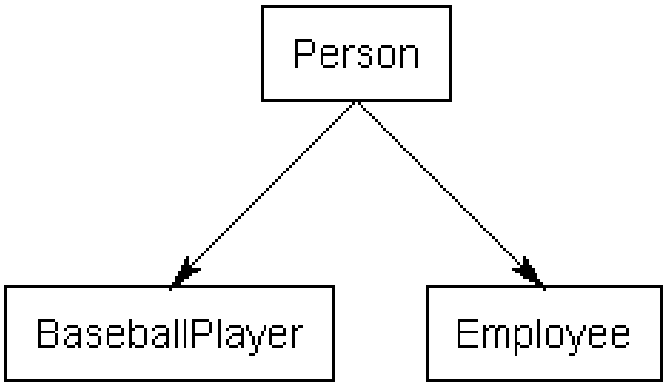
Now let's write another class that also inherits from Person. This time, we'll write an Employee class. An employee "is a" person, so using inheritance is appropriate:

```
// Employee publicly inherits from Person
class Employee: public Person
{
public:
    std::string m_strEmployerName;
    double m_dHourlySalary;
    long m_lEmployeeID;

    Employee(std::string strEmployerName, double dHourlySalary, long
lEmployeeID)
        : m_strEmployerName(strEmployerName),
m_dHourlySalary(dHourlySalary),
            m_lEmployeeID(lEmployeeID)
    {
    }

    double GetHourlySalary() { return m_dHourlySalary; }
    void PrintNameAndSalary()
```

```
    {
        std::cout << m_strName << ": " << m_dHourlySalary << std::endl;
    }
};
```

Employee inherits m_strName, m_nAge, and m_bIsMale from Person (as well as the three access functions), and adds three more member variables and a couple of member function of it's own. Note that PrintNameAndSalary() uses variables both from the class it belongs to (Employee) and the parent class (Person).

This gives us a derivation chart that looks like this:



Note that Employee and BaseballPlayer don't have any direct relationship, even though they both inherit from Person.

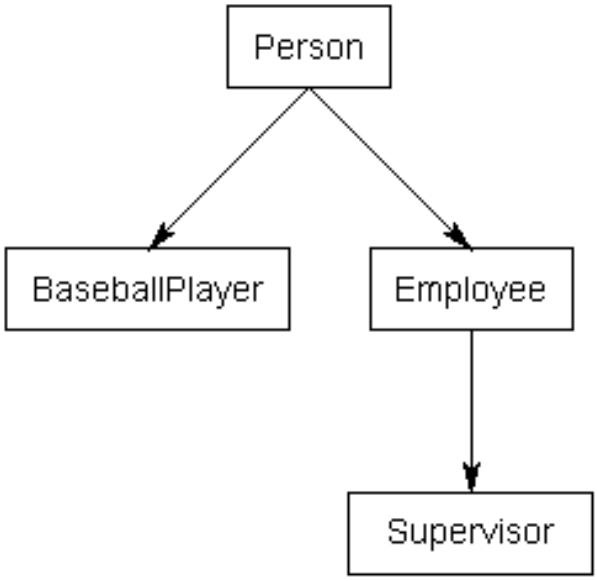
Inheritance chains

It's possible to inherit from a parent that is itself derived from another class. There is nothing noteworthy or special when doing so — everything proceeds as in the examples above.

For example, let's write a Supervisor class. A supervisor is an employee, which is a person. We've already written an Employee class, so let's use that as the base class from which to derive Supervisor:

```
class Supervisor: public Employee
{
public:
    // This Supervisor can oversee a max of 5 employees
    int m_nOverseesIDs[5];
};
```

Now our derivation chart looks like this:



All Supervisor objects inherit the functions and variables from Employee, and add their own m_nOverseesIDs member variable.

By constructing such inheritance chains, we can create a set of reusable classes that are very general (at the top) and become progressively more specific at each level of inheritance.

Conclusion

Inheriting from a base class means we don't have to redefine the information from the base class in our derived classes. We automatically receive the member functions and member variables of the base class through inheritance, and then simply add the additional functions or member variables we want. This not only saves work, but also means that if we ever update or modify the base class (eg. add new functions, or fix a bug), all of our derived classes will automatically inherit the changes!

For example, if we ever added a new function to Person, both Employee and Supervisor would automatically gain access to it. If we added a new variable to Employee, Supervisor would also gain access to it. This allows us to construct new classes in an easy, intuitive, and low-maintenance way!

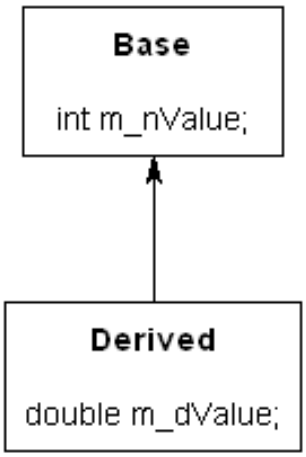
11.3 — Order of construction of derived classes

In the previous lesson on basic inheritance in C++, you learned that classes can inherit members and functions from other classes. In this lesson, we're going to take a closer look at the order of construction that happens when a derived class is instantiated.

First, let's introduce some new classes that will help us illustrate some important points.

```
1  class Base
2  {
3  public:
4      int m_nValue;
5
6      Base(int nValue=0)
7          : m_nValue(nValue)
8      {
9      }
10 };
11
12 class Derived: public Base
13 {
14 public:
15     double m_dValue;
16
17     Derived(double dValue=0.0)
18         : m_dValue(dValue)
19     {
20     }
21 };
```

In this example, Derived is derived from Base. Because Derived inherits functions and variables from Base, it is convenient to think of Derived as a two part class: one part Derived, and one part Base.



You've already seen plenty examples of what happens when we instantiate a normal (non-derived) class:

```
1  int main()
2  {
3      Base cBase;
4
5      return 0;
   }
```

Base is a non-derived class because it does not inherit from anybody. C++ allocates memory for Base, then calls Base's default constructor to do the initialization.

Now let's take a look at what happens when we instantiate a derived class:

```
1  int main()
2  {
3      Derived cDerived;
4
5      return 0;
   }
```

If you were to try this yourself, you wouldn't notice any difference from the previous example where we instantiate the non-derived class. But behind the scenes, things are slightly different. As mentioned, Derived is really two parts: a Base part, and a Derived part. When C++ constructs derived objects, it does so in pieces, starting with the base portion of the class. Once that is complete, it then walks through the inheritance tree and constructs each derived portion of the class.

So what actually happens in this example is that the Base portion of Derived is constructed first. Once the Base portion is finished, the Derived portion is constructed. At this point, there are no more derived classes, so we are done.

This process is actually easy to illustrate.

```
1  #include <iostream>
2  using namespace std;
3
4  class Base
5  {
6  public:
7      int m_nValue;
8
9      Base(int nValue=0)
10         : m_nValue(nValue)
11     {
12         cout << "Base" << endl;
13     }
   };
```

```
14  class Derived: public Base
15  {
16  public:
17      double m_dValue;
18
19      Derived(double dValue=0.0)
20          : m_dValue(dValue)
21      {
22          cout << "Derived" << endl;
23      }
24  };
25
26  int main()
27  {
28      cout << "Instantiating Base" << endl;
29      Base cBase;
30
31      cout << "Instantiating Derived" << endl;
        Derived cDerived;

        return 0;
    }
```

This program produces the following result:

```
Instantiating Base
Base
Instantiating Derived
Base
Derived
```

As you can see, when we constructed Derived, the Base portion of Derived got constructed first. This makes sense: logically, a child can not exist without a parent. It's also the safe way to do things: the child class often uses variables and functions from the parent, but the parent class knows nothing about the child. Instantiating the parent class first ensures those variables are already initialized by the time the derived class is created and ready to use them.

Order of construction for inheritance chains

It is sometimes the case that classes are derived from other classes, which are themselves derived from other classes. For example:

```
1  class A
2  {
3  public:
4      A()
5      {
6          cout << "A" << endl;
7      }
8  };

   class B: public A
```

```
{
public:
    B()
    {
        cout << "B" << endl;
    }
};

class C: public B
{
public:
    C()
    {
        cout << "C" << endl;
    }
};

class D: public C
{
public:
    D()
    {
        cout << "D" << endl;
    }
};
```

Remember that C++ always constructs the "first" or "most base" class first. It then walks through the inheritance tree in order and constructs each successive derived class.

Here's a short program that illustrates the order of creation all along the inheritance chain.

```
int main()
{
    cout << "Constructing A: " << endl;
    A cA;

    cout << "Constructing B: " << endl;
    B cB;

    cout << "Constructing C: " << endl;
    C cC;

    cout << "Constructing D: " << endl;
    D cD;
}
```

This code prints the following:

```
Constructing A:
A
Constructing B:
A
B
```

```
Constructing C:
A
B
C
Constructing D:
A
B
C
D
```

Conclusion

You will note that our example classes in this section have all used default constructors. In the next lesson, we will take a closer look at the special role of constructors in the process of constructing derived classes.

11.4 — Constructors and initialization of derived classes

In the past two lessons,we've looked at some basics about inheritance in C++ and explored the order that derived classes are initialized. In this lesson, we'll take a closer look at the role of constructors in the initialization of derived classes. To do so, we will continue to use the simple Base and Derived class we developed in the previous lesson:

```
1  class Base
2  {
3  public:
4      int m_nValue;
5
6      Base(int nValue=0)
7          : m_nValue(nValue)
8      {
9      }
10 };
11
12 class Derived: public Base
13 {
14 public:
15     double m_dValue;
16
17     Derived(double dValue=0.0)
18         : m_dValue(dValue)
19     {
20     }
21 };
```

With non-derived classes, constructors only have to worry about their own members. For example, consider Base. We can create a Base object like this:

```
1  int main()
2  {
3      Base cBase(5); // use Base(int) constructor
4
5      return 0;
6  }
```

Here's what actually happens when cBase is instantiated:

1. Memory for cBase is set aside
2. The appropriate Base constructor is called
3. The initialization list initializes variables
4. The body of the constructor executes

5. Control is returned to the caller

This is pretty straightforward. With derived classes, things are slightly more complex:

```
1  int main()
2  {
3      Derived cDerived(1.3); // use Derived(double) constructor
4
5      return 0;
   }
```

Here's what actually happens when cDerived is instantiated:

1. Memory for cDerived is set aside (enough for both the Base and Derived portions).
2. The appropriate Derived constructor is called
3. **The Base object is constructed first using the appropriate Base constructor**
4. The initialization list initializes variables
5. The body of the constructor executes
6. Control is returned to the caller

The only real difference between this case and the non-inherited case is that before the Derived constructor can do anything substantial, the Base constructor is called first. The Base constructor sets up the Base portion of the object, control is returned to the Derived constructor, and the Derived constructor is allowed to finish up it's job.

Initializing base class members

One of the current shortcomings of our Derived class as written is that there is no way to initialize m_nValue when we create a Derived object. What if we want to set both m_dValue (from the Derived potion of the object) and m_nValue (from the Base portion of the object) when we create a Derived object?

New programmers often attempt to solve this problem as follows:

```
1  class Derived: public Base
2  {
3  public:
4      double m_dValue;
5
6      Derived(double dValue=0.0, int nValue=0)
7          // does not work
8          : m_dValue(dValue), m_nValue(nValue)
9      {
       }
10 };
```

This is a good attempt, and is almost the right idea. We definitely need to add another parameter to our constructor, otherwise C++ will have no way of knowing what value we want to initialize m_nValue to.

However, C++ prevents classes from initializing inherited member variables in the initialization list of a constructor. In other words, the value of a variable can only be set in an initialization list of a constructor belonging to the same class as the variable.

Why does C++ do this? The answer has to do with const and reference variables. Consider what would happen if m_nValue were const. Because const variables must be initialized with a value at the time of creation, the base class constructor must set it's value when the variable is created. However, when the base class constructor finishes, the derived class constructors initialization lists are then executed. Each derived class would then have the opportunity to initialize that variable, potentially changing it's value! By restricting the initialization of variables to the constructor of the class those variables belong to, C++ ensures that all variables are initialized only once.

The end result is that the above example does not work because m_nValue was inherited from Base, and only non-inherited variables can be changed in the initialization list.

However, inherited variables can still have their values changed in the body of the constructor using an assignment. Consequently, new programmers often also try this:

```
1  class Derived: public Base
2  {
3  public:
4      double m_dValue;
5
6      Derived(double dValue=0.0, int nValue=0)
7          : m_dValue(dValue)
8      {
9          m_nValue = nValue;
10     }
11 };
```

While this actually works in this case, it wouldn't work if m_nValue were a const or a reference (because const values and references have to be initialized in the initialization list of the constructor). It's also inefficient because m_nValue gets assigned a value twice: once in the initialization list of the Base class constructor, and then again in the body of the Derived class constructor.

So how do we properly initialize m_nValue when creating a Derived class object?

In all of the examples so far, when we instantiate a Derived class object, the Base class portion has been created using the default Base constructor. Why does it always use the default Base constructor? Because we never told it to do otherwise!

Fortunately, C++ gives us the ability to explicitly choose which Base class constructor will be called! To do this, simply add a call to the base class Constructor in the initialization list of the derived class:

```
class Derived: public Base
{
public:
    double m_dValue;

    Derived(double dValue=0.0, int nValue=0)
        : Base(nValue), // Call Base(int) constructor with value nValue!
            m_dValue(dValue)
    {
    }
};
```

Now, when we execute this code:

```
int main()
{
    Derived cDerived(1.3, 5); // use Derived(double) constructor

    return 0;
}
```

The base class constructor Base(int) will be used to initialize m_nValue to 5, and the derived class constructor will be used to initialize m_dValue to 1.3!

In more detail, here's what happens:

1. Memory for cDerived is allocated.
2. The Derived(double, int) constructor is called, where dValue = 1.3, and nValue = 5
3. The compiler looks to see if we've asked for a particular Base class constructor. We have! So it calls Base(int) with nValue = 5.
4. The base class constructor initialization list sets m_nValue to 5
5. The base class constructor body executes
6. The base class constructor returns
7. The derived class constuctor initialization list sets m_dValue to 1.3
8. The derived class constructor body executes
9. The derived class constructor returns

This may seem somewhat complex, but it's actually very simple. All that's happening is that the Derived constructor is calling a specific Base constructor to initialize the Base portion of the object. Because m_nValue lives in the Base portion of the object, the Base constructor is the only constructor that can initialize it's value.

Another example

Let's take a look at another pair of class we've previously worked with:

```
#include <string>
class Person
{
public:
    std::string m_strName;
    int m_nAge;
    bool m_bIsMale;

    std::string GetName() { return m_strName; }
    int GetAge() { return m_nAge; }
    bool IsMale() { return m_bIsMale; }

    Person(std::string strName = "", int nAge = 0, bool bIsMale = false)
        : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
    {
    }
};

// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_dBattingAverage;
    int m_nHomeRuns;

    BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0)
       : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
    {
    }
};
```

As we'd previously written it, BaseballPlayer only initializes it's own members and does not specify a Person constructor to use. The means every BaseballPlayer we create is going to use the default Person constructor, which will initialize the name to blank and age to 0. Because it makes sense to give our BaseballPlayer a name and age when we create them, we should modify this constructor to add those parameters.

Here's our new BaseballPlayer class with a constructor that calls the Person constructor to initialize the inherited Person member variables.

```
// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_dBattingAverage;
    int m_nHomeRuns;

    BaseballPlayer(std::string strName = "", int nAge = 0, bool bIsMale =
false,
        double dBattingAverage = 0.0, int nHomeRuns = 0)
        : Person(strName, nAge, bIsMale), // call Person(std::string, int,
bool) to initialize these fields
            m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
```

```
    {
    }
};
```

Now we can create baseball players like this:

```
int main()
{
    BaseballPlayer cPlayer("Pedro Cerrano", 32, true, 0.342, 42);

    return 0;
}
```

To prove that it works:

```
int main()
{
    BaseballPlayer cPlayer("Pedro Cerrano", 32, true, 0.342, 42);

    using namespace std;
    cout << cPlayer.m_strName << endl;
    cout << cPlayer.m_nAge << endl;
    cout << cPlayer.m_nHomeRuns;

    return 0;
}
```

This outputs:

```
Pedro Cerrano
32
42
```

As you can see, the name and age in the base class were properly initialized, as was the number of home runs in the derived class.

Inheritance chains

Classes in an inheritance chain work in exactly the same way.

```
#include <iostream>
using namespace std;

class A
{
public:
    A(int nValue)
    {
        cout << "A: " << nValue << endl;
    }
};

```

```
11
12  class B: public A
13  {
14  public:
15      B(int nValue, double dValue)
16      : A(nValue)
17      {
18          cout << "B: " << dValue << endl;
19      }
20  };
21
22  class C: public B
23  {
24  public:
25      C(int nValue, double dValue, char chValue)
26      : B(nValue, dValue)
27      {
28          cout << "C: " << chValue << endl;
29      }
30  };
31
32  int main()
    {
        C cClass(5, 4.3, 'R');

        return 0;
    }
```

In this example, class C is derived from class B, which is derived from class A. So what happens when we instantiate an object of class C?

First, main() calls C(int, double, char). The C constructor calls B(int, double). The B constructor calls A(int). Because A is not inherited, this is the first class we'll construct. A is constructed, prints the value 5, and returns control to B. B is constructed, prints the value 4.3, and returns control to C. C is constructed, prints the value 'R', and returns control to main(). And we're done!

Thus, this program prints:

```
A: 5
B: 4.3
C: R
```

It is worth mentioning that constructors can only call constructors from their immediate parent/base class. Consequently, the C constructor could not call or pass parameters to the A constructor directly. The C constructor can only call the B constructor (which has the responsibility of calling the A constructor).

Destructors

When a derived class is destroyed, each destructor is called in the reverse order of construction. In the above example, when cClass is destroyed, the C destructor is called first, then the B destructor, then the A destructor.

Summary

Although it is true that the most base class is initialized first, this actually only happens after each constructor has called the parent constructor in turn. This gives us the opportunity to specify which of the parent's constructors we want to use to initialize inherited members. Once the base constructor has finished constructing the base portion of the class, control returns to the derived constructor and it executes as normal.

One of the primary advantages of using a base class constructor to initialize the base class members is that if the base class constructor is ever changed, both the base class and all inherited classes will automatically use the changes! This helps keep maintenance and duplicate code down.

11.5 — Inheritance and access specifiers

In the previous lessons on inheritance, we've been making all of our data members public in order to simplify the examples. In this section, we'll talk about the role of access specifiers in the inheritance process, as well as cover the different types of inheritance possible in C++.

To this point, you've seen the private and public access specifiers, which determine who can access the members of a class. As a quick refresher, public members can be accessed by anybody. Private members can only be accessed by member functions of the same class. Note that this means derived classes can not access private members!

```
class Base
{
private:
    int m_nPrivate; // can only be accessed by Base member functions (not derived classes)
public:
    int m_nPublic; // can be accessed by anybody
};
```

When dealing with inherited classes, things get a bit more complex.

First, there is a third access specifier that we have yet to talk about because it's only useful in an inheritance context. The **protected** access specifier restricts access to member functions of the same class, or those of derived classes.

```
class Base
{
public:
    int m_nPublic; // can be accessed by anybody
private:
    int m_nPrivate; // can only be accessed by Base member functions (but not derived classes)
protected:
    int m_nProtected; // can be accessed by Base member functions, or derived classes.
};

class Derived: public Base
{
public:
    Derived()
    {
        // Derived's access to Base members is not influenced by the type of inheritance used,
        // so the following is always true:

        m_nPublic = 1; // allowed: can access public base members from
```

```
derived class
        m_nPrivate = 2; // not allowed: can not access private base
members from derived class
        m_nProtected = 3; // allowed: can access protected base members
from derived class
    }
};

int main()
{
    Base cBase;
    cBase.m_nPublic = 1; // allowed: can access public members from
outside class
    cBase.m_nPrivate = 2; // not allowed: can not access private members
from outside class
    cBase.m_nProtected = 3; // not allowed: can not access protected
members from outside class
}
```

Second, when a derived class inherits from a base class, the access specifiers may change depending on the method of inheritance. There are three different ways for classes to inherit from other classes: public, private, and protected.

To do so, simply specify which type of access you want when choosing the class to inherit from:

```
// Inherit from Base publicly
class Pub: public Base
{
};

// Inherit from Base privately
class Pri: private Base
{
};

// Inherit from Base protectedly
class Pro: protected Base
{
};

class Def: Base // Defaults to private inheritance
{
};
```

If you do not choose an inheritance type, C++ defaults to private inheritance (just like members default to private access if you do not specify otherwise).

That gives us 9 combinations: 3 member access specifiers (public, private, and protected), and 3 inheritance types (public, private, and protected).

The rest of this section will be devoted to explaining the difference between these.

Before we get started, the following should be kept in mind as we step through the examples. There are three ways that members can be accessed:

- A class can always access it's own members regardless of access specifier.
- The public accesses the members of a class based on the access specifiers of that class.
- A derived class accesses inherited members based on the access specifiers of its immediate parent. A derived class can always access it's own members regardless of access specifier.

This may be a little confusing at first, but hopefully will become clearer as we step through the examples.

Public inheritance

Public inheritance is by far the most commonly used type of inheritance. In fact, very rarely will you use the other types of inheritance, so your primary focus should be on understanding this section. Fortunately, public inheritance is also the easiest to understand. When you inherit a base class publicly, all members keep their original access specifications. Private members stay private, protected members stay protected, and public members stay public.

```
1  class Base
2  {
3  public:
4      int m_nPublic;
5  private:
6      int m_nPrivate;
7  protected:
8      int m_nProtected;
9  };
10
11 class Pub: public Base
12 {
13     // Public inheritance means:
14     // m_nPublic stays public
15     // m_nPrivate stays private
16     // m_nProtected stays protected
17
18     Pub()
19     {
20         // The derived class always uses the immediate parent's class
21 access specifications
22         // Thus, Pub uses Base's access specifiers
23         m_nPublic = 1; // okay: anybody can access public members
24         m_nPrivate = 2; // not okay: derived classes can't access private
25 members in the base class!
26         m_nProtected = 3; // okay: derived classes can access protected
   members
   }
};

int main()
{
```

```
27      // Outside access uses the access specifiers of the class being
28  accessed.
29      // In this case, the access specifiers of cPub.  Because Pub has
30  inherited publicly from Base,
31      // no access specifiers have been changed.
32      Pub cPub;
33      cPub.m_nPublic = 1; // okay: anybody can access public members
34      cPub.m_nPrivate = 2; // not okay: can not access private members from
35  outside class
36      cPub.m_nProtected = 3; // not okay: can not access protected members
    from outside class
    }
```

This is fairly straightforward. The things worth noting are:

1. Derived classes can not directly access private members of the base class.
2. The protected access specifier allows derived classes to directly access members of the base class while not exposing those members to the public.
3. The derived class uses access specifiers from the base class.
4. The outside uses access specifiers from the derived class.

To summarize in table form:

Public inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public	Public	Yes	Yes
Private	Private	No	No
Protected	Protected	Yes	No

Private inheritance

With private inheritance, all members from the base class are inherited as private. This means private members stay private, and protected and public members become private.

Note that this does not affect that way that the derived class accesses members inherited from its parent! It only affects the code trying to access those members through the derived class.

```
1   class Base
2   {
3   public:
4       int m_nPublic;
5   private:
6       int m_nPrivate;
7   protected:
8       int m_nProtected;
9   };

    class Pri: private Base
```

```
{
    // Private inheritance means:
    // m_nPublic becomes private
    // m_nPrivate stays private
    // m_nProtected becomes private

    Pri()
    {
        // The derived class always uses the immediate parent's class access specifications
        // Thus, Pub uses Base's access specifiers
        m_nPublic = 1; // okay: anybody can access public members
        m_nPrivate = 2; // not okay: derived classes can't access private members in the base class!
        m_nProtected = 3; // okay: derived classes can access protected members
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    // Note that because Pri has inherited privately from Base,
    // all members of Base have become private when access through Pri.
    Pri cPri;
    cPri.m_nPublic = 1; // not okay: m_nPublic is now a private member when accessed through Pri
    cPri.m_nPrivate = 2; // not okay: can not access private members from outside class
    cPri.m_nProtected = 3; // not okay: m_nProtected is now a private member when accessed through Pri

    // However, we can still access Base members as normal through Base:
    Base cBase;
    cBase.m_nPublic = 1; // okay, m_nPublic is public
    cBase.m_nPrivate = 2; // not okay, m_nPrivate is private
    cBase.m_nProtected = 3; // not okay, m_nProtected is protected
}
```

To summarize in table form:

Private inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public	Private	Yes	No
Private	Private	No	No
Protected	Private	Yes	No

Protected inheritance

Protected inheritance is the last method of inheritance. It is almost never used, except in very particular cases. With protected inheritance, the public and protected members become protected, and private members stay private.

To summarize in table form:

Protected inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public	Protected	Yes	No
Private	Private	No	No
Protected	Protected	Yes	No

Protected inheritance is similar to private inheritance. However, classes derived from the derived class still have access to the public and protected members directly. The public (stuff outside the class) does not.

Summary

The way that the access specifiers, inheritance types, and derived classes interact causes a lot of confusion. To try and clarify things as much as possible:

First, the base class sets it's access specifiers. The base class can always access it's own members. The access specifiers only affect whether outsiders and derived classes can access those members.

Second, derived classes have access to base class members based on the access specifiers of the immediate parent. The way a derived class accesses inherited members is not affected by the inheritance method used!

Finally, derived classes can change the access type of inherited members based on the inheritance method used. This does not affect the derived classes own members, which have their own access specifiers. It only affects whether outsiders and classes derived from the derived class can access those inherited members.

A final example:

```
1  class Base
2  {
3  public:
4      int m_nPublic;
5  private:
6      int m_nPrivate;
7  protected:
8      int m_nProtected;
   };
```

Base can access it's own members without restriction. The public can only access m_nPublic. Derived classes can access m_nPublic and m_nProtected.

```
1  class D2: private Base
2  {
3  public:
4      int m_nPublic2;
5  private:
6      int m_nPrivate2;
7  protected:
       int m_nProtected2;
   }
```

D2 can access it's own members without restriction. D2 can access Base's members based on Base's access specifiers. Thus, it can access m_nPublic and m_nProtected, but not m_nPrivate. Because D2 inherited Base privately, m_nPublic, m_nPrivate, and m_nProtected are now private when accessed through D2. This means the public can not access any of these variables when using a D2 object, nor can any classes derived from D2.

```
1  class D3: public D2
2  {
3  public:
4      int m_nPublic3;
5  private:
6      int m_nPrivate3;
7  protected:
       int m_nProtected3;
   };
```

D3 can access it's own members without restriction. D3 can access D2′s members based on D2′s access specifiers. Thus, D3 has access to m_nPublic2 and m_nProtected2, but not m_nPrivate2. D3 access to Base members is controlled by the access specifier of it's immediate parent. This means D3 does not have access to any of Base's members because they all became private when D2 inherited them.

11.6 — Adding, changing, and hiding members in a derived class

In the introduction to inheritance lesson, we mentioned that one of the biggest benefits of using derived classes is the ability to reuse already written code. You can inherit the base class functionality and then add new functionality, modify existing functionality, or hide functionality you don't want. In this lesson, we'll take a closer look at how this is done.

First, let's start with a simple base class:

```
#include <iostream>
using namespace std;

class Base
{
protected:
    int m_nValue;

public:
    Base(int nValue)
        : m_nValue(nValue)
    {
    }

    void Identify() { cout << "I am a Base" << endl; }
};
```

Now, let's create a derived class that inherits from Base. Because we want Derived to be able to set the value of m_nValue when Derived objects are instantiated, we'll make the Derived constructor call the Base constructor in the initialization list.

```
class Derived: public Base
{
public:
    Derived(int nValue)
        :Base(nValue)
    {
    }
};
```

We'll develop Derived over the course of this lesson.

Adding new functionality

Because we have access to the source code of the Base class, we could add functionality directly to Base. However, there may be times when we do not want to, or can not. Consider the case where you have just purchased a library of code from a 3rd party vendor, but need some extra functionality. You could add to the original code, but this isn't the best solution. What if the vendor sends you an update? Either your additions will be overwritten, or you'll have to manually migrate them. It's also common to for developers to release header files containing class definitions, but release the implementation code precompiled — this means you can use the code, but you won't have the ability to modify it directly.

In either case, the best answer is to derive your own class, and add the functionality you want to the derived class.

One obvious omission from the Base class is a way for the public to access m_nValue. Normally we'd write an access function in the Base class — but for the sake of example we're going to add it to the derived class instead. Because m_nValue has been declared as protected in the Base class, Derived has direct access to it.

To add new functionality to a derived class, simply declare that functionality in the derived class like normal:

```
1  class Derived: public Base
2  {
3  public:
4      Derived(int nValue)
5          :Base(nValue)
6      {
7      }
8
9      int GetValue() { return m_nValue; }
   };
```

Now the public will be able to call GetValue() in order to access the value of m_nValue.

```
1  int main()
2  {
3      Derived cDerived(5);
4      cout << "cDerived has value " << cDerived.GetValue() << endl;
5
6      return 0;
   }
```

This produces the result:

```
cDerived has value 5
```

Although it may be obvious, objects of type Base have no access to the GetValue() function in Derived. The following does not work:

```
1  int main()
2  {
3      Base cBase(5);
4      cout << "cBase has value " << cBase.GetValue() << endl;
5
6      return 0;
7  }
```

This is because there is no GetValue() function in Base. GetValue() belongs to Derived. Because Derived is a Base, Derived has access to stuff in Base. However, Base does not have access to anything in Derived.

Redefining functionality

When a member function is called with a derived class object, the the compiler first looks to see if that member exists in the derived class. If not, it begins walking up the inheritance chain and checking whether the member has been defined in any of the inherited classes. It uses the first one it finds.

Consequently, take a look at the following example:

```
1  int main()
2  {
3      Base cBase(5);
4      cBase.Identify();
5
6      Derived cDerived(7);
7      cDerived.Identify()
8
9      return 0;
10 }
```

This prints

```
I am a Base
I am a Base
```

When cDerived.Identify() is called, the compiler looks to see if Identify() has been defined in the Derived class. It hasn't. Then it starts looking in the inherited classes (which in this case is Base). Base has defined a Identify() function, so it uses that one. In other words, Base::Identify() was used because Derived::Identify() doesn't exist.

However, if we had defined Derived::Identify() in the Derived class, it would have been used instead. This means that we can make functions work differently with our derived classes by redefining them in the derived class!

In our above example, it would be more accurate if `cDerived.Identify()` printed "I am a Derived". Let's modify Identify() so it returns the correct response when we call Identify() with a Derived object.

To modify a function the way a function defined in a base class works in the derived class, simply redefine the function in the derived class.

```
1  class Derived: public Base
2  {
3  public:
4      Derived(int nValue)
5          :Base(nValue)
6      {
7      }
8
9      int GetValue() { return m_nValue; }
10
11     // Here's our modified function
12     void Identify() { cout << "I am a Derived" << endl; }
13 };
```

Here's the same example as above, using the new Derived::Identify() function:

```
1  int main()
2  {
3      Base cBase(5);
4      cBase.Identify();
5
6      Derived cDerived(7);
7      cDerived.Identify()
8
9      return 0;
10 }
```

```
I am a Base
I am a Derived
```

A word of warning: the prototype (return value, function name, and parameters) of the function in the derived class must be exactly identical to that in the base class. If the return value does not match, the compiler will issue an error. If the parameters or function name are different, C++ will treat this as an added function rather than a modified function!

Also note that when you redefine a function in the derived class, the derived function does not inherit the access specifier of the function with the same name in the base class. It uses whatever access specifier it is defined under in the derived class. Therefore, a function that is defined as private in the base class can redefined as public in the derived class, or vice-versa!

Adding to existing functionality

Sometimes we don't want to completely replace a base class function, but instead want to add additional functionality to it. In the above example, note that Derived::Identify() completely hides Base::Identify()! This may not be what we want. It is possible to have our Derived function call the Base function of the same name (in order to reuse code) and then add additional functionality to it.

To have a derived function call a base function of the same name, simply do a normal function call, but prefix the function with the scope qualifier (the name of the base class and two colons). The following example redefines Derived::Identify() so it first calls Base::Identify() and then does it's own additional stuff.

```
1  class Derived: public Base
2  {
3  public:
4      Derived(int nValue)
5          :Base(nValue)
6      {
7      }
8
9      int GetValue() { return m_nValue; }
10
11     void Identify()
12     {
13         Base::Identify(); // call Base::Identify() first
14         cout << "I am a Derived"; // then identify ourselves
15     }
16 };
```

Now consider the following example:

```
1  int main()
2  {
3      Base cBase(5);
4      cBase.Identify();
5
6      Derived cDerived(7);
7      cDerived.Identify()
8
9      return 0;
10 }
```

```
I am a Base
I am a Base
I am a Derived
```

When `cDerived.Identify()` is executed, it resolves to Derived::Identify(). However, the first thing Derived::Identify() does is call Base::Identify(), which prints "I am a Base". When Base::Identify() returns, Derived::Identify() continues executing and prints "I am a Derived".

This is all pretty straightforward. The real lesson to take away from this is that if you want to call a function in a base class that has been redefined in the derived class, you need to use the scope resolution operator (::) to explicitly say which version of the function you want.

If we had defined Derived::Identify() like this:

```
1  void Identify()
2  {
3      Identify(); // Note: no scope resolution!
4      cout << "I am a Derived"; // then identify ourselves
   }
```

Identify() without a scope resolution qualifier would default to the Identify() in the current class, which would be Derived::Identify(). This would cause Derived::Identify() to call itself, which would lead to an infinite loop!

Hiding functionality

In C++, it is not possible to remove functionality from a class. However, it is possible to hide existing functionality.

As mentioned above, if you redefine a function, it uses whatever access specifier it's declared under in the derived class. Therefore, we could redefine a public function as private in our derived class, and the public would lose access to it. However, C++ also gives us the ability to change a base member's access specifier in the derived class without even redefining the member! This is done by simply naming the member (using the scope resolution operator) to have it's access changed in the derived class under the new access specifier.

For example, consider the following Base:

```
1  class Base
2  {
3  private:
4      int m_nValue;
5
6  public:
7      Base(int nValue)
8          : m_nValue(nValue)
9      {
10     }
11
12 protected:
13     void PrintValue() { cout << m_nValue; }
14 };
```

Because Base::PrintValue() has been declared as protected, it can only be called by Base or it's derived classes. The public can not access it.

Let's define a Derived class that changes the access specifier of PrintValue() to public:

```
class Derived: public Base
{
public:
    Derived(int nValue)
        : Base(nValue)
    {
    }

    // Base::PrintValue was inherited as protected, so the public has no
access
    // But we're changing it to public by declaring it in the public
section
    Base::PrintValue;
};
```

This means that this code will now work:

```
int main()
{
    Derived cDerived(7);

    // PrintValue is public in Derived, so this is okay
    cDerived.PrintValue(); // prints 7
    return 0;
}
```

Note that Base::PrintValue does not have the function call operator (()) attached to it.

We can also use this to make public members private:

```
class Base
{
public:
    int m_nValue;
};

class Derived: public Base
{
private:
    Base::m_nValue;

public:
    Derived(int nValue)
    {
        m_nValue = nValue;
    }
};

int main()
{
    Derived cDerived(7);

    // The following won't work because m_nValue has been redefined as
```

```
20  private
21      cout << cDerived.m_nValue;
22
23      return 0;
    }
```

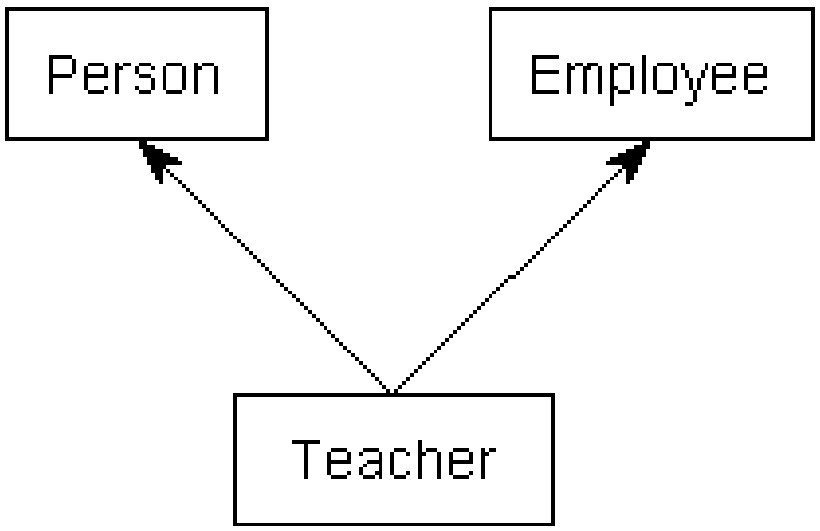
Note that this allowed us to take a poorly designed base class and encapsulate it's data in our derived class.
(Alternatively, instead of inheriting Base's members publicly and making m_nValue private by overriding it's access specifier, we could have inherited Base privately, which would have caused all of Base's member to be inherited privately in the first place).

One word of caution: you can only change the access specifiers of base members the class would normally be able to access. Therefore, you can never change the access specifier of a base member from private to protected or public, because derived classes do not have access to private members of the base class.

11.7 — Multiple inheritance

So far, all of the examples of inheritance we've presented have been single inheritance — that is, each inherited class has one and only one parent. However, C++ provides the ability to do multiple inheritance. **Multiple inheritance** enables a derived class to inherit members from more than one parent.

Let's say we wanted to write a program to keep track of a bunch of teachers. A teacher is a person. However, a teacher is also an employee (they are their own employer if working for themselves). Multiple inheritance can be used to create a Teacher class that inherits properties from both Person and Employee. To use multiple inheritance, simply specify each base class (just like in single inheritance), separated by a comma.




```
1  #include <string>
2  class Person
3  {
4  private:
5      std::string m_strName;
6      int m_nAge;
7      bool m_bIsMale;
8
9  public:
10     Person(std::string strName, int nAge, bool bIsMale)
11         : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
12     {
13     }

       std::string GetName() { return m_strName; }
       int GetAge() { return m_nAge; }
```

```
    bool IsMale() { return m_bIsMale; }
};

class Employee
{
private:
    std::string m_strEmployer;
    double m_dWage;

public:
    Employee(std::string strEmployer, double dWage)
        : m_strEmployer(strEmployer), m_dWage(dWage)
    {
    }

    std::string GetEmployer() { return m_strEmployer; }
    double GetWage() { return m_dWage; }
};

// Teacher publicly inherits Person and Employee
class Teacher: public Person, public Employee
{
private:
     int m_nTeachesGrade;

public:
    Teacher(std::string strName, int nAge, bool bIsMale, std::string
strEmployer, double dWage, int nTeachesGrade)
        : Person(strName, nAge, bIsMale), Employee(strEmployer, dWage),
m_nTeachesGrade(nTeachesGrade)
    {
    }
};
```

Problems with multiple inheritance

While multiple inheritance seems like a simple extension of single inheritance, multiple inheritance introduces a lot of issues that can markedly increase the complexity of programs and make them a maintenance nightmare. Let's take a look at some of these situations.

First, ambiguity can result when multiple base classes contain a function with the same name. For example:

```
class USBDevice
{
private:
    long m_lID;

public:
    USBDevice(long lID)
        : m_lID(lID)
    {
    }
```

```
9
10      long GetID() { return m_lID; }
11  };
12
13  class NetworkDevice
14  {
15  private:
16      long m_lID;
17
18  public:
19      NetworkDevice(long lID)
20          : m_lID(lID)
21      {
22      }
23
24      long GetID() { return m_lID; }
25  };
26
27  class WirelessAdaptor: public USBDevice, public NetworkDevice
28  {
29  public:
30      WirelessAdaptor(long lUSBID, long lNetworkID)
31          : USBDevice(lUSBID), NetworkDevice(lNetworkID)
32      {
33      }
34  };
35
36  int main()
37  {
38      WirelessAdaptor c54G(5442, 181742);
39      cout << c54G.GetID(); // Which GetID() do we call?
40
41      return 0;
42  }
43
```

When `c54G.GetID()` is evaluated, the compiler looks to see if WirelessAdaptor contains a function named GetID(). It doesn't. The compiler then looks to see if any of the base classes have a function named GetID(). See the problem here? The problem is that c54G actually contains TWO GetID() functions: one inherited from USBDevice, and one inherited from WirelessDevice. Consequently, this function call is ambiguous, and you will receive a compiler error if you try to compile it.

However, there is a way to work around this problem: you can explicitly specify which version you meant to call:

```
1   int main()
2   {
3       WirelessAdaptor c54G(5442, 181742);
4       cout << c54G.USBDevice::GetID();
5
6       return 0;
```

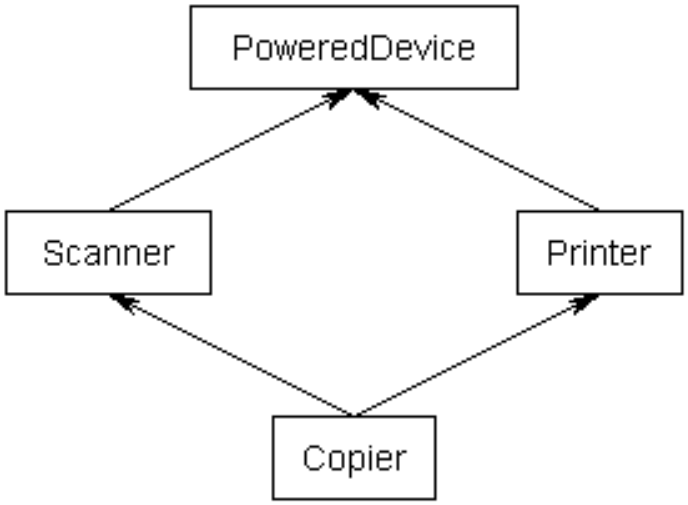
```
6 }
```

While this workaround is pretty simple, you can see how things can get complex when your class inherits from four or six base classes, which inherit from other classes themselves. The potential for naming conflicts increases exponentially as you inherit more classes, and each of these naming conflicts needs to be resolved explicitly.

Second, and more serious is the diamond problem, which your author likes to call the "diamond of doom". This occurs when a class multiply inherits from two classes which each inherit from a single base class. This leads to a diamond shaped inheritance pattern.

For example, consider the following set of classes:

```
1  class PoweredDevice
2  {
3  };
4
5  class Scanner: public PoweredDevice
6  {
7  };
8
9  class Printer: public PoweredDevice
10 {
11 };
12
13 class Copier: public Scanner, public Printer
14 {
15 };
```



Scanners and printers are both powered devices, so they derived from PoweredDevice. However, a copy machine incorporates the functionality of both Scanners and Printers.

There are many issues that arise in this context, including whether Copier should have one or two copies of PoweredDevice, and how to resolve certain types of ambiguous references. While most of these issues can be addressed through explicit scoping, the maintenance overhead added to your classes in order to deal with the added complexity can cause development time to skyrocket.

Is multiple inheritance more trouble than it's worth?

As it turns out, most of the problems that can be solved using multiple inheritance can be solved using single inheritance as well. Many object-oriented languages (eg. Smalltalk, PHP) do not even support multiple inheritance. Many relatively modern languages such as Java and C# restricts classes to single inheritance of normal classes, but allow multiple inheritance of interface classes (which we will talk about later). The driving idea behind disallowing multiple inheritance in these languages is that it simply makes the language too complex, and ultimately causes more problems than it fixes.

Many authors and experienced programmers believe multiple inheritance in C++ should be avoided at all costs due to the many potential problems it brings. Your author does not agree with this approach, because there are times and situations when multiple inheritance is the best way to proceed. However, multiple inheritance should be used extremely judiciously.

As an interesting aside, you have already been using classes written using multiple inherited without knowing it: the iostream library objects cin and cout are both implemented using multiple inheritance.

11.8 — Virtual base classes

Note: This section is an advanced topic and can be skipped or skimmed if desired.

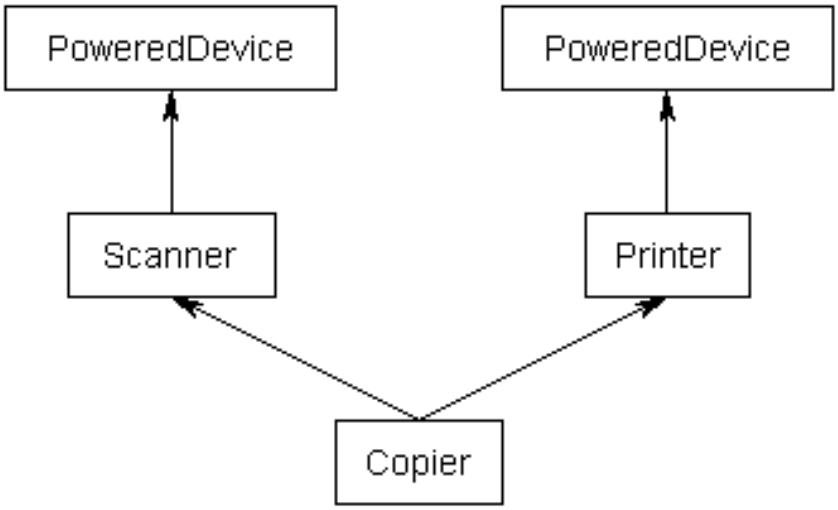
In the previous section on multiple inheritance, we left off talking about the "diamond problem". In this section, we will resume this discussion.

Virtual base classes

Here is our example from the previous lesson, with some constructors:

```
1  class PoweredDevice
2  {
3  public:
4      PoweredDevice(int nPower)
5      {
6          cout << "PoweredDevice: " << nPower << endl;
7      }
8  };
9
10 class Scanner: public PoweredDevice
11 {
12 public:
13     Scanner(int nScanner, int nPower)
14         : PoweredDevice(nPower)
15     {
16         cout << "Scanner: " << nScanner << endl;
17     }
18 };
19
20 class Printer: public PoweredDevice
21 {
22 public:
23     Printer(int nPrinter, int nPower)
24         : PoweredDevice(nPower)
25     {
26         cout << "Printer: " << nPrinter << endl;
27     }
28 };
29
30 class Copier: public Scanner, public Printer
31 {
32 public:
33     Copier(int nScanner, int nPrinter, int nPower)
34         : Scanner(nScanner, nPower), Printer(nPrinter, nPower)
35     {
36     }
37 };
```

If you were to create a Copier class object, by default you would end up with two copies of the PoweredDevice class — one from Printer, and one from Scanner. This has the following structure:



We can create a short example that will show this in action:

```
1  int main()
2  {
3      Copier cCopier(1, 2, 3);
   }
```

This produces the result:

```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2
```

As you can see, PoweredDevice got constructed twice.

While this is sometimes what you want, other times you may want only one copy of PoweredDevice to be shared by both Scanner and Printer. To share a base class, simply insert the "virtual" keyword in the inheritance list of the derived class. This creates what is called a **virtual base class**, which means there is only one base object that is shared. Here is the an example (without constructors for simplicity) showing how to use to virtual keyword to create a shared base class:

```
1  class PoweredDevice
2  {
3  };
4
   class Scanner: virtual public PoweredDevice
```

```
{
};

class Printer: virtual public PoweredDevice
{
};

class Copier: public Scanner, public Printer
{
};
```

Now, when you create a Copier class, you will get only one copy of PoweredDevice that will be shared by both Scanner and Printer.

However, this leads to one more problem: if Scanner and Printer share a PoweredDevice base class, who is responsible for creating it? The answer, as it turns out, is Copier. The Copier constructor is responsible for creating PoweredDevice. Consequently, this is one time when Copier is allowed to call a non-immediate-parent constructor directly:

```
class PoweredDevice
{
public:
    PoweredDevice(int nPower)
    {
        cout << "PoweredDevice: " << nPower << endl;
    }
};

class Scanner: virtual public PoweredDevice
{
public:
    Scanner(int nScanner, int nPower)
        : PoweredDevice(nPower)
    {
        cout << "Scanner: " << nScanner << endl;
    }
};

class Printer: virtual public PoweredDevice
{
public:
    Printer(int nPrinter, int nPower)
        : PoweredDevice(nPower)
    {
        cout << "Printer: " << nPrinter << endl;
    }
};

class Copier: public Scanner, public Printer
{
public:
    Copier(int nScanner, int nPrinter, int nPower)
        :   Scanner(nScanner,   nPower),   Printer(nPrinter,   nPower),
```

```
29  PoweredDevice(nPower)
30      {
31      }
32  };
```

This time, our previous example:

```
1  int main()
2  {
3      Copier cCopier(1, 2, 3);
4  }
```

produces the result:

```
PoweredDevice: 3
Scanner: 1
Printer: 2
```

As you can see, PoweredDevice only gets constructed once.

There are a few details that we would be remiss if we did not mention.

First, virtual base classes are created before non-virtual base classes, which ensures all bases get created before their derived classes.

Second, note that the Scanner and Printer constructors still have calls to the PoweredDevice constructor. If we are creating an instance of Copier, these constructor calls are simply ignored because Copier is responsible for creating the PoweredDevice, not Scanner or Printer. However, if we were to create an instance of Scanner or Printer, the virtual keyword is ignored, those constructor calls would be used, and normal inheritance rules apply.

Third, if a class inherits one or more classes that have virtual parents, the most derived class is responsible for constructing the virtual base class. In this case, Copier inherits Printer and Scanner, both of which have a PoweredDevice virtual base class. Copier, the most derived class, is responsible for creation of PoweredDevice. Note that this is true even in a single inheritance case: if Copier was singly inherited from Printer, and Printer was virtually inherited from PoweredDevice, Copier is still responsible for creating PoweredDevice.

Virtual Functions

12.1 — Pointers and references to the base class of derived objects

In the previous chapter, you learned all about how to use inheritance to derive new classes from existing classes. In this chapter, we are going to focus on one of the most important and powerful aspects of inheritance — virtual functions.

But before we discuss what virtual functions are, let's first set the table for why we need them.

In the chapter on construction of derived classes, you learned that when you create a derived class, it is composed of multiple parts: one part for each inherited class, and a part for itself.

For example, here's a simple case:

```
1  class Base
2  {
3  protected:
4      int m_nValue;
5
6  public:
7      Base(int nValue)
8          : m_nValue(nValue)
9      {
10     }
11
12     const char* GetName() { return "Base"; }
13     int GetValue() { return m_nValue; }
14 };
15
16 class Derived: public Base
17 {
18 public:
19     Derived(int nValue)
20         : Base(nValue)
21     {
22     }
23
24     const char* GetName() { return "Derived"; }
25     int GetValueDoubled() { return m_nValue * 2; }
26 };
```

When we create a Derived object, it contains a Base part (which is constructed first), and a Derived part (which is constructed second). Remember that inheritance implies an is-a relationship between two classes. Since a Derived is-a Base, it is appropriate that Derived contain a Base part of it.

Pointers, references, and derived classes

It should be fairly intuitive that we can set Derived pointers and references to Derived objects:

```
1  int main()
```

```
{
    using namespace std;
    Derived cDerived(5);
    cout << "cDerived is a " << cDerived.GetName() << " and has value " << cDerived.GetValue() << endl;

    Derived &rDerived = cDerived;
    cout << "rDerived is a " << rDerived.GetName() << " and has value " << rDerived.GetValue() << endl;

    Derived *pDerived = &cDerived;
    cout << "pDerived is a " << pDerived->GetName() << " and has value " << pDerived->GetValue() << endl;

    return 0;
}
```

This produces the following output:

```
cDerived is a Derived and has value 5
rDerived is a Derived and has value 5
pDerived is a Derived and has value 5
```

However, since Derived has a Base part, a more interesting question is whether C++ will let us set a Base pointer or reference to a Derived object. It turns out, we can!

```
int main()
{
    using namespace std;
    Derived cDerived(5);

    // These are both legal!
    Base &rBase = cDerived;
    Base *pBase = &cDerived;

    cout << "cDerived is a " << cDerived.GetName() << " and has value " << cDerived.GetValue() << endl;
    cout << "rBase is a " << rBase.GetName() << " and has value " << rBase.GetValue() << endl;
    cout << "pBase is a " << pBase->GetName() << " and has value " << pBase->GetValue() << endl;

    return 0;
}
```

This produces the result:

```
cDerived is a Derived and has value 5
rBase is a Base and has value 5
pBase is a Base and has value 5
```

This result may not be quite what you were expecting at first!

It turns out that because rBase and pBase are a Base reference and pointer, they can only see members of Base (or any classes that Base inherited). So even though Derived::GetName() is an override of Base::GetName(), the Base pointer/reference can not see Derived::GetName(). Consequently, they call Base::GetName(), which is why rBase and pBase report that they are a Base rather than a Derived.

Note that this also means it is not possible to call Derived::GetValueDoubled() using rBase or pBase. They are unable to see anything in Derived.

Here's another slightly more complex example that we'll build on in the next lesson:

```
1  #include <string>
2  class Animal
3  {
4  protected:
5      std::string m_strName;
6
7      // We're making this constructor protected because
8      // we don't want people creating Animal objects directly,
9      // but we still want derived classes to be able to use it.
10     Animal(std::string strName)
11         : m_strName(strName)
12     {
13     }
14
15 public:
16     std::string GetName() { return m_strName; }
17     const char* Speak() { return "???"; }
18 };
19
20 class Cat: public Animal
21 {
22 public:
23     Cat(std::string strName)
24         : Animal(strName)
25     {
26     }
27
28     const char* Speak() { return "Meow"; }
29 };
30
31 class Dog: public Animal
32 {
33 public:
34     Dog(std::string strName)
35         : Animal(strName)
    {
    }

    const char* Speak() { return "Woof"; }
};

int main()
```

```
{
    Cat cCat("Fred");
    cout << "cCat is named " << cCat.GetName() << ", and it says " <<
cCat.Speak() << endl;

    Dog cDog("Garbo");
    cout << "cDog is named " << cDog.GetName() << ", and it says " <<
cDog.Speak() << endl;

    Animal *pAnimal = &cCat;
    cout << "pAnimal is named " << pAnimal->GetName() << ", and it says "
<< pAnimal->Speak() << endl;

    Animal *pAnimal = &cDog;
    cout << "pAnimal is named " << pAnimal->GetName() << ", and it says "
<< pAnimal->Speak() << endl;

    return 0;
}
```

This produces the result:

```
cCat is named Fred, and it says Meow
cDog is named Garbo, and it says Woof
pAnimal is named Fred, and it says ???
pAnimal is named Garbo, and it says ???
```

We see the same issue here. Because pAnimal is an Animal pointer, it can only see the Animal class. Consequently, `pAnimal->Speak()` calls Animal::Speak() rather than the Dog::Speak() or Cat::Speak() function.

Use for pointers and references to base classes

Now you might be saying, "The above examples seem kind of silly. Why would I set a pointer or reference to the base class of a derived object when I can just use the derived object?" It turns out that there are quite a few good reasons.

First, let's say you wanted to write a function that printed an animal's name and sound. Without using a pointer to a base class, you'd have to write it like this:

```
void Report(Cat &cCat)
{
    cout << cCat.GetName() << " says " << cCat.Speak() << endl;
}

void Report(Dog &cDog)
{
    cout << cDog.GetName() << " says " << cDog.Speak() << endl;
}
```

Not too difficult, but consider what would happen if we had 30 different animal types instead of 2. You'd have to write 30 almost identical functions! Plus, if you ever added a new type of animal, you'd have to write a new function for that one too. This is a huge waste of time considering the only real difference is the type of the parameter.

However, because Cat and Dog are derived from Animal, Cat and Dog have an Animal part. Therefore, it makes sense that we should be able to do something like this:

```
1  void Report(Animal &rAnimal)
2  {
3      cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;
   }
```

This would let us pass in any class derived from Animal, even ones that we haven't thought of yet! Instead of one function per animal, we get one function that works with all classes derived from Animal!

The problem, is of course, that because cAnimal is an Animal reference, `cAnimal.Speak()` will call Animal::Speak() instead of the derived version of Speak().

Second, let's say you had 3 cats and 3 dogs that you wanted to keep in an array for easy access. Because arrays can only hold objects of one type, without a pointer or reference to a base class, you'd have to do it like this:

```
1  Cat acCats[] = { Cat("Fred"), Cat("Tyson"), Cat("Zeke") };
2  Dog acDogs[] = { Dog("Garbo"), Dog("Pooky"), Dog("Truffle") };
3
4  for (int iii=0; iii < 3; iii++)
5      cout << acCats[iii].GetName() << " says " << acCats[iii].Speak() <<
   endl;
6
7  for (int iii=0; iii < 3; iii++)
8      cout << acDogs[iii].GetName() << " says " << acDogs[iii].Speak() <<
   endl;
```

Now, consider what would happen if you had 30 different types of animals. You'd need 30 arrays, one for each type of animal!

However, because both Cat and Dog are Animal, it makes sense that we should be able to do something like this:

```
1  Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");
2  Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");
3
4  // Set up an array of pointers to animals, and set those pointers to our
   Cat and Dog objects
5  Animal *apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson,
6  &cZeke };
7  for (int iii=0; iii < 6; iii++)
       cout << apcAnimals[iii]->GetName() << " says " << apcAnimals[iii]-
```

```
    >Speak() << endl;
```

While this compiles and executes, unfortunately the fact that apcAnimals is a pointer to an Animal means that `apcAnimals[iii]->Speak()` will call Animal::Speak() instead of the proper derived class version of Speak().

Although both of these techniques could save us a lot of time and energy, they have the same problem. The pointer or reference to the base class calls the base version of the function rather than the derived version. If only there was some way to make those base pointers call the derived version of a function instead of the base version…

Want to take a guess what virtual functions are for? :)

12.2 — Virtual functions

In the previous lesson on pointers and references to the base class of derived objects, we took a look at a number of examples where using pointers or references to a base class had the potential to simplify code. However, in every case, we ran up against the problem that the base pointer or reference was only able to call the base version of a function, not a derived version.

Here's a simple example of this behavior:

```
1  class Base
2  {
3  protected:
4
5  public:
6      const char* GetName() { return "Base"; }
7  };
8
9  class Derived: public Base
10 {
11 public:
12     const char* GetName() { return "Derived"; }
13 };
14
15 int main()
16 {
17     Derived cDerived;
18     Base &rBase = cDerived;
19     cout << "rBase is a " << rBase.GetName() << endl;
20 }
```

This example prints the result:

```
rBase is a Base
```

Because rBase is a Base pointer, it calls Base::GetName(), even though it's actually pointing to the Base portion of a Derived object.

In this lesson, we will address this issue using virtual functions.

Virtual functions

A **virtual function** is a special type of function that resolves to the most-derived version of the function with the same signature. To make a function virtual, simply place the "virtual" keyword before the function declaration.

Note that virtual functions and virtual base classes are two entirely different concepts, even though they share the same keyword.

Here's the above example with a virtual function:

```
1  class Base
2  {
```

```
protected:

public:
    virtual const char* GetName() { return "Base"; }
};

class Derived: public Base
{
public:
    virtual const char* GetName() { return "Derived"; }
};

int main()
{
    Derived cDerived;
    Base &rBase = &cDerived;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

This example prints the result:

```
rBase is a Derived
```

Because rBase is a pointer to the Base portion of a Derived object, when **rBase.GetName()** is evaluated, it would normally resolve to Base::GetName(). However, Base::GetName() is virtual, which tells the program to go look and see if there are any more-derived versions of the function available. Because the Base object that rBase is pointing to is actually part of a Derived object, the program will check every inherited class between Base and Derived and use the most-derived version of the function that it finds. In this case, that is Derived::GetName()!

Let's take a look at a slightly more complex example:

```
class A
{
public:
    virtual const char* GetName() { return "A"; }
};

class B: public A
{
public:
    virtual const char* GetName() { return "B"; }
};

class C: public B
{
public:
    virtual const char* GetName() { return "C"; }
};
```

```
class D: public C
{
public:
    virtual const char* GetName() { return "D"; }
};

int main()
{
    C cClass;
    A &rBase = cClass;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

What do you think this program will output?

Let's look at how this works. First, we instantiate a C class object. rBase is an A pointer, which we set to point to the A portion of the C object. Finally, we call rBase.GetName(). rBase.GetName() evaluates to A::GetName(). However, A::GetName() is virtual, so the compiler will check all the classes between A and C to see if it can find a more-derived match. First, it checks B::GetName(), and finds a match. Then it checks C::GetName() and finds a better match. It does not check D::GetName() because our original object was a C, not a D. Consequently, rBase.GetName() resolves to C::GetName().

As a result, our program outputs:

```
rBase is a C
```

A more complex example

Let's take another look at the Animal example we were working with in the previous lesson. Here's the original class:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
    const char* Speak() { return "???"; }

```

```
};

class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    const char* Speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    const char* Speak() { return "Woof"; }
};
```

And here's the class with virtual functions:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
    virtual const char* Speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Meow"; }
```

```
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Woof"; }
};
```

Note that we didn't make Animal::GetName() virtual. This is because GetName() is never overridden in any of the derived classes, therefore there is no need.

Now, using the virtual Speak() function, the following function should work correctly:

```
void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;
}

int main()
{
    Cat cCat("Fred");
    Dog cDog("Garbo");

    Report(cCat);
    Report(cDog);
}
```

Indeed, this program produces the result:

```
Fred says Meow
Garbo says Woof
```

When cAnimal.Speak() is evaluated, the program notes that it is a virtual function. In the case where rAnimal is pointing to the Animal portion of a Cat object, the program looks at all the classes between Animal and Cat to see if it can find a more derived function. In that case, it finds Cat::Speak(). In the case where rAnimal points to the Animal portion of a Dog object, the program resolves the function call to Dog::Speak().

Similarly, the following array example now works as expected:

```
Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");
Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");

// Set up an array of pointers to animals, and set those pointers to our
Cat and Dog objects
Animal *apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson,
```

```
&cZeke };
for (int iii=0; iii < 6; iii++)
    cout << apcAnimals[iii]->GetName() << " says " << apcAnimals[iii]->Speak() << endl;
```

Which produces the result:

```
Fred says Meow
Garbo says Woof
Pooky says Woof
Truffle says Woof
Tyson says Meow
Zeke says Meow
```

Even though these two examples only use Cat and Dog, any other classes we derive from Animal would also work with our Report() function and Animal array without further modification! This is perhaps the biggest benefit of virtual functions — the ability to structure your code in such a way that newly derived classes will automatically work with the old code without modification!

A word of warning: the signature of the derived class function must exactly match the signature of the base class virtual function in order for the derived class function to be used. If the derived class function has different parameter types, the program will likely still compile fine, but the virtual function will not resolve as intended.

Use of the virtual keyword

Technically, the virtual keyword is not needed in derived class. For example:

```
class Base
{
protected:

public:
    virtual const char* GetName() { return "Base"; }
};

class Derived: public Base
{
public:
    const char* GetName() { return "Derived"; } // note lack of virtual keyword
};

int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

prints

```
rBase is a Derived
```

Exactly the same as if Derived::GetName() was explicitly tagged as virtual. Only the most base class function needs to be tagged as virtual for all of the derived functions to work virtually. However, having the keyword virtual on the derived functions does not hurt, and it serves as a useful reminder that the function is a virtual function rather than a normal one. Consequently, it's generally a good idea to use the virtual keyword for virtualized functions in derived classes even though it's not strictly necessary.

Return types of virtual functions

Under normal circumstances, the return type of a virtual function and it's override must match. Thus, the following will not work:

```
1  class Base
2  {
3  public:
4      virtual int GetValue() { return 5; }
5  };
6
7  class Derived: public Base
8  {
9  public:
10     virtual double GetValue() { return 6.78; }
11 };
```

However, there is one special case in which this is not true. If the return type of a virtual function is a pointer or a reference to a class, override functions can return a pointer or a reference to a derived class. These are called covariant return types. Here is an example:

```
1  class Base
2  {
3  public:
4      // This version of GetThis() returns a pointer to a Base class
5      virtual Base* GetThis() { return this; }
6  };
7
8  class Derived: public Base
9  {
10     // Normally override functions have to return objects of the same type
   as the base function
11     // However, because Derived is derived from Base, it's okay to return
   Derived* instead of Base*
12     virtual Derived* GetThis() { return this; }
   };
```

Note that some older compilers (eg. Visual Studio 6) do not support covariant return types.

12.3 — Virtual destructors, virtual assignment, and overriding virtualization

Virtual destructors

Although C++ provides a default destructor for your classes if you do not provide one yourself, it is sometimes the case that you will want to provide your own destructor (particularly if the class needs to deallocate memory). You should **always** make your destructors virtual if you're dealing with inheritance. Consider the following example:

```
1  class Base
2  {
3  public:
4      ~Base()
5      {
6          cout << "Calling ~Base()" << endl;
7      }
8  };
9
10 class Derived: public Base
11 {
12 private:
13     int* m_pnArray;
14
15 public:
16     Derived(int nLength)
17     {
18         m_pnArray = new int[nLength];
19     }
20
21     ~Derived() // note: not virtual
22     {
23         cout << "Calling ~Derived()" << endl;
24         delete[] m_pnArray;
25     }
26 };
27
28 int main()
29 {
30     Derived *pDerived = new Derived(5);
31     Base *pBase = pDerived;
32     delete pBase;
33
34     return 0;
35 }
```

Because pBase is a Base pointer, when pBase is deleted, the program looks to see if the Base destructor is virtual. It's not, so it assumes it only needs to call the Base destructor. We can see this in the fact that the above example prints:

```
Calling ~Base()
```

However, we really want the delete function to call Derived's destructor (which will call Base's destructor in turn). We do this by making Base's destructor virtual:

```
1  class Base
2  {
3  public:
4      virtual ~Base()
5      {
6          cout << "Calling ~Base()" << endl;
7      }
8  };
9
10 class Derived: public Base
11 {
12 private:
13     int* m_pnArray;
14
15 public:
16     Derived(int nLength)
17     {
18         m_pnArray = new int[nLength];
19     }
20
21     virtual ~Derived()
22     {
23         cout << "Calling ~Derived()" << endl;
24         delete[] m_pnArray;
25     }
26 };
27
28 int main()
29 {
30     Derived *pDerived = new Derived(5);
31     Base *pBase = pDerived;
32     delete pBase;
33
34     return 0;
35 }
```

Now this program produces the following result:

```
Calling ~Derived()
Calling ~Base()
```

Again, whenever you are dealing with inheritance, you should make your destructors virtual.

Virtual assignment

It is possible to make the assignment operator virtual. However, unlike the destructor case where virtualization is always a good idea, virtualizing the assignment operator really opens up a bag full of worms and gets into some advanced topics outside of the scope of this tutorial. Consequently, we are going to recommend you leave your assignments non-virtual for now, in the interest of simplicity.

Overriding virtualization

Very rarely you may want to override the virtualization of a function. For example, consider the following code:

```
class Base
{
public:
    virtual const char* GetName() { return "Base"; }
};

class Derived: public Base
{
public
    virtual const char* GetName() { return "Derived"; }
};
```

There may be cases where you want a Base pointer to a Derived object to call Base::GetName() instead of Derived::GetName(). To do so, simply use the scope resolution operator:

```
int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    // Calls Base::GetName() instead of the virtualized Derived::GetName()
    cout << rBase.Base::GetName() << endl;
}
```

You probably won't use this very often, but it's good to know it's at least possible.

The downside of virtual functions

Since most of the time you'll want your functions to be virtual, why not just make all functions virtual? The answer is because it's inefficient — resolving a virtual function call takes longer than a resolving a regular one. Furthermore, the compiler also has to allocate an extra pointer for each class object that has one or more virtual functions. We'll talk about this more in the next couple of lessons.

12.4 — Early binding and late binding

In this chapter and the next, we are going to take a closer look at how virtual functions are implemented. While this information is not strictly necessary to effectively use virtual functions, it is interesting. Nevertheless, you can consider both sections optional reading.

When a C++ program is executed, it executes sequentially, beginning at the top of main(). When a function call is encountered, the point of execution jumps to the beginning of the function being called. How does the CPU know to do this?

When a program is compiled, the compiler converts each statement in your C++ program into one or more lines of machine language. Each line of machine language is given it's own unique sequential address. This is no different for functions — when a function is encountered, it is converted into machine language and given the next available address. Thus, each function ends up with a unique machine language address.

Binding refers to the process that is used to convert identifiers (such as variable and function names) into machine language addresses. Although binding is used for both variables and functions, in this lesson we're going to focus on function binding.

Early binding

Most of the function calls the compiler encounters will be direct function calls. A direct function call is a statement that directly calls a function. For example:

```
1  #include <iostream>
2
3  void PrintValue(int nValue)
4  {
5      std::cout << nValue;
6  }
7
8  int main()
9  {
10     PrintValue(5); // This is a direct function call
11     return 0;
12 }
```

Direct function calls can be resolved using a process known as early binding. **Early binding** (also called static binding) means the compiler is able to directly associate the identifier name (such as a function or variable name) with a machine address. Remember that all functions have a unique machine address. So when the compiler encounters a function call, it replaces the function call with a machine language instruction that tells the CPU to jump to the address of the function.

Let's take a look at a simple calculator program that uses early binding:

```
1  #include <iostream>
2  using namespace std;
3
4  int Add(int nX, int nY)
5  {
6      return nX + nY;
7  }
```

```
8  int Subtract(int nX, int nY)
9  {
10     return nX - nY;
11 }
12
13 int Multiply(int nX, int nY)
14 {
15     return nX * nY;
16 }
17
18 int main()
19 {
20     int nX;
21     cout << "Enter a number: ";
22     cin >> nX;
23
24     int nY;
25     cout << "Enter another number: ";
26     cin >> nY;
27
28     int nOperation;
29     do
30     {
31         cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
32         cin >> nOperation;
33     } while (nOperation < 0 || nOperation > 2);
34
35     int nResult = 0;
36     switch (nOperation)
37     {
38         case 0: nResult = Add(nX, nY); break;
39         case 1: nResult = Subtract(nX, nY); break;
40         case 2: nResult = Multiply(nX, nY); break;
41     }
42
43     cout << "The answer is: " << nResult << endl;
44
45     return 0;
46 }
```

Because Add(), Subtract(), and Multiply() are all direct function calls, the compiler will use early binding to resolve the Add(), Subtract(), and Multiply() function calls. The compiler will replace the Add() function call with an instruction that tells the CPU to jump to the address of the Add() function. The same holds true for for Subtract() and Multiply().

Late Binding

In some programs, it is not possible to know which function will be called until runtime (when the program is run). This is known as **late binding** (or dynamic binding). In C++, one way to get late binding is to use function pointers. To review function pointers briefly, a function pointer is a type of pointer that points to a function instead of a variable. The function that a function pointer points to can be called by using the function call operator (()) on the pointer.

For example, the following code calls the Add() function:

```
int Add(int nX, int nY)
{
    return nX + nY;
}

int main()
{
    // Create a function pointer and make it point to the Add function
    int (*pFcn)(int, int) = Add;
    cout << pFcn(5, 3) << endl; // add 5 + 3

    return 0;
}
```

Calling a function via a function pointer is also known as an indirect function call. The following calculator program is functionally identical to the calculator example above, except it uses a function pointer instead of a direct function call:

```
#include <iostream>
using namespace std;

int Add(int nX, int nY)
{
    return nX + nY;
}

int Subtract(int nX, int nY)
{
    return nX - nY;
}

int Multiply(int nX, int nY)
{
    return nX * nY;
}

int main()
{
    int nX;
    cout << "Enter a number: ";
    cin >> nX;

    int nY;
    cout << "Enter another number: ";
    cin >> nY;

    int nOperation;
    do
    {
        cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
        cin >> nOperation;
```

```
    } while (nOperation < 0 || nOperation > 2);

    // Create a function pointer named pFcn (yes, the syntax is ugly)
    int (*pFcn)(int, int);

    // Set pFcn to point to the function the user chose
    switch (nOperation)
    {
        case 0: pFcn = Add; break;
        case 1: pFcn = Subtract; break;
        case 2: pFcn = Multiply; break;
    }

    // Call the function that pFcn is pointing to with nX and nY as
parameters
    cout << "The answer is: " << pFcn(nX, nY) << endl;

    return 0;
}
```

In this example, instead of calling the Add(), Subtract(), or Multiply() function directly, we've instead set pFcn to point at the function we wish to call. Then we call the function through the pointer. The compiler is unable to use early binding to resolve the function call `pFcn(nX, nY)` because it can not tell which function pFcn will be pointing to at compile time!

Late binding is slightly less efficient since it involves an extra level of indirection. With early binding, the compiler can tell the CPU to jump directly to the function's address. With late binding, the program has to read the address held in the pointer and then jump to that address. This involves one extra step, making it slightly slower. However, the advantage of late binding is that it is more flexible than early binding, because decisions about what function to call do not need to be made until run time.

In the next lesson, we'll take a look at how late binding is used to implement virtual functions.

12.5 — The virtual table

To implement virtual functions, C++ uses a special form of late binding known as the virtual table. The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as "vtable", "virtual function table", "virtual method table", or "dispatch table".

Because knowing how the virtual table works is not necessary to use virtual functions, this section can be considered optional reading.

The virtual table is actually quite simple, though it's a little complex to describe in words. First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it's own virtual table. This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.

Second, the compiler also adds a hidden pointer to the base class, which we will call *__vptr. *__vptr is set (automatically) when a class instance is created so that it points to the virtual table for that class. Unlike the *this pointer, which is actually a function parameter used by the compiler to resolve self-references, *__vptr is a real pointer. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that *__vptr is inherited by derived classes, which is important.

By now, you're probably confused as to how these things all fit together, so let's take a look at a simple example:

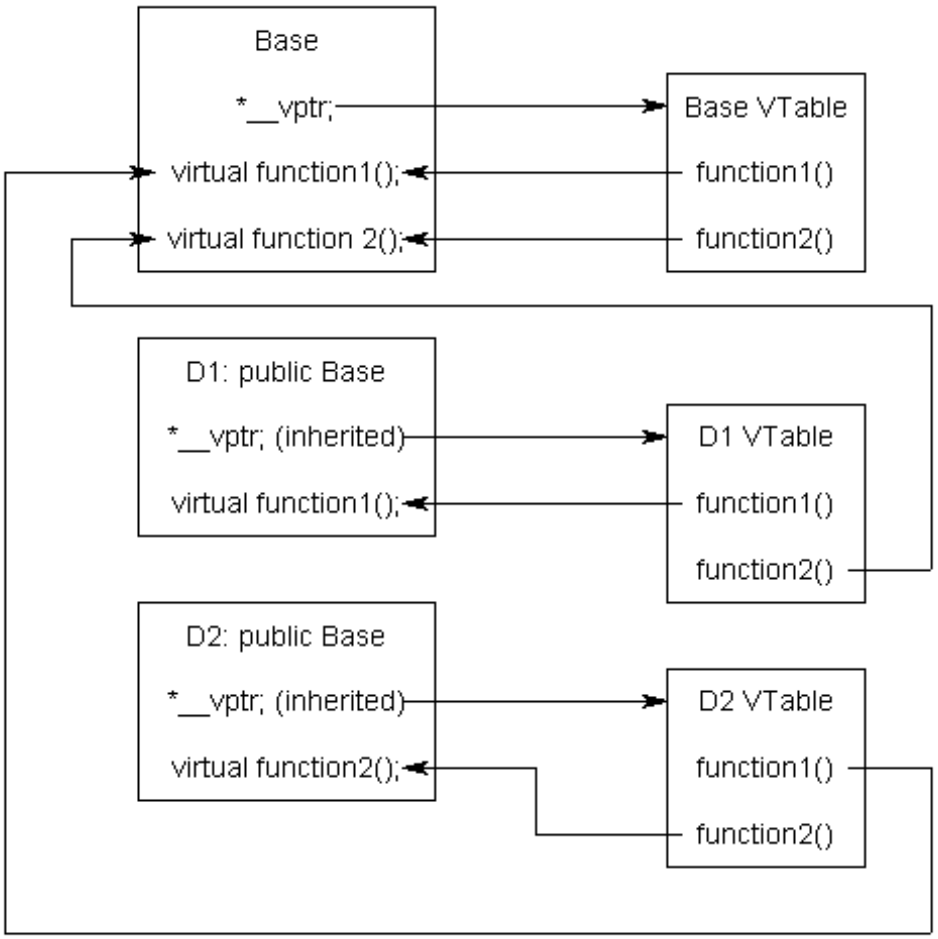
```
1  class Base
2  {
3  public:
4      virtual void function1() {};
5      virtual void function2() {};
6  };
7
8  class D1: public Base
9  {
10 public:
11     virtual void function1() {};
12 };
13
14 class D2: public Base
15 {
16 public:
17     virtual void function2() {};
18 };
```

Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.

The compiler also adds a hidden pointer to the most base class that uses virtual functions. Although the compiler does this automatically, we'll put it in the next example just to show where it's added:

```
1  class Base
2  {
3  public:
4      FunctionPointer *__vptr;
```

```
4       virtual void function1() {};
5       virtual void function2() {};
6   };
7
8   cla
9   {
10  pub
11
12  };
13
14  cla
15  {
16  pub

17
18  };
```



When a class. For example l table for Base. W irtual table for D1 o

Now, let two virtual function d one for function ed out with the most

Base's v Base. Base has no points to Base::fu

D1′s virt of both D1 and Base. However, D1 has overridden function1(), making D1::function1() more derived than Base::function1(). Consequently, the entry for function1 points to D1::function1(). D1 hasn't overridden function2(), so the entry for function2 will point to Base::function2().

D2′s virtual table is similar to D1, except the entry for function1 points to Base::function1(), and the entry for function2 points to D2::function2().

Here's a picture of this graphically:

Although this diagram is kind of crazy looking, it's really quite simple: the *__vptr in each class points to the virtual table for that class. The entries in the virtual table point to the most-derived version of the function objects of that class are allowed to call.

So consider what happens when we create an object of type D1:

```
1  int main()
2  {
3      D1 cClass;
   }
```

Because cClass is a D1 object, cClass has it's *__vptr set to the D1 virtual table.

Now, let's set a base pointer to D1:

```
1  int main()
2  {
3      D1 cClass;
4      Base *pClass = &cClass;
   }
```

Note that because pClass is a base pointer, it only points to the Base portion of cClass. However, also note that *__vptr is in the Base portion of the class, so pClass has access to this pointer. Finally, note that pClass->__vptr points to the D1 virtual table! Consequently, even though pClass is of type Base, it still has access to D1′s virtual table.

So what happens when we try to call pClass->function1()?

```
1  int main()
2  {
3      D1 cClass;
4      Base *pClass = &cClass;
5      pClass->function1();
   }
```

First, the program recognizes that function1() is a virtual function. Second, uses pClass->__vptr to get to D1′s virtual table. Third, it looks up which version of function1() to call in D1′s virtual table. This has been set to D1::function1(). Therefore, pClass->function1() resolves to D1::function1()!

Now, you might be saying, "But what if Base really pointed to a Base object instead of a D1 object. Would it still call D1::function1()?". The answer is no.

```
1  int main()
2  {
3      Base cClass;
4      Base *pClass = &cClass;
5      pClass->function1();
   }
```

In this case, when cClass is created, __vptr points to Base's virtual table, not D1′s virtual table. Consequently, pClass->__vptr will also be pointing to Base's virtual table. Base's virtual table entry for function1() points to Base::function1(). Thus, pClass->function1() resolves to Base::function1(), which is the most-derived version of function1() that a Base object should be able to call.

By using these tables, the compiler and program are able to ensure function calls resolve to the appropriate virtual function, even if you're only using a pointer or reference to a base class!

Calling a virtual function is slower than calling a non-virtual function for a couple of reasons: First, we have to use the *__vptr to get to the appropriate virtual table. Second, we have to index the virtual table to find the correct function to call. Only then can we call the function. As a result, we have to do 3 operations to find the function to call, as opposed to 2 operations for a normal indirect function call, or one operation for a direct function call. However, with modern computers, this added time is usually fairly insignificant.

12.6 — Pure virtual functions, abstract base classes, and interface classes

Finally, we arrive at the end of our long journey through inheritance! This is the last topic we will cover on the subject. So congratulations in advance on making it through the hardest part of the language!

Pure virtual (abstract) functions and abstract base classes

So far, all of the virtual functions we have written have a body (a definition). However, C++ allows you to create a special kind of virtual function called a **pure virtual function** (or **abstract function**) that has no body at all! A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.

To create a pure virtual function, rather than define a body for the function, we simply assign the function the value 0.

```
1  class Base
2  {
3  public:
4      const char* SayHi() { return "Hi"; } // a normal non-virtual function
5
6      virtual const char* GetName() { return "Base"; } // a normal virtual
   function
7
8      virtual int GetValue() = 0; // a pure virtual function
9  };
```

When we add a pure virtual function to our class, we are effectively saying, "it is up to the derived classes to implement this function".

Using a pure virtual function has two main consequences: First, any class with one or more pure virtual functions becomes an **abstract base class**, which means that it can not be instantiated! Consider what would happen if we could create an instance of Base:

```
1  int main()
2  {
3      Base cBase; // pretend this was legal
4      cBase.GetValue(); // what would this do?
   }
```

Second, any derived class must define a body for this function, or that derived class will be considered an abstract base class as well.

Let's take a look at an example of a pure virtual function in action. In a previous lesson, we wrote a simple Animal base class and derived a Cat and a Dog class from it. Here's the code as we left it:

```
1  #include <string>
2  class Animal
3  {
4  protected:
5      std::string m_strName;
6
7      // We're making this constructor protected because
       // we don't want people creating Animal objects directly,
```

```
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
    virtual const char* Speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Woof"; }
};
```

We've prevented people from allocating objects of type Animal by making the constructor protected. However, there's one problem that has not been addressed. It is still possible to create derived classes that do not redefine Speak(). For example:

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
        : Animal(strName)
    {
    }

    // We forgot to redefine Speak
};

int main()
{
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

This will print:

```
Betsy says ???
```

What happened? We forgot to redefine Speak, so cCow.Speak() resolved to Animal.Speak(), which isn't what we wanted.

A better solution to this problem is to use a pure virtual function:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

public:
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

    std::string GetName() { return m_strName; }
    virtual const char* Speak() = 0; // pure virtual function
};
```

There are a couple of things to note here. First, Speak() is now a pure virtual function. This means Animal is an abstract base class, and can not be instantiated. Consequently, we do not need to make the constructor protected any longer (though it doesn't hurt). Second, because our Cow class was derived from Animal, but we did not define Cow::Speak(), Cow is also an abstract base class. Now when we try to compile this code:

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
        : Animal(strName)
    {
    }

    // We forgot to redefine Speak
};

int main()
{
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

The compiler will give us a warning because Cow is an abstract base class and we can not create instances of abstract base classes:

```
C:\\Test.cpp(141) : error C2259: 'Cow' : cannot instantiate abstract class
due to following members:
        C:\Test.cpp(128) : see declaration of 'Cow'
C:\\Test.cpp(141) : warning C4259: 'const char *__thiscall
Animal::Speak(void)' : pure virtual function was not defined
```

This tells us that we will only be able to instantiate Cow if Cow provides a body for Speak().

Let's go ahead and do that:

```
1  class Cow: public Animal
2  {
3  public:
4      Cow(std::string strName)
5          : Animal(strName)
6      {
7      }
8
9      virtual const char* Speak() { return "Moo"; }
10 };
11
12 int main()
13 {
14     Cow cCow("Betsy");
15     cout << cCow.GetName() << " says " << cCow.Speak() << endl;
16 }
```

Now this program will compile and print:

```
Betsy says Moo
```

A pure virtual function is useful when we have a function that we want to put in the base class, but only the derived classes know what it should return. A pure virtual function makes it so the base class can not be instantiated, and the derived classes are forced to define these function before they can be instantiated. This helps ensure the derived classes do not forget to redefine functions that the base class was expecting them to.

Interface classes

An **interface class** is a class that has no members variables, and where all of the functions are pure virtual! In other words, the class is purely a definition, and has no actual implementation. Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

Interface classes are often named beginning with an I. Here's a sample interface class:

```
1  class IErrorLog
2  {
3      virtual bool OpenLog(const char *strFilename) = 0;
```

```
4       virtual bool CloseLog() = 0;
5
6       virtual bool WriteError(const char *strErrorMessage) = 0;
    };
```

Any class inheriting from IErrorLog must provide implementations for all three functions in order to be instantiated. You could derive a class named FileErrorLog, where OpenLog() opens a file on disk, CloseLog() closes it, and WriteError() writes the message to the file. You could derive another class called ScreenErrorLog, where OpenLog() and CloseLog() do nothing, and WriteError() prints the message in a pop-up message box on the screen.

Now, let's say you need to write some code that uses an error log. If you write your code so it includes FileErrorLog or ScreenErrorLog directly, then you're effectively stuck using that kind of error log. For example, the following function effectively forces callers of MySqrt() to use a FileErrorLog, which may or may not be what they want.

```
1   double MySqrt(double dValue, FileErrorLog &cLog)
2   {
3       if (dValue < 0.0)
4       {
5           cLog.WriteError("Tried to take square root of value less than 0");
6           return 0.0;
7       }
8       else
            return dValue;
    }
```

A much better way to implement this function is to use IErrorLog instead:

```
1   double MySqrt(double dValue, IErrorLog &cLog)
2   {
3       if (dValue < 0.0)
4       {
5           cLog.WriteError("Tried to take square root of value less than 0");
6           return 0.0;
7       }
8       else
            return dValue;
    }
```

Now the caller can pass in any class that conforms to the IErrorLog interface. If they want the error to go to a file, they can pass in an instance of FileErrorLog. If they want it to go to the screen, they can pass in an instance of ScreenErrorLog. Or if they want to do something you haven't even thought of, such as sending an email to someone when there's an error, they can derive a new class from IErrorLog (eg. EmailErrorLog) and use an instance of that! By using IErrorLog, your function becomes more independent and flexible.

Interface classes have become extremely popular because they are easy to use, easy to extend, and easy to maintain. In fact, some modern languages, such as Java and C#, have added an "interface" keyword that allows programmers to directly define an interface class without having

to explicitly mark all of the member functions as abstract. Furthermore, although Java and C# will not let you use multiple inheritance on normal classes, they will let you multiply inherit as many interfaces as you like. Because interfaces have no data and no function bodies, they avoid a lot of the traditional problems with multiple inheritance while still providing much of the flexibility.

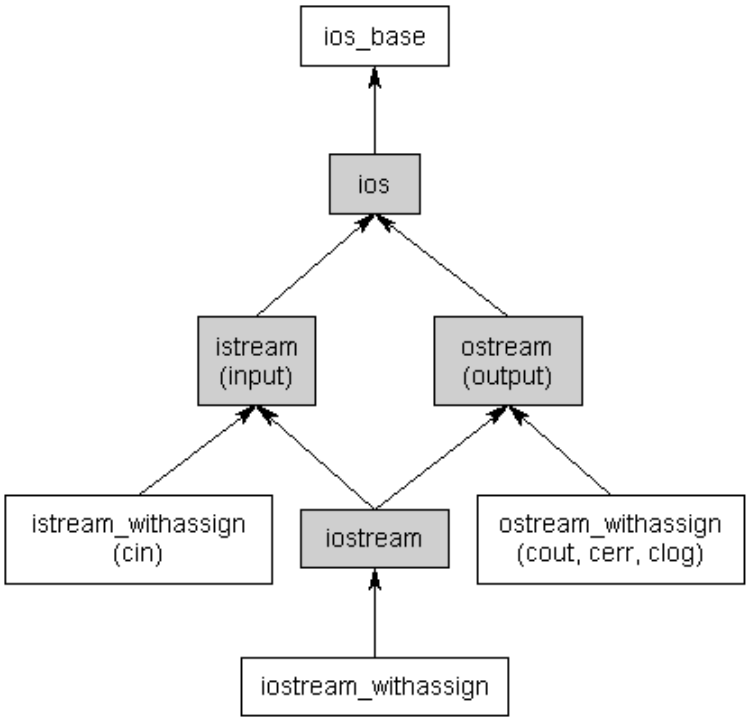
Input and output (I/O)

13.1 — Input and output (I/O) streams

Input and output functionality is not defined as part of the core C++ language, but rather is provided through the C++ standard library (and thus resides in the std namespace). In previous lessons, you included the iostream library header and made use of the cin and cout objects to do simple I/O. In this lesson, we'll take a look at the iostream library in more detail.

The iostream library

When you include the iostream header, you gain access to a whole heirarchy of classes responsible for providing I/O functionality (including one class that is actually named iostream). The class heirarchy for the non-file-I/O classes looks like this:



The first thing you may notice about this hierarchy is that it uses multiple inheritance (that thing we told you to avoid if at all possible). However, the iostream library has been designed and extensively tested in order to avoid any of the typical multiple inheritance problems, so you can use it freely without worrying.

Streams

The second thing you may notice is that the word "stream" is used an awful lot. At it's most basic, I/O in C++ is implemented with streams. Abstractly, a **stream** can be thought of as a

sequence of bytes of infinite length that is used as a buffer to hold data that is waiting to be processed.

Typically we deal with two different types of streams. **Input streams** are used to hold input from a data producer, such as a keyboard, a file, or a network. For example, the user may press a key on the keyboard while the program is currently not expecting any input. Rather than ignore the users keypress, the data is put into an input stream, where it will wait until the program is ready for it.

Conversely, **output streams** are used to hold output for a particular data consumer, such as a monitor, a file, or a printer. When writing data to an output device, the device may not be ready to accept that data yet — for example, the printer may still be warming up when the program writes data to it's output stream. The data will sit in the output stream until the printer begins consuming it.

Some devices, such as files and networks, are capable of being both input and output sources.

The nice thing about streams is the programmer only has to learn how to interact with the streams in order to read and write data to many different kinds of devices. The details about how the stream interfaces with the actual devices they are hooked up to is left up to the environment or operating system.

Input/output in C++

Although the ios class is generally derived from ios_base, ios is typically the most base class you will be working directly with. The ios class defines a bunch of stuff that is common to both input and output streams. We'll deal with this stuff in a future lesson.

The **istream** class is the primary class used when dealing with input streams. With input streams, the **extraction operator (>>)** is used to remove values from the stream. This makes sense: when the user presses a key on the keyboard, the key code is placed in an input stream. Your program then extracts the value from the stream so it can be used.

The **ostream** class is the primary class used when dealing with output streams. With output streams, the **insertion operator (<<)** is used to put values in the stream. This also makes sense: you insert your values into the stream, and the data consumer (eg. monitor) uses them.

The **iostream** class can handle both input and output, allowing bidirectional I/O.

Finally, there are a bunch of classes that end in "_withassign". These stream classes are derived from istream, ostream, and iostream (respectively) with an assignment operator defined, allowing you to assign one stream to another. In most cases, you won't be dealing with these classes directly.

Standard streams in C++

A **standard stream** is a pre-connected stream provided to a computer program by it's environment. C++ comes with four predefined standard stream objects that have already been set up for your use. The first two, you have seen before:

1. **cin** — an istream_withassign class tied to the standard input (typically the keyboard)
2. **cout** — an ostream_withassign class tied to the standard output (typically the monitor)
3. **cerr** — an ostream_withassign class tied to the standard error (typically the monitor), providing unbuffered output
4. **clog** — an ostream_withassign class tied to the standard error (typically the monitor), providing buffered output

Unbuffered output is typically handled immediately, whereas buffered output is typically stored and written out as a block. Because clog isn't used very often, it is often omitted from the list of standard streams.

A basic example

Here's an example of input and output using the standard streams:

```
1  #include <iostream>
2
3  int main()
4  {
5      using namespace std;
6      // First we'll use the insertion operator on cout to print text to the
   monitor
7      cout << "Enter your age: " << endl;
8
9      // Then we'll use the extraction operator on cin to get input from the
   user
10     int nAge;
11     cin >> nAge;
12
13     if (nAge <= 0)
14     {
15         // In this case we'll use the insertion operatior on cerr to print
   an error message
16         cerr << "Oops, you entered an invalid age!" << endl;
17         exit(1);
18     }
19
20     // Otherwise we'll use insertion again on cout to print a result
21     cout << "You entered " << nAge << " years old" << endl;
22
23     return 0;
   }
```

In the next lesson, we'll take a look at some more I/O related functionality in more detail.

13.2 — Input with istream

The iostream library is fairly complex — so we will not be able to cover it in it's entirety in these tutorials. However, we will show you the most commonly used functionality. In this section, we will look at various aspects of the input class (istream).

Note: All of the I/O functionality in this lesson lives in the std namespace. That means all I/O objects and functions either have to be prefixed with "std::", or the "using namespace std;" statement has to be used.

The extraction operator

As seen in many lessons now, we can use the extraction operator (>>) to read information from an input stream. C++ has predefined extraction operations for all of the built-in data types, and you've already seen how you can overload the extraction operator for your own classes.

When reading strings, one common problem with the extraction operator is how to keep the input from overflowing your buffer. Given the following example:

```
1 char buf[10];
2 cin >> buf;
```

what happens if the user enters 18 characters? The buffer overflows, and bad stuff happens. Generally speaking, it's a bad idea to make any assumption about how many characters your user will enter.

One way to handle this problem is through use of manipulators. A **manipulator** is an object that is used to modify a stream when applied with the extraction (>>) or insertion (<<) operators. One manipulator you have already worked with extensively is "endl", which both prints a newline character and flushes any buffered output.

C++ provides a manipulator known as **setw** (in the iomanip.h header) that can be used to limit the number of characters read in from a stream. To use setw(), simply provide the maximum number of characters to read as a parameter, and insert it into your input statement like such:

```
1 #include <iomanip.h>
2 char buf[10];
3 cin >> setw(10) >> buf;
```

This program will now only read the first 9 characters out of the stream (leaving room for a terminator). Any remaining characters will be left in the stream until the next extraction.

Extraction and whitespace

The one thing that we have omitted to mention so far is that the extraction operator works with "formatted" data — that is, it skips whitespace (blanks, tabs, and newlines).

Take a look at the following program:

```
1  int main()
2  {
3      char ch;
4      while (cin >> ch)
5          cout << ch;
6
7      return 0;
8  }
```

When the user inputs the following:

```
Hello my name is Alex
```

The insertion operator skips the spaces and the newline. Consequently, the output is:

```
HellomynameisAlex
```

Oftentimes, you'll want to get user input but not discard whitespace. To do this, the istream class provides many functions that can be used for this purpose.

One of the most useful is the **get()** function, which simply gets a character from the input stream. Here's the same program as above using get():

```
1  int main()
2  {
3      char ch;
4      while (cin.get(ch))
5          cout << ch;
6
7      return 0;
8  }
```

Now when we use the input:

```
Hello my name is Alex
```

The output is:

```
Hello my name is Alex
```

get() also has a string version that takes a maximum number of characters to read:

```
1  int main()
2  {
```

```
3       char strBuf[11];
4       cin.get(strBuf, 11);
5       cout << strBuf << endl;
6
7       return 0;
    }
```

If we input:

```
Hello my name is Alex
```

The output is:

```
Hello my n
```

Note that we only read the first 10 characters (we had to leave one character for a terminator). The remaining characters were left in the input stream.

One important thing to note about get() is that it does not read in a newline character! This can cause some unexpected results:

```
1   int main()
2   {
3       char strBuf[11];
4       // Read up to 10 characters
5       cin.get(strBuf, 11);
6       cout << strBuf << endl;
7
8       // Read up to 10 more characters
9       cin.get(strBuf, 11);
10      cout << strBuf << endl;
11      return 0;
12  }
```

If the user enters:

```
Hello!
```

The program will print:

```
Hello!
```

and then terminate! Why didn't it ask for 10 more characters? The answer is because the first get() read up to the newline and then stopped. The second get() saw there was still input in the cin stream and tried to read it. But the first character was the newline, so it stopped immediately.

Consequently, there is another function called **getline()** that works exactly like get() but reads the newline as well.

```
1   int main()
```

```
{
    char strBuf[11];
    // Read up to 10 characters
    cin.getline(strBuf, 11);
    cout << strBuf << endl;

    // Read up to 10 more characters
    cin.getline(strBuf, 11);
    cout << strBuf << endl;
    return 0;
}
```

This code will perform as you expect, even if the user enters a string with a newline in it.

If you need to know how many character were extracted by the last call of getline(), use **gcount()**:

```
int main()
{
    char strBuf[100];
    cin.getline(strBuf, 100);
    cout << strBuf << endl;
    cout << cin.gcount() << " characters were read" << endl;

    return 0;
}
```

A special version of getline() for std::string

There is a special version of getline() that lives outside the istream class that is used for reading in variables of type std::string. This special version is not a member of either ostream nor istream, and is included in the string header. Here is an example of it's use:

```
#include <string>
#include <iostream>

int main()
{
    using namespace std;
    string strBuf;
    getline(cin, strBuf);
    cout << strBuf << endl;

    return 0;
}
```

A few more useful istream functions

There are a few more useful input functions that you might want to make use of:

ignore() discards the first character in the stream.
ignore(int nCount) discards the first nCount characters.
peek() allows you to read a character from the stream without removing it from the stream.
unget() returns the last character read back into the stream so it can be read again by the next call.
putback(char ch) allows you to put a character of your choice back into the stream to be read by the next call.

istream contains many other functions and variants of the above mentioned functions that may be useful, depending on what you need to do. However, those topics are really more suited for a tutorial or book focusing on the standard library (such as the excellent "The C++ Standard Template Library" by Nicolai M. Josuttis).

13.3 — Output with ostream and ios

In this section, we will look at various aspects of the iostream output class (ostream).

Note: All of the I/O functionality in this lesson lives in the std namespace. That means all I/O objects and functions either have to be prefixed with "std::", or the "using namespace std;" statement has to be used.

The insertion operator

The insertion operator (<<) is used to put information into an output stream. C++ has predefined insertion operations for all of the built-in data types, and you've already seen how you can overload the insertion operator for your own classes.

In the lesson on streams, you saw that both istream and ostream were derived from a class called ios. One of the jobs of ios (and ios_base) is to control the formatting options for output.

Formatting

There are two ways to change the formatting options: flags, and manipulators. You can think of **flags** as boolean variables that can be turned on and off. **Manipulators** are objects placed in a stream that affect the way things are input and output.

To switch a flag on, use the **setf()** function, with the appropriate flag as a parameter. For example, by default, C++ does not print a + sign in front of positive numbers. However, by using the ios::showpos flag, we can change this behavior:

```
1 cout.setf(ios::showpos); // turn on the ios::showpos flag
2 cout << 27 << endl;
```

This results in the following output:

```
+27
```

It is possible to turn on multiple ios flags at once using the OR (|) operator:

```
1 cout.setf(ios::showpos | ios::uppercase); // turn on the ios::showpos and
2 ios::uppercase flag
  cout << 27 << endl;
```

To turn a flag off, use the **unsetf()** function:

```
1 cout.setf(ios::showpos); // turn on the ios::showpos flag
2 cout << 27 << endl;
```

```
3 cout.unsetf(ios::showpos); // turn off the ios::showpos flag
4 cout << 28 << endl;
```

This results in the following output:

```
+27
28
```

There's one other bit of trickiness when using setf() that needs to be mentioned. Many flags belong to groups, called format groups. A **format group** is a group of flags that perform similar (sometimes mutually exclusive) formatting options. For example, a format group named "basefield" contains the flags "oct", "dec", and "hex", which controls the base of integral values. By default, the "dec" flag is set. Consequently, if we do this:

```
1 cout.setf(ios::hex); // try to turn on hex output
2 cout << 27 << endl;
```

We get the following output:

```
27
```

It didn't work! The reason why is because setf() only turns flags on — it isn't smart enough to turn mutually exclusive flags off. Consequently, when we turned ios::hex on, ios::dec was still on, and ios::dec apparently takes precedence. There are two ways to get around this problem.

First, we can turn off ios::dec so that only ios::hex is set:

```
1 cout.unsetf(ios::dec); // turn off decimal output
2 cout.setf(ios::hex); // turn on hexadecimal output
3 cout << 27 << endl;
```

Now we get output as expected:

```
1b
```

The second way is to use a different form of setf() that takes two parameters: the first parameter is the flag to set, and the second is the formatting group it belongs to. When using this form of setf(), all of the flags belonging to the group are turned off, and only the flag passed in is turned on. For example:

```
1 // Turn on ios::hex as the only ios::basefield flag
2 cout.setf(ios::hex, ios::basefield);
3 cout << 27 << endl;
```

This also produces the expected output:

```
1b
```

Using setf() and unsetf() tends to be awkward, so C++ provides a second way to change the formatting options: manipulators. The nice thing about manipulators is that they are smart enough to turn on and off the appropriate flags. Here is an example of using some manipulators to change the base:

```
1 cout << hex << 27 << endl; // print 27 in hex
2 cout << 28 << endl; // we're still in hex
3 cout << dec << 29 << endl; // back to decimal
```

This program produces the output:

```
1b
1c
29
```

In general, using manipulators is much easier than setting and unsetting flags. Many options are available via both flags and manipulators (such as changing the base), however, other options are only available via flags or via manipulators, so it's important to know how to use both.

Useful formatters

Here is a list of some of the more useful flags, manipulators, and member functions. Flags live in the ios class, manipulators lives in the std namespace, and the member functions live in the ostream class.

Group	Flag	Meaning
	boolalpha	If set, booleans print "true" or "false". If not set, booleans print 0 or 1

Manipulator	Meaning
boolalpha	Booleans print "true" or "false"
noboolalpha	Booleans print 0 or 1 (default)

Example:

```
1 cout << true << " " << false << endl;
2
3 cout.setf(ios::boolalpha);
4 cout << true << " " << false << endl;
5
6 cout << noboolalpha << true << " " << false << endl;
7
  cout << boolalpha << true << " " << false << endl;
```

Result:

```
0 1
true false
```

```
0 1
true false
```

Group	Flag	Meaning
	showpos	If set, prefix positive numbers with a +

Manipulator	Meaning
showpos	Prefixes positive numbers with a +
noshowpos	Doesn't prefix positive numbers with a +

Example:

```
1  cout << 5 << endl;
2
3  cout.setf(ios::showpos);
4  cout << 5 << endl;
5
6  cout << noshowpos << 5 << endl;
7
   cout << showpos << 5 << endl;
```

Result:

```
5
+5
5
+5
```

Group	Flag	Meaning
	uppercase	If set, uses upper case letters

Manipulator	Meaning
uppercase	Uses upper case letters
nouppercase	Uses lower case letters

Example:

```
1  cout << 12345678.9 << endl;
2
3  cout.setf(ios::uppercase);
4  cout << 12345678.9 << endl;
5
6  cout << nouppercase << 12345678.9<< endl;
7
   cout << uppercase << 12345678.9 << endl;
```

Result:

```
1.23457e+007
1.23457E+007
1.23457e+007
1.23457E+007
```

Group	Flag	Meaning
basefield	dec	Prints values in decimal (default)
basefield	hex	Prints values in hexadecimal
basefield	oct	Prints values in octal
basefield	(none)	Prints values according to leading characters of value

Manipulator	Meaning
dec	Prints values in decimal
hex	Prints values in hexadecimal
oct	Prints values in octal

Example:

```
1  cout << 27 << endl;
2
3  cout.setf(ios::dec, ios::basefield);
4  cout << 27 << endl;
5
6  cout.setf(ios::oct, ios::basefield);
7  cout << 27 << endl;
8
9  cout.setf(ios::hex, ios::basefield);
10 cout << 27 << endl;
11
12 cout << dec << 27 << endl;
   cout << oct << 27 << endl;
   cout << hex << 27 << endl;
```

Result:

```
27
27
33
1b
27
33
1b
```

By now, you should be able to see the relationship between setting formatting via flag and via manipulators. In future examples, we will use manipulators unless they are not available.

Precision, notation, and decimal points

Using manipulators (or flags), it is possible to change the precision and format with which floating point numbers are displayed. There are several formatting options that combine in somewhat complex ways, so we will take a closer look at this.

Group	Flag	Meaning
floatfield	fixed	Uses decimal notation for floating-point numbers
floatfield	scientific	Uses scientific notation for floating-point numbers
floatfield	(none)	Uses fixed for numbers with few digits, scientific otherwise
floatfield	showpoint	Always show a decimal point and trailing 0′s for floating-point values

Manipulator	Meaning
fixed	Use decimal notation for values
scientific	Use scientific notation for values
showpoint	Show a decimal point and trailing 0′s for floating-point values
noshowpoint	Don’t show a decimal point and trailing 0′s for floating-point values
setprecision(int)	Sets the precision of floating-point numbers (defined in iomanip.h)

Member function	Meaning
precision()	Returns the current precision of floating-point numbers
precision(int)	Sets the precision of floating-point numbers and returns old precision

If fixed or scientific notation is used, precision determines how many decimal places in the fraction is displayed. Note that if the precision is less than the number of significant digits, the number will be rounded.

```
1  cout << fixed << endl;
2  cout << setprecision(3) << 123.456 << endl;
3  cout << setprecision(4) << 123.456 << endl;
4  cout << setprecision(5) << 123.456 << endl;
5  cout << setprecision(6) << 123.456 << endl;
6  cout << setprecision(7) << 123.456 << endl;
7
8  cout << scientific << endl;
9  cout << setprecision(3) << 123.456 << endl;
10 cout << setprecision(4) << 123.456 << endl;
11 cout << setprecision(5) << 123.456 << endl;
   cout << setprecision(6) << 123.456 << endl;
   cout << setprecision(7) << 123.456 << endl;
```

Produces the result:

```
123.456
123.4560
123.45600
123.456000
```

```
123.4560000

1.235e+002
1.2346e+002
1.23456e+002
1.234560e+002
1.2345600e+002
```

If neither fixed nor scientific are being used, precision determines how many significant digits should be displayed. Again, if the precision is less than the number of significant digits, the number will be rounded.

```
1  cout << setprecision(3) << 123.456 << endl;
2  cout << setprecision(4) << 123.456 << endl;
3  cout << setprecision(5) << 123.456 << endl;
4  cout << setprecision(6) << 123.456 << endl;
5  cout << setprecision(7) << 123.456 << endl;
```

Produces the following result:

```
123
123.5
123.46
123.456
123.456
```

Using the showpoint manipulator or flag, you can make the stream write a decimal point and trailing zeros.

```
1  cout << showpoint << endl;
2  cout << setprecision(3) << 123.456 << endl;
3  cout << setprecision(4) << 123.456 << endl;
4  cout << setprecision(5) << 123.456 << endl;
5  cout << setprecision(6) << 123.456 << endl;
6  cout << setprecision(7) << 123.456 << endl;
```

Produces the following result:

```
123.
123.5
123.46
123.456
123.4560
```

Here's a summary table with some more examples:

Option	**Precision**	**12345.0**	**0.12345**
Normal	3	1.23e+004	0.123

	4	1.235e+004	0.1235
	5	12345	0.12345
	6	12345	0.12345
Showpoint	3	1.23e+004	0.123
	4	1.235e+004	0.1235
	5	12345.	0.12345
	6	12345.0	0.123450
Fixed	3	12345.000	0.123
	4	12345.0000	0.1235
	5	12345.00000	0.12345
	6	12345.000000	0.123450
Scientific	3	1.235e+004	1.235e-001
	4	1.2345e+004	1.2345e-001
	5	1.23450e+004	1.23450e-001
	6	1.234500e+004	1.234500e-001

Width, fill characters, and justification

Typically when you print numbers, the numbers are printed without any regard to the space around them. However, it is possible to left or right justify the printing of numbers. In order to do this, we have to first define a field width, which defines the number of output spaces a value will have. If the actual number printed is smaller than the field width, it will be left or right justified (as specified). If the actual number is larger than the field width, it will not be truncated — it will overflow the field.

Group	**Flag**	**Meaning**
adjustfield	internal	Left-justifies the sign of the number, and right-justifies the value
adjustfield	left	Left-justifies the sign and value
adjustfield	right	Right-justifies the sign and value (default)

Manipulator	**Meaning**
internal	Left-justifies the sign of the number, and right-justifies the value
left	Left-justifies the sign and value
right	Right-justifies the sign and value
setfill(char)	Sets the parameter as the fill character (defined in iomanip.h)
setw(int)	Sets the field width for input and output to the parameter (defined in iomanip.h)

Member function	Meaning
fill()	Returns the current fill character
fill(char)	Sets the fill character and returns the old fill character
width()	Returns the current field width
width(int)	Sets the current field width and returns old field width

In order to use any of these formatters, we first have to set a field width. This can be done via the width(int) member function, or the setw() manipulator. Note that right justification is the default.

```
1 cout << -12345 << endl; // print default value with no field width
2 cout << setw(10) << -12345 << endl; // print default with field width
3 cout << setw(10) << left << -12345 << endl; // print left justified
4 cout << setw(10) << right << -12345 << endl; // print right justified
5 cout << setw(10) << internal << -12345 << endl; // print internally
  justified
```

This produces the result:

```
-12345
    -12345
-12345
    -12345
-    12345
```

One thing to note is that setw() and width() only affect the next output statement. They are not persistent like some other flags/manipulators.

Now, let's set a fill character and do the same example:

```
1 cout.fill('*');
2 cout << -12345 << endl; // print default value with no field width
3 cout << setw(10) << -12345 << endl; // print default with field width
4 cout << setw(10) << left << -12345 << endl; // print left justified
5 cout << setw(10) << right << -12345 << endl; // print right justified
6 cout << setw(10) << internal << -12345 << endl; // print internally
  justified
```

This produces the output:

```
-12345
****-12345
-12345****
****-12345
-****12345
```

Note that all the blank spaces in the field have been filled up with the fill character.

The ostream class and iostream library contain other output functions, flags, and manipulators that may be useful, depending on what you need to do. As with the istream class, those topics are really more suited for a tutorial or book focusing on the standard library (such as the excellent "The C++ Standard Template Library" by Nicolai M. Josuttis).

13.4 — Stream classes for strings

So far, all of the I/O examples you have seen have been writing to cout or reading from cin. However, there is another set of classes called the stream classes for strings that allow you to use the familiar insertions (<<) and extraction (>>) operators to work with strings. Like istream and ostream, the string streams provide a buffer to hold data. However, unlike cin and cout, these streams are not connected to an I/O channel (such as a keyboard, monitor, etc…). One of the primary uses of string streams is to buffer output for display at a later time, or to process input line-by-line.

There are six string classes for streams: istringstream (derived from istream), ostringstream (derived from ostream), and stringstream (derived from iostream) are used for reading and writing normal characters width strings. wistringstream, wostringstream, and wstringstream are used for reading and writing wide character strings. To use the stringstreams, you need to #include the sstream header.

There are two ways to get data into a stringstream:

1) Use the insertion (<<) operator:

```
1 stringstream os;
2 os << "en garde!" << endl; // insert "en garde!" into the stringstream
```

2) Use the str(string) function to set the value of the buffer:

```
1 stringstream os;
2 os.str("en garde!"); // set the stringstream buffer to "en garde!"
```

There are similarly two ways to get data out of a stringstream:

1) Use the str() function to retrieve the results of the buffer:

```
1 stringstream os;
2 os << "12345 67.89" << endl;
3 cout << os.str();
```

This prints:

```
12345 67.89
```

2) Use the extraction (>>) operator:

```
1 stringstream os;
```

```
2  os << "12345 67.89"; // insert a string of numbers into the stream
3
4  string strValue;
5  os >> strValue;
6
7  string strValue2;
8  os >> strValue2;
9
10 // print the numbers separated by a dash
   cout << strValue << " - " << strValue2 << endl;
```

This program prints:

```
12345 - 67.89
```

Note that the >> operator iterates through the string -- each successive use of >> returns the next extractable value in the stream. On the other hand, str() returns the whole value of the stream, even if the >> has already been used on the stream.

Conversion between strings and numbers

Because the insertion and extraction operators know how to work with all of the basic data types, we can use them in order to convert strings to numbers or vice versa.

First, let's take a look at converting numbers into a string:

```
1  stringstream os;
2
3  int nValue = 12345;
4  double dValue = 67.89;
5  os << nValue << " " << dValue;
6
7  string strValue1, strValue2;
8  os >> strValue1 >> strValue2;
9
   cout << strValue1 << " " << strValue2 << endl;
```

This snippet prints:

```
12345 67.89
```

Now let's convert a numerical string to a number:

```
1  stringstream os;
2  os << "12345 67.89"; // insert a string of numbers into the stream
3  int nValue;
4  double dValue;
5
   os >> nValue >> dValue;
```

```
6
7  cout << nValue << " " << dValue << endl;
```

This program prints:

```
12345 67.89
```

Clearing a stringstream for reuse

There are several ways to empty a stringstream's buffer.

1) Set it to the empty string using str():

```
1  stringstream os;
2  os << "Hello ";
3
4  os.str(""); // erase the buffer
5
6  os << "World!";
   cout << os.str();
```

2) Call erase() on the result of str():

```
1  stringstream os;
2  os << "Hello ";
3
4  os.str(std::string()); // erase the buffer
5
6  os << "World!";
   cout << os.str();
```

Both of these programs produce the following result:

```
World!
```

When clearing out a stringstream, it is also generally a good idea to call the clear() function:

```
1  stringstream os;
2  os << "Hello ";
3
4  os.str(""); // erase the buffer
5  os.clear(); // reset error flags
6
7  os << "World!";
   cout << os.str();
```

clear() resets any error flags that may have been set and returns the stream back to the ok state. We will talk more about the stream state and error flags in the next lesson.

13.5 — Stream states and input validation

Stream states

The ios_base class contains several state flags that are used to signal various conditions that may occur when using streams:

Flag	Meaning
goodbit	Everything is okay
badbit	Some kind of fatal error occurred (eg. the program tried read past the end of a file)
eofbit	The stream has reached the end of a file
failbit	A non-fatal error occurred (eg. the user entered letters when the program was expecting an integer)

Although these flags live in ios_base, because ios is derived from ios_base and ios takes less typing than ios_base, they are generally accessed through ios (eg. as std::ios::failbit).

ios_base also provides a number of member functions in order to conveniently access these states:

Member function	Meaning
good()	Returns true if the goodbit is set (the stream is ok)
bad()	Returns true if the badbit is set (a fatal error occurred)
eof()	Returns true if the eofbit is set (the stream is at the end of a file)
fail()	Returns true if the failbit is set (a non-fatal error occurred)
clear()	Clears all flags and restores the stream to the goodbit state
clear(state)	Clears all flags and sets the state flag passed in
rdstate()	Returns the currently set flags
setstate(state)	Sets the state flag passed in

The most commonly dealt with bit is the failbit, which is set when the user enters invalid input. For example, consider the following program:

```
1 cout << "Enter your age: ";
2 int nAge;
3 cin >> nAge;
```

Note that this program is expecting the user to enter an integer. However, if the user enters non-numeric data, such as "Alex", cin will be unable to extract anything to nAge, and the failbit will be set.

If an error occurs and a stream is set to anything other than goodbit, further stream operations on that stream will be ignored. This condition can be cleared by calling the clear() function.

Input validation

Input validation is the process of checking whether the user input meets some set of criteria. Input validation can generally be broken down into two types: string and numeric.

With string validation, we accept all user input as a string, and then accept or reject that string depending on whether it is formatted appropriately. For example, if we ask the user to enter a telephone number, we may want to ensure the data they enter has ten digits. In most languages (especially scripting languages like Perl and PHP), this is done via regular expressions. However, C++ does not have built-in regular expression support (it's supposedly coming with the next revision of C++), so typically this is done by examining each character of the string to make sure it meets some criteria.

With numerical validation, we are typically concerned with making sure the number the user enters is within a particular range (eg. between 0 and 20). However, unlike with string validation, it's possible for the user to enter things that aren't numbers at all — and we need to handle these cases too.

To help us out, C++ provides a number of useful functions that we can use to determine whether specific characters are numbers or letters. The following functions live in the cctype header:

Function	Meaning
isalnum(int)	Returns non-zero if the parameter is a letter or a digit
isalpha(int)	Returns non-zero if the parameter is a letter
iscntrl(int)	Returns non-zero if the parameter is a control character
isdigit(int)	Returns non-zero if the parameter is a digit
isgraph(int)	Returns non-zero if the parameter is printable character that is not whitespace
isprint(int)	Returns non-zero if the parameter is printable character (including whitespace)
ispunct(int)	Returns non-zero if the parameter is neither alphanumeric nor whitespace
isspace(int)	Returns non-zero if the parameter is whitespace
isxdigit(int)	Returns non-zero if the parameter is a hexadecimal digit (0-9, a-f, A-F)

String validation

Let's do a simple case of string validation by asking the user to enter their name. Our validation criteria will be that the user enters only alphabetic characters or spaces. If anything else is encountered, the input will be rejected.

When it comes to variable length inputs, the best way to validate strings (besides using a regular expression library) is to step through each character of the string and ensure it meets the validation criteria. That's exactly what we'll do here.

```
1  #include <cctype>
2  #include <string>
3  #include <iostream>
4  using namespace std;
5
6  while (1)
7  {
8      // Get user's name
9      cout << "Enter your name: ";
10     string strName;
11     getline(cin, strName); // get the entire line, including spaces
12
13     bool bRejected=false; // has strName been rejected?
14
15     // Step through each character in the string until we either hit
16     // the end of the string, or we rejected a character
17     for (unsigned int nIndex=0; nIndex < strName.length() && !bRejected;
   nIndex++)
18     {
19         // If the current character is an alpha character, that's fine
20         if (isalpha(strName[nIndex]))
21             continue;
22
23         // If it's a space, that's fine too
24         if (strName[nIndex]==' ')
25             continue;
26
27         // Otherwise we're rejecting this input
28         bRejected = true;
29     }
30
31     // If the input has been accepted, exit the while loop
32     // otherwise we're going to loop again
33     if (!bRejected)
34         break;
35 }
```

Note that this code isn't perfect: the user could say their name was "asf w jweo s di we ao" or some other bit of gibberish, or even worse, just a bunch of spaces. We could address this somewhat by refining our validation criteria to only accept strings that contain at least one character and at most one space.

Now let's take a look at another example where we are going to ask the user to enter their phone number. Unlike a user's name, which is variable-length and where the validation criteria are the same for every character, a phone number is a fixed length but the validation criteria differ depending on the position of the character. Consequently, we are going to take a different approach to validating our phone number input. In this case, we're going to write a function that will check the user's input against a predetermined template to see whether it matches. The template will work as follows:

A # will match any digit in the user input.
A @ will match any alphabetic character in the user input.
A _ will match any whitespace.
A ? will match anything.
Otherwise, the characters in the user input and the template must match exactly.

So, if we ask the function to match the template "(###) ###-####", that means we expect the user to enter a '(' character, three numbers, a ')' character, a space, three numbers, a dash, and four more numbers. If any of these things doesn't match, the input will be rejected.

Here is the code:

```
1  bool InputMatches(string strUserInput, string strTemplate)
2  {
3      if (strTemplate.length() != strUserInput.length())
4          return false;
5
6      // Step through the user input to see if it matches
7      for (unsigned int nIndex=0; nIndex < strTemplate.length(); nIndex++)
8      {
9          switch (strTemplate[nIndex])
10         {
11             case '#': // match a digit
12                 if (!isdigit(strUserInput[nIndex]))
13                     return false;
14                 break;
15             case '_': // match a whitespace
16                 if (!isspace(strUserInput[nIndex]))
17                     return false;
18                 break;
19             case '@': // match a letter
20                 if (!isalpha(strUserInput[nIndex]))
21                     return false;
22                 break;
23             case '?': // match anything
24                 break;
25             default: // match the exact character
26                 if (strUserInput[nIndex] != strTemplate[nIndex])
                       return false;
         }
     }

     return true;
}
```

```
27
28  int main()
29  {
30      string strValue;
31
32      while (1)
33      {
34          cout << "Enter a phone number (###) ###-####: ";
35          getline(cin, strValue); // get the entire line, including spaces
36          if (InputMatches(strValue, "(###) ###-####"))
37              break;
38      }
39
40      cout << "You entered: " << strValue << endl;
41  }
```

Using this function, we can force the user to match our specific format exactly. However, this function is still subject to several constraints: if #, @, _, and ? are valid characters in the user input, this function won't work, because those symbols have been given special meanings. Also, unlike with regular expressions, there is no template symbol that means "a variable number of characters can be entered". Thus, such a template could not be used to ensure the user enters two words separated by a whitespace, because it can not handle the fact that the words are of variable lengths. For such problems, the non-template approach is generally more appropriate.

Numeric validation

When dealing with numeric input, the obvious way to proceed is to use the extraction operator to extract input to a numeric type. By checking the failbit, we can then tell whether the user entered a number or not.

Let's try this approach:

```
1   int main()
2   {
3       int nAge;
4
5       while (1)
6       {
7           cout << "Enter your age: ";
8           cin >> nAge;
9
10          if (cin.fail()) // no extraction took place
11          {
12              cin.clear(); // reset the state bits back to goodbit so we can
13  use ignore()
14              cin.ignore(1000, '\n'); // clear out the bad input from the
15  stream
16              continue; // try again
17          }
18
19          if (nAge <= 0) // make sure nAge is positive
```

```
            continue;

    break;
    }

    cout << "You entered: " << nAge << endl;
}
```

If the user enters a number, cin.fail() will be false, and we will hit the break statement, exiting the loop. If the user enters input starting with a letter, cin.fail() will be true, and we will go into the conditional.

However, there's one more case we haven't tested for, and that's when the user enters a string that starts with numbers but then contains letters (eg. "34abcd56″). In this case, the starting numbers (34) will be extracted into nAge, the remainder of the string ("abcd56″) will be left in the input stream, and the failbit will NOT be set. This causes two potential problems:

1) If you want this to be valid input, you now have garbage in your stream.
2) If you don't want this to be valid input, it is not rejected (and you have garbage in your stream).

Let's fix the first problem. This is easy:

```
int main()
{
    int nAge;

    while (1)
    {
        cout << "Enter your age: ";
        cin >> nAge;

        if (cin.fail()) // no extraction took place
        {
            cin.clear(); // reset the state bits back to goodbit so we can use ignore()
            cin.ignore(1000, '\n'); // clear out the bad input from the stream
            continue; // try again
        }

        cin.ignore(1000, '\n'); // clear out any additional input from the stream

        if (nAge <= 0) // make sure nAge is positive
            continue;

    break;
    }

    cout << "You entered: " << nAge << endl;
```

```
}
```

If you don't want such input to be valid, we'll have to do a little extra work. Fortunately, the previous solution gets us half way there. We can use the gcount() function to determine how many characters were ignored. If our input was valid, gcount() should return 1 (the newline character that was discarded). If it returns more than 1, the user entered something that wasn't extracted properly, and we should ask them for new input. Here's an example of this:

```
int main()
{
    int nAge;

    while (1)
    {
        cout << "Enter your age: ";
        cin >> nAge;

        if (cin.fail()) // no extraction took place
        {
            cin.clear(); // reset the state bits back to goodbit so we can use ignore()
            cin.ignore(1000, '\n'); // clear out the bad input from the stream
            continue; // try again
        }

        cin.ignore(1000, '\n'); // clear out any additional input from the stream
        if (cin.gcount() > 1) // if we cleared out more than one additional character
            continue; // we'll consider this input to be invalid

        if (nAge <= 0) // make sure nAge is positive
            continue;

    break;
    }

    cout << "You entered: " << nAge << endl;
}
```

Numeric validation as a string

The above example was quite a bit of work simply to get a simple value! Another way to process numeric input is to read it in as a string, process it as a string, and if it passes the validation, convert it to a numeric type. The following program makes use of that methodology:

```
int main()
{
    int nAge;

    while (1)
```

```
    {
        cout << "Enter your age: ";
        string strAge;
        cin >> strAge;

        // Check to make sure each character is a digit
        bool bValid = true;
        for (unsigned int nIndex=0; nIndex < strAge.length(); nIndex++)
            if (!isdigit(strAge[nIndex]))
            {
                bValid = false;
                break;
            }
        if (!bValid)
            continue;

        // At this point, we have something that can be converted to a number
        // So we'll use stringstream to do that conversion
        stringstream strStream;
        strStream << strAge;
        strStream >> nAge;

        if (nAge <= 0) // make sure nAge is positive
            continue;

    break;
    }

    cout << "You entered: " << nAge << endl;
}
```

Whether this approach is more or less work than straight numeric extraction depends on your validation parameters and restrictions.

As you can see, doing input validation in C++ is a lot of work. Fortunately, many such tasks (eg. doing numeric validation as a string) can be easily turned into functions that can be reused in a wide variety of situations.

13.6 — Basic file I/O

File I/O in C++ works very similarly to normal I/O (with a few minor added complexities). There are 3 basic file I/O classes in C++: ifstream (derived from istream), ofstream (derived from ostream), and fstream (derived from iostream). These classes do file input, output, and input/output respectively. To use the file I/O classes, you will need to include the fstream.h header.

Unlike the cout, cin, cerr, and clog streams, which are already ready for use, file streams have to be explicitly set up by the programmer. However, this is extremely simple: to open a file for reading and/or writing, simply instantiate an object of the appropriate file I/O class, with the name of the file as a parameter. Then use the insertion (<<) or extraction (>>) operator to read/write to the file. Once you are done, there are several ways to close a file: explicitly call the close() function, or just let the file I/O variable go out of scope (the file I/O class destructor will close the file for you).

File output

To do file output in the following example, we're going to use the ofstream class. This is extremely straighforward:

```
1  #include <fstream>
2  #include <iostream>
3
4  int main()
5  {
6      using namespace std;
7
8      // ofstream is used for writing files
9      // We'll make a file called Sample.dat
10     ofstream outf("Sample.dat");
11
12     // If we couldn't open the output file stream for writing
13     if (!outf)
14     {
15         // Print an error and exit
16         cerr << "Uh oh, Sample.dat could not be opened for writing!" <<
17 endl;
18         exit(1);
19     }
20
21     // We'll write two lines into this file
22     outf << "This is line 1" << endl;
23     outf << "This is line 2" << endl;
24
25     return 0;
26
```

```
    // When outf goes out of scope, the ofstream
    // destructor will close the file
}
```

If you look in your project directory, you should see a file called Sample.dat. If you open it with a text editor, you will see that it indeed contains two lines we wrote to the file.

Note that it is also possible to use the put() function to write a single character to the file.

File input

Now, we'll take the file we wrote in the last example and read it back in from disk. Note that ifstream returns a 0 if we've reached the end of the file (EOF). We'll use this fact to determine how much to read.

```
#include <fstream>
#include <iostream>
#include <string>

int main()
{
    using namespace std;

    // ifstream is used for reading files
    // We'll read from a file called Sample.dat
    ifstream inf("Sample.dat");

    // If we couldn't open the output file stream for reading
    if (!inf)
    {
        // Print an error and exit
        cerr << "Uh oh, Sample.dat could not be opened for reading!" << endl;
        exit(1);
    }

    // While there's still stuff left to read
    while (inf)
    {
        // read stuff from the file into a string and print it
        std::string strInput;
        inf >> strInput;
        cout << strInput << endl;
    }

    return 0;

    // When inf goes out of scope, the ifstream
    // destructor will close the file
}
```

This produces the result:

```
This
is
line
1
This
is
line
2
```

Hmmm, that wasn't quite what we wanted. Remember that the extraction operator deals with "formatted output", and breaks on whitespace. In order to read in entire lines, we'll have to use the getline() function.

```
1  #include <fstream>
2  #include <iostream>
3  #include <string>
4
5  int main()
6  {
7      using namespace std;
8
9      // ifstream is used for reading files
10     // We'll read from a file called Sample.dat
11     ifstream inf("Sample.dat");
12
13     // If we couldn't open the input file stream for reading
14     if (!inf)
15     {
16         // Print an error and exit
17         cerr << "Uh oh, Sample.dat could not be opened for reading!" <<
18 endl;
19         exit(1);
20     }
21
22     // While there's still stuff left to read
23     while (inf)
24     {
25         // read stuff from the file into a string and print it
26         std::string strInput;
27         getline(inf, strInput);
28         cout << strInput << endl;
29     }
30
31     return 0;
32
33     // When inf goes out of scope, the ifstream
34     // destructor will close the file
35 }
```

This produces the result:

```
This is line 1
This is line 2
```

Buffered output

Output in C++ may be buffered. This means that anything that is output to a file stream may not be written to disk immediately. Instead, several output operations may be batched and handled together. This is done primarily for performance reasons. When a buffer is written to disk, this is called **flushing** the buffer. One way to cause the buffer to be flushed is to close the file — the contents of the buffer will be flushed to disk, and then the file will be closed.

Buffering is usually not a problem, but in certain circumstance it can cause complications for the unwary. The main culprit in this case is when there is data in the buffer, and then program terminates immediately (either by crashing, or by calling exit()). In these cases, the destructors for the file stream classes are not executed, which means the files are never closed, which means the buffers are never flushed. In this case, the data in the buffer is not written to disk, and is lost forever. This is why it is always a good idea to explicitly close any open files before calling exit().

It is possible to flush the buffer manually using the ostream flush() function. Calling flush() can be useful to ensure the contents of the buffer are written to disk immediately, just in case the program crashes.

File modes

What happens if we try to write to a file that already exists? Running the output example again shows that the original file is completely overwritten each time the program is run. What if, instead, we wanted to append some more data to the end of the file? It turns out that the file stream constructors take an optional second parameter that allows you to specify information about how the file should be opened. This parameter is called mode, and the valid flags that it accepts live in the Ios class.

Ios file mode	Meaning
app	Opens the file in append mode
ate	Seeks to the end of the file before reading/writing
binary	Opens the file in binary mode (instead of text mode)
in	Opens the file in read mode (default for ifstream)
nocreate	Opens the file only if it already exists
noreplace	Opens the file only if it does not already exist
out	Opens the file in write mode (default for ofstream)
trunc	Erases the file if it already exists

It is possible to specify multiple flags by bitwise ORing them together (using the | operator). Note that ios::in and ios::out are already defaults for the ifstream and ofstream classes respectively. If you opt to use fstream (which can do both input and output), you explicitly have to pass in ios::in and/or ios::out depending on which mode you'd like to use.

Let's write a program that appends two more lines to the Sample.dat file we previously created:

```
1  int main()
2  {
3      using namespace std;
4
5      // We'll pass the ios:app flag to tell the ofstream to append
6      // rather than rewrite the file.  We do not need to pass in ios::out
7      // because ofstream defaults to ios::out
8      ofstream outf("Sample.dat", ios::app);
9
10     // If we couldn't open the output file stream for writing
11     if (!outf)
12     {
13         // Print an error and exit
14         cerr << "Uh oh, Sample.dat could not be opened for writing!" <<
15 endl;
16         exit(1);
17     }
18
19     outf << "This is line 3" << endl;
20     outf << "This is line 4" << endl;
21
22     return 0;
23
24     // When outf goes out of scope, the ofstream
25     // destructor will close the file
26 }
```

Now if we take a look at Sample.dat (using one of the above sample programs that prints its contents, or loading it in a text editor), we will see the following:

```
This is line 1
This is line 2
This is line 3
This is line 4
```

Explicitly opening files using open()

Just like it is possible to explicitly close a file stream using close(), it's also possible to explicitly open a file stream using open(). open() works just like the file stream constructors — it takes a file name and an optional file mode.

For example:

```
1  ofstream outf("Sample.dat");
```

```
2  outf << "This is line 1" << endl;
3  outf << "This is line 2" << endl;
4  outf.close(); // explicitly close the file
5
6  // Oops, we forgot something
7  outf.open("Sample.dat", ios::app);
8  outf << "This is line 3" << endl;
   outf.close();
```

13.7 — Random file I/O

The file pointer

Each file stream class contains a file pointer that is used to keep track of the current read/write position within the file. When something is read from or written to a file, the reading/writing happens at the file pointer's current location. By default, when opening a file for reading or writing, the file pointer is set to the beginning of the file. However, if a file is opened in append mode, the file pointer is moved to the end of the file, so that writing does not overwrite any of the current contents of the file.

Random file access with seekg() and seekp()

So far, all of the file access we've done has been sequential — that is, we've read or written the file contents in order. However, it is also possible to do random file access — that is, skip around to various points in the file to read it's contents. This can be useful when your file is full of records, and you wish to retrieve a specific record. Rather than reading all of the records until you get to the one you want, you can skip directly to the record you wish to retrieve.

Random file access is done by manipulating the file pointer using the seekg() function (for input) and seekp() function (for output). In case you are wondering, the g stands for "get" and the p for "put".

The seekg() and seekp() functions take two parameters. The first parameter is an offset that determines how many bytes to move the file pointer. The second parameter is an Ios flag that specifies what the offset parameter should be offset from.

Ios seek flag	Meaning
beg	The offset is relative to the beginning of the file (default)
cur	The offset is relative to the current location of the file pointer
end	The offset is relative to the end of the file

A positive offset means move the file pointer towards the end of the file, whereas a negative offset means move the file pointer towards the beginning of the file.

Here are some examples:

```
1 inf.seekg(14, ios::cur); // move forward 14 bytes
2 inf.seekg(-18, ios::cur); // move backwards 18 bytes
3 inf.seekg(22, ios::beg); // move to 22nd byte in file
4 inf.seekg(24); // move to 24th byte in file
  inf.seekg(-28, ios::end); // move to the 28th byte before end of the file
```

Moving to the beginning or end of the file is easy:

```
1  inf.seekg(0, ios::beg); // move to beginning of file
2  inf.seekg(0, ios::end); // move to end of file
```

Let's do an example using seekg() and the input file we created in the last lesson. That input file looks like this:

```
This is line 1
This is line 2
This is line 3
This is line 4
```

Heres the example:

```
1  int main()
2  {
3      using namespace std;
4
5      ifstream inf("Sample.dat");
6
7      // If we couldn't open the input file stream for reading
8      if (!inf)
9      {
10         // Print an error and exit
           cerr << "Uh oh, Sample.dat could not be opened for reading!" <<
   endl;
11         exit(1);
12     }
13
14     string strData;
15
16     inf.seekg(5); // move to 5th character
17     // Get the rest of the line and print it
18     getline(inf, strData);
19     cout << strData << endl;
20
21     inf.seekg(8, ios::cur); // move 8 more bytes into file
22     // Get rest of the line and print it
23     getline(inf, strData);
24     cout << strData << endl;
25
26     inf.seekg(-15, ios::end); // move 15 bytes before end of file
27     // Get rest of the line and print it
28     getline(inf, strData);
29     cout << strData << endl;

       return 0;
   }
```

This produces the result:

```
is line 1
line 2
his is line 4
```

Two other useful functions are tellg() and tellp(), which return the absolute position of the file pointer. This can be used to determine the size of a file:

```
1  ifstream inf("Sample.dat");
2  inf.seekg(0, ios::end); // move to end of file
3  cout << inf.tellg();
```

This prints:

```
64
```

which is how long sample.dat is in bytes (assuming a carriage return after the last line).

Reading and writing a file at the same time using fstream

The fstream class is capable of both reading and writing a file at the same time — almost! The big caveat here is that it is not possible to switch between reading and writing arbitrarily. Once a read or write has taken place, the only way to switch between the two is to perform a seek operation. If you don't actually want to move the file pointer, you can always seek to the current position:

```
1  iofile.seekg(iofile.tellg(), ios::beg); // seek to current file position
```

If you do not do this, any number of strange and bizarre things may occur.

(Note: Although it may seem that `inf.seekg(0, ios::cur)` would also work, it appears some compilers may optimize this away.)

One other bit of trickiness: Unlike istream, where we could say `while (inf)` to determine if there was more to read, this will not work with fstream.

Let's do a file I/O example using fstream. We're going to write a program that opens a file, reads it's contents, and changes the any vowels it finds to a '#' symbol.

```
1  int main()
2  {
3      using namespace std;
4
5      // Note we have to specify both in and out because we're using fstream
6      fstream iofile("Sample.dat", ios::in | ios::out);
7
8      // If we couldn't open iofile, print an error
9      if (!iofile)
```

```
10          // Print an error and exit
11          cerr << "Uh oh, Sample.dat could not be opened!" << endl;
12          exit(1);
13      }
14
15      char chChar; // we're going to do this character by character
16
17      // While there's still data to process
18      while (iofile.get(chChar))
19      {
20          switch (chChar)
21          {
22              // If we find a vowel
23              case 'a':
24              case 'e':
25              case 'i':
26              case 'o':
27              case 'u':
28              case 'A':
29              case 'E':
30              case 'I':
31              case 'O':
32              case 'U':
33
34                  // Back up one character
35                  iofile.seekg(-1, ios::cur);
36
37                  // Because we did a seek, we can now safely do a write, so
38                  // let's write a # over the vowel
39                  iofile << '#';
40
41                  // Now we want to go back to read mode so the next call
42                  // to get() will perform correctly.  We'll seekg() to the
43  current
44                  //  location  because  we  don't  want  to  move  the  file
45  pointer.
46                  iofile.seekg(iofile.tellg(), ios::beg);
47
48                  break;
49          }
50      }
51
52      return 0;
53  }
```

Other useful file functions

To delete a file, simply use the remove() function.

Also, the is_open() function will return true if the stream is currently open, and false otherwise.

A warning about writing pointers to disk

While streaming variables to a file is quite easy, things become more complicated when you're dealing with pointers. Remember that a pointer simply holds the address of the variable it is pointing to. Although it is possible to read and write addresses to disk, it is extremely dangerous to do so. This is because a variable's address may differ from execution to execution. Consequently, although a variable may have lived at address 0x0012FF7C when you wrote that address to disk, it may not live there any more when you read that address back in!

For example, let's say you had an integer named nValue that lived at address 0x0012FF7C. You assigned nValue the value 5. You also declared a pointer named *pnValue that points to nValue. pnValue holds nValue's address of 0x0012FF7C. You want to save these for later, so you write the value 5 and the address 0x0012FF7C to disk.

A few weeks later, you run the program again and read these values back from disk. You read the value 5 into another variable named nValue, which lives at 0x0012FF78. You read the address 0x0012FF7C into a new pointer named *pnValue. Because pnValue now points to 0x0012FF7C when the nValue lives at 0x0012FF78, pnValue is no longer pointing to nValue, and trying to access pnValue will lead you into trouble.

Rule: Do not write addresses to files. The variables that were originally at those addresses may be at different addresses when you read their values back in from disk, and the addresses will be invalid.

Templates

14.1 — Function templates

The need for function templates

In previous chapters, you've learned how to write functions and classes that help make programs easier to write, safer, and more maintainable. While functions and classes are powerful and flexible tools for effective programming, in certain cases they can also be somewhat limiting because of C++'s requirement that you specify the type of all parameters.

For example, let's say you wanted to write a function to calculate the maximum of two numbers. You might do so like this:

```
1  int max(int nX, int nY)
2  {
3      return (nX > nY) ? nX : nY;
4  }
```

This function would work great — for integers. What happens later when you realize your max() function needs to work with doubles? Traditionally, the answer would be to overload the max() function and create a new version that works with doubles:

```
1  double max(double dX, double dY)
2  {
3      return (dX > dY) ? dX : dY;
4  }
```

Note that the code for the implementation of the double version of maximum() is exactly the same as for the int version of max()! In fact, this implementation would work for all sorts of different types: chars, ints, doubles, and if you've overloaded the > operator, even classes! However, because C++ requires you to make your variables specific types, you're stuck writing one function for each type you wish to use.

Having to specify different "flavors" of the same function where the only thing that changes is the type of the parameters can become a severe maintenance headache and time-waster, and it also violates the general programming guideline that duplicate code should be minimized as much as possible. Wouldn't it be nice if we could write one version of max() that was able to work with parameters of ANY type?

This is where function templates come in!

What is a function template?

If you were to look up the word "template" in the dictionary, you'd find a definition that was similar to the following: "a template is a model that serves as a pattern for creating similar objects". One type of template that is very easy to understand is that of a stencil. A stencil is an object (eg. a piece of cardboard) with a shape cut out of it (eg. the letter J). By placing the stencil on top of another object, then spraying paint through the hole, you can very quickly produce stenciled patterns in many different colors! Note that you only need to create a given stencil once — you can then use it as many times as you like to create stenciled patterns in whatever color(s) you like. Even better, you don't have to decide the color of the stenciled pattern you want to create until you decide to actually use the stencil.

In C++, **function templates** are functions that serve as a pattern for creating other similar functions. The basic idea behind function templates is to create a function without having to specify the exact type(s) of some or all of the variables. Instead, we define the function using placeholder types, called **template type parameters**. Once we have created a function using these placeholder types, we have effectively created a "function stencil".

It turns out that you can't call a function template directly — this is because the compiler doesn't know how to handle placeholder types directly. Instead, when you call a template function, the compiler "stencils" out a copy of the template, replacing the placeholder types with the actual variable types in your function call! Using this methodology, the compiler can create multiple "flavors" of a function from one template! We'll take a look at this process in more detail in the next lesson.

Creating function templates in C++

At this point, you're probably wondering how to actually create function templates in C++. It turns out, it's not all that difficult.

Let's take a look at the int version of max() again:

```
1  int max(int nX, int nY)
2  {
3      return (nX > nY) ? nX : nY;
4  }
```

Note that there are 3 places where specific types are used: parameters nX, nY, and the return value all specify that they must be integers. To create a function template, we're going to replace these specific types with placeholder types. In this case, because we have only one type that needs replacing (int), we only need one template type parameter. Let's call our this placeholder type "Type". You can name your placeholder types almost anything you want, so long as it's not a reserved word. Here's our new function with a placeholder type:

```
1  Type max(Type tX, Type tY)
2  {
3      return (tX > tY) ? tX : tY;
4  }
```

(Note: I also changed the Hungarian notation on the variables to reflect that they are not necessarily integers any longer — they are whatever type Type is!)

This is a good start — however, it won't compile because the compiler doesn't know what "Type" means! In order to tell the compiler that Type is meant to be a placeholder type, we need to formally tell the compiler that Type is a template type parameter. This is done using what is called a **template parameter declaration**:

```
1  template <typename Type> // this is the template parameter declaration
2  Type max(Type tX, Type tY)
3  {
4      return (tX > tY) ? tX : tY;
5  }
```

Believe it or not, we're done! This will compile!

Now, let's take a slightly closer look at the template parameter declaration. We start with the keyword *template* — this tells the compiler that what follows is going to be a list of template parameters. We place all of our parameters inside angled brackets (<>). To create a template type parameter, use either the keyword *typename* or *class*. There is no difference between the two keywords in this context, and you will usually see people use the class keyword. However, we prefer the newer typename keyword, because it makes it clearer that the template type parameter doesn't have to be a class. After the typename or class keyword, all that's left is to pick a name for your placeholder type. Traditionally, with function that have only one template type parameter, the name "Type" (often shortened to "T") is used. If the template function uses multiple template type parameter, they can be separated by commas:

```
1  template <typename T1, typename T2>
2  // template function here
```

Using function templates

Using a function template is extremely straightforward — you can use it just like any other function:

```
1  int nValue = max(3, 7); // returns 7
2  double dValue = max(6.34, 18.523); // returns 18.523
3  char chValue = max('a', '6'); // returns 'a'
```

Note that all three of these calls to max() have parameters of different types!

As you can see, template functions can save a lot of time, because you only need to write one function, and it will work with many different types. Once you get used to writing function templates, you'll find they actually don't take any longer to write than functions with actual types. Template functions reduce code maintenance, because duplicate code is reduced

significantly. And finally, template functions can be safer, because there is no need to copy functions and change types by hand whenever you need the function to work with a new type!

Template functions do have a few drawbacks, and we would be remiss not to mention them. First, older compilers generally do not have very good template support. However, modern compilers are much better at supporting and implementing template functionality properly. Second, template functions produce crazy-looking error messages that are much harder to decipher than those of regular functions. However, these drawbacks are fairly minor compared with the power and flexibility templates bring to your programming toolkit!

Note: The standard library already comes with a templated max() function. If you use the statement “using namespace std;” the compiler will be unable to tell whether you want your version of max() or std::max().

14.2 — Function template instances

Function template instances

It's worth taking a brief look at how template functions are implemented in C++, because future lessons will build off of some of these concepts. It turns out that C++ does not compile the template function directly. Instead, at compile time, when the compiler encounters a call to a template function, it replicates the template function and replaces the template type parameters with actual types! The function with actual types is called a **function template instance**.

Let's take a look at an example of this process. First, we have a templated function:

```
1  template <typename Type> // this is the template parameter declaration
2  Type max(Type tX, Type tY)
3  {
4      return (tX > tY) ? tX : tY;
   }
```

When compiling your program, the compiler encounters a call to the templated function:

```
1  int nValue = max(3, 7); // calls max(int, int)
```

The compiler says, "oh, we want to call max(int, int)". The compiler replicates the function template and creates the template instance max(int, int):

```
1  int max(int tX, int tY)
2  {
3      return (tX > tY) ? tX : tY;
4  }
```

This is now a "normal function" that can be compiled into machine language.

Now, let's say later in your code, you called max() again using a different type:

```
1  double dValue = max(6.34, 18.523); // calls max(double, double)
```

C++ automatically creates a template instance for max(double, double):

```
1  double max(double tX, double tY)
2  {
3      return (tX > tY) ? tX : tY;
4  }
```

and then compiles it into machine language.

It's worth noting that the compiler is smart enough to know it only needs to create one template instance per set of unique type parameters. It's also worth noting that if you create a template function but do not call it, no template instances will be created.

Operators, function calls, and function templates

Template functions will work with both built-in types (eg. char, int, double, etc…) and classes, with one caveat. When the compiler compiles the template instance, it compiles it just like a normal function. In a normal function, any operators or function calls that you use with your types must be defined, or you will get a compiler error. Similarly, any operators or function calls in your template function must be defined for any types the function template is instantiated for. Let's take a look at this in more detail.

First, we'll create a simple class:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5  public:
6      Cents(int nCents)
7          : m_nCents(nCents)
8      {
9      }
10 };
```

Now, let's see what happens when we try to call our templated max() function with the Cents class:

```
1  Cents cNickle(5);
2  Cents cDime(10);
3
4  Cents cBigger = max(cNickle, cDime);
```

C++ will create a template instance for max() that looks like this:

```
1  Cents max(Cents tX, Cents tY)
2  {
3      return (tX > tY) ? tX : tY;
4  }
```

And then it will try to compile this function. See the problem here? C++ has no idea how to evaluate `tX > tY`! Consequently, this will produce a compile error.

To get around this problem, simply overload the > operator for any class we wish to use max() with:

```
1  class Cents
2  {
```

```
private:
    int m_nCents;
public:
    Cents(int nCents)
        : m_nCents(nCents)
    {
    }

    friend bool operator>(Cents &c1, Cents&c2)
    {
        return (c1.m_nCents > c2.m_nCents) ? true: false;
    }
};
```

Now C++ will know how to compare `tX > tY` when tX and tY are objects of the Cents class! As a result, our max() function will now work with two objects of type Cents.

Another example

Let's do one more example of a function template. The following function template will calculate the average of a number of objects in an array:

```
template <class T>
T Average(T *atArray, int nNumValues)
{
    T tSum = 0;
    for (int nCount=0; nCount < nNumValues; nCount++)
        tSum += atArray[nCount];

    tSum /= nNumValues;
    return tSum;
}
```

Now let's see it in action:

```
int anArray[] = { 5, 3, 2, 1, 4 };
cout << Average(anArray, 5) << endl;

double dnArray[] = { 3.12, 3.45, 9.23, 6.34 };
cout << Average(dnArray, 4) << endl;
```

This produces the values:

```
3
5.535
```

As you can see, it works great for built-in types!

Now let's see what happens when we call this function on our Cents class:

```
Cents cArray[] = { Cents(5), Cents(10), Cents(15), Cents(14) };
```

```
2  cout << Average(cArray, 4) << endl;
```

The compiler goes berserk and produces a ton of error messages! The first error message will be something like this:

```
c:\test.cpp(45) : error C2679: binary '<<' : no operator found which takes a
right-hand operand of type 'Cents' (or there is no acceptable conversion)
```

Remember that Average() returns a Cents object, and we are trying to stream that object to cout using the << operator. However, we haven't defined the << operator for our Cents class yet. Let's do that:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5  public:
6      Cents(int nCents)
7          : m_nCents(nCents)
8      {
9      }
10
11     friend ostream& operator<< (ostream &out, const Cents &cCents)
12     {
13         out << cCents.m_nCents << " cents ";
14         return out;
15     }
16 };
```

If we compile again, we will get another error:

```
c:\test.cpp(14) : error C2676: binary '+=' : 'Cents' does not define this
operator or a conversion to a type acceptable to the predefined operator
```

This error is actually being caused by the function template instance created when we call Average(Cents*, int). Remember that when we call a templated function, the compiler "stencils" out a copy of the function where the template type parameters (the placeholder types) have been replaced with the actual types in the function call. Here is the function template instance for Average() when T is a Cents object:

```
1  template <class T>
2  Cents Average(Cents  *atArray, int nNumValues)
3  {
4      Cents tSum = 0;
5      for (int nCount=0; nCount < nNumValues; nCount++)
6          tSum += atArray[nCount];
7
8      tSum /= nNumValues;
9      return tSum;
10 }
```

The reason we are getting an error message is because of the following line:

```
1  tSum += atArray[nCount];
```

In this case, tSum is a Cents object, but we have not defined the += operator for Cents objects! We will need to define this function in order for Average() to be able to work with Cents. Looking forward, we can see that Average() also uses the /= operator, so we will go ahead and define that as well:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5  public:
6      Cents(int nCents)
7          : m_nCents(nCents)
8      {
9      }
10
11     friend ostream& operator<< (ostream &out, const Cents &cCents)
12     {
13         out << cCents.m_nCents << " cents ";
14         return out;
15     }
16
17     void operator+=(Cents cCents)
18     {
19         m_nCents += cCents.m_nCents;
20     }
21
22     void operator/=(int nValue)
23     {
24         m_nCents /= nValue;
25     }
26 };
```

Finally, our code will compile and run! Here is the result:

```
11 cents
```

Note that we didn't have to modify Average() at all to make it work with objects of type Cents. We simply had to define the operators used to implement Average() for the Cents class, and the compiler took care of the rest!

14.3 — Template classes

In the previous two lessons, you learn how function templates and function template instances could be used to generalize functions to work with many different data types. While this is a great start down the road to generalized programming, it doesn't solve all of our problems. Let's take a look at an example of one such problem, and see what templates can do for us further.

Templates and container classes

In the lesson on container classes, you learned how to use composition to implement classes that contained multiple instances of other classes. As one example of such a container, we took a look at the IntArray class. Here is a simplified example of that class:

```
1  #ifndef INTARRAY_H
2  #define INTARRAY_H
3
4  #include <assert.h> // for assert()
5
6  class IntArray
7  {
8  private:
9      int m_nLength;
10     int *m_pnData;
11
12 public:
13     IntArray()
14     {
15         m_nLength = 0;
16         m_pnData = 0;
17     }
18
19     IntArray(int nLength)
20     {
21         m_pnData = new int[nLength];
22         m_nLength = nLength;
23     }
24
25     ~IntArray()
26     {
27         delete[] m_pnData;
28     }
29
30     void Erase()
31     {
32         delete[] m_pnData;
33         // We need to make sure we set m_pnData to 0 here, otherwise it
34 will
35         // be left pointing at deallocated memory!
36         m_pnData = 0;
```

```
        m_nLength = 0;
    }

    int& operator[](int nIndex)
    {
        assert(nIndex >= 0 && nIndex < m_nLength);
        return m_pnData[nIndex];
    }

    int GetLength() { return m_nLength; }
};

#endif
```

While this class provides an easy way to create arrays of integers, what if we want to create an array of doubles? Using traditional programming methods, we'd have to create an entirely new class! Here's an example of DoubleArray, an array class used to hold doubles.

```
#ifndef DOUBLEARRAY_H
#define DOUBLEARRAY_H

#include <assert.h> // for assert()

class DoubleArray
{
private:
    int m_nLength;
    double *m_pdData;

public:
    DoubleArray()
    {
        m_nLength = 0;
        m_pdData= 0;
    }

    DoubleArray(int nLength)
    {
        m_pdData= new double[nLength];
        m_nLength = nLength;
    }

    ~DoubleArray()
    {
        delete[] m_pdData;
    }

    void Erase()
    {
        delete[] m_pdData;
        // We need to make sure we set m_pnData to 0 here, otherwise it
will
        // be left pointing at deallocated memory!
```

```
        m_pdData= 0;
        m_nLength = 0;
    }

    double& operator[](int nIndex)
    {
        assert(nIndex >= 0 && nIndex < m_nLength);
        return m_pdData[nIndex];
    }

    // The length of the array is always an integer
    // It does not depend on the data type of the array
    int GetLength() { return m_nLength; }
};

#endif
```

Although the code listings are lengthy, you'll note the two classes are almost identical! In fact, the only substantive difference is the contained data type. As you likely have guessed, this is another area where templates can be put to good use to free us from having to create classes that are bound to one specific data type.

Creating template classes is works pretty much identically to creating template functions, so we'll proceed by example. Here's the IntArray classes, templatated version:

```
#ifndef ARRAY_H
#define ARRAY_H

#include <assert.h> // for assert()

template <typename T>
class Array
{
private:
    int m_nLength;
    T *m_ptData;

public:
    Array()
    {
        m_nLength = 0;
        m_ptData = 0;
    }

    Array(int nLength)
    {
        m_ptData= new T[nLength];
        m_nLength = nLength;
    }

    ~Array()
    {
        delete[] m_ptData;
```

```
25      }
26
27      void Erase()
28      {
29          delete[] m_ptData;
30          // We need to make sure we set m_pnData to 0 here, otherwise it
    will
31          // be left pointing at deallocated memory!
32          m_ptData= 0;
33          m_nLength = 0;
34      }
35
36      T& operator[](int nIndex)
37      {
38          assert(nIndex >= 0 && nIndex < m_nLength);
39          return m_ptData[nIndex];
40      }
41
42      // The length of the array is always an integer
43      // It does not depend on the data type of the array
44      int GetLength(); // templated GetLength() function defined below
45  };
46
    template <typename T>
    int Array<T>::GetLength() { return m_nLength; }

    #endif
```

As you can see, this version is almost identical to the IntArray version, except we've added the template declaration, and changed the contained data type from int to T.

Note that we've also defined the GetLength() function outside of the class declaration. This isn't necessary, but new programmers typically stumble when trying to do this for the first time due to the syntax, so an example is instructive. Each templated member function declared outside the class declaration needs its own template declaration. Also, note that the name of the templated array class is Array<T>, not Array — Array would refer to a non-templated version of a class named Array.

Here's a short example using the above templated array class:

```
1   int main()
2   {
3       Array<int> anArray(12);
4       Array<double> adArray(12);
5
6       for (int nCount = 0; nCount < 12; nCount++)
7       {
8           anArray[nCount] = nCount;
9           adArray[nCount] = nCount + 0.5;
10      }

        for (int nCount = 11; nCount >= 0; nCount----;)
```

```
        std::cout << anArray[nCount] << "\t" << adArray[nCount] <<
std::endl;

    return 0;
}
```

This example prints the following:

```
11      11.5
10      10.5
9        9.5
8        8.5
7        7.5
6        6.5
5        5.5
4        4.5
3        3.5
2        2.5
1        1.5
0        0.5
```

Templated classes are instanced in the same way templated functions are — the compile stencils a copy upon demand with the template parameter replaced by the actual data type the user needs and then compiles the copy. If you don't ever use a template class, the compile won't even compile it.

Template classes are ideal for implementing container classes, because it is highly desirable to have containers work across a wide variety of data types, and templates allow you to do so without duplicating code. Although the syntax is ugly, and the error messages can be cryptic, template classes are truly one of C++'s best and most useful features.

A note for users using older compilers

Some older compilers (eg. Visual Studio 6) have a bug where the definition of template class functions must be put in the same file as the template class is defined in. Thus, if the template class were defined in X.h, the function definitions would have to also go in X.h (not X.cpp). This issue should be fixed in most/all modern compilers.

14.4 — Expression parameters and template specialization

In previous lessons, you've learned how to use template type parameters to create functions and classes that are type independent. However, template type parameters are not the only type of template parameters available. Template classes (not template functions) can make use of another kind of template parameter known as an expression parameter.

Expression parameters

A template expression parameter is a parameter that does not substitute for a type, but is instead replaced by a value. An expression parameter can be any of the following:

- A value that has an integral type or enumeration
- A pointer or reference to an object
- A pointer or reference to a function
- A pointer or reference to a class member function

In the following example, we create a buffer class that uses both a type parameter and an expression parameter. The type parameter controls the data type of the buffer array, and the expression parameter controls how large the buffer array is.

```
1  template <typename T, int nSize> // nSize is the expression parameter
2  class Buffer
3  {
4  private:
5      // The expression parameter controls the size of the array
6      T m_atBuffer[nSize];
7
8  public:
9      T* GetBuffer() { return m_atBuffer; }
10
11     T& operator[](int nIndex)
12     {
13         return m_atBuffer[nIndex];
14     }
15 };
16
17 int main()
18 {
19     // declare an integer buffer with room for 12 chars
20     Buffer<int, 12> cIntBuffer;
21
22     // Fill it up in order, then print it backwards
23     for (int nCount=0; nCount < 12; nCount++)
24         cIntBuffer[nCount] = nCount;
```

```
21
22      for (int nCount=11; nCount >= 0; nCount--)
23          std::cout << cIntBuffer[nCount] << " ";
24      std::cout << std::endl;
25
26      // declare a char buffer with room for 31 chars
27      Buffer<char, 31> cCharBuffer;
28
29      // strcpy a string into the buffer and print it
30      strcpy(cCharBuffer.GetBuffer(), "Hello there!");
31      std::cout << cCharBuffer.GetBuffer() << std::endl;
32
        return 0;
}
```

This code produces the following:

```
11 10 9 8 7 6 5 4 3 2 1 0
Hello there!
```

One noteworthy thing about the above example is that we do not have to dynamically allocate the m_atBuffer member array! This is because for any given instance of the Buffer class, nSize is actually constant. For example, if you instantiate a Buffer, the compiler replaces nSize with 12. Thus m_atBuffer is of type int[12], which can be allocated statically.

Template specialization

When instantiating a template class for a given type, the compiler stencils out a copy of each templated member function, and replaces the template type parameters with the actual types used in the variable declaration. This means a particular member function will have the same implementation details for each instanced type. While most of the time, this is exactly what you want, occasionally there are cases where it is useful to implement a templated member function slightly different for a specific data type. Template specialization lets you accomplish exactly this.

Let's take a look at a very simple example:

```
1   using namespace std;
2
3   template <typename T>
4   class Storage
5   {
6   private:
7       T m_tValue;
8   public:
9       Storage(T tValue)
10      {
11          m_tValue = tValue;
        }

        ~Storage()
```

```
    {
    }

    void Print()
    {
        std::cout << m_tValue << std::endl;;
    }
};
```

The above code will work fine for many data types:

```
int main()
{
    // Define some storage units
    Storage<int> nValue(5);
    Storage<double> dValue(6.7);

    // Print out some values
    nValue.Print();
    dValue.Print();
}
```

This prints:

```
5
6.7
```

Now, let's say we want double values to output in scientific notation. To do so, we will need to use template specialization to create a specialized version of the Print() function for doubles. This is extremely simple: simply define the specialized function outside of the class definition, replacing the template type with the specific type you wish to redefine the function for. Here is our specialized Print() function for doubles:

```
void Storage<double>::Print()
{
    std::cout << std::scientific << m_tValue << std::endl;
}
```

When the compiler goes to instantiate Storage<double>::Print(), it will see we've already defined one, and it will use the one we've defined instead of stenciling out a version from the generic templated member function.

As a result, when we rerun the above program, it will print:

```
5
6.700000e+000
```

Now let's take a look at another example where template specialization can be useful. Consider what happens if we try to use our templated Storage class with datatype char*:

```
int main()
{
    using namespace std;

    // Dynamically allocate a temporary string
    char *strString = new char[40];

    // Ask user for their name
    cout << "Enter your name: ";
    cin >> strString;

    // Store the name
    Storage<char*> strValue(strString);

    // Delete the temporary string
    delete strString;

    // Print out our value
    strValue.Print(); // This will print garbage
}
```

As it turns out, instead of printing the name the user input, strValue.Print() prints garbage! What's going on here?

When Storage is instantiated for type char*, the constructor for Storage<char*> looks like this:

```
Storage<char*>::Storage(char* tValue)
{
    m_tValue = tValue;
}
```

In other words, this just does a pointer assignment! As a result, m_tValue ends up pointing at the same memory location as strString. When we delete strString in main(), we end up deleting the value that m_tValue was pointing at! And thus, we get garbage when trying to print that value.

Fortunately, we can fix this problem using template specialization. Instead of doing a pointer copy, we'd really like our constructor to make a copy of the input string. So let's write a specialized constructor for datatype char* that does exactly that:

```
Storage<char*>::Storage(char* tValue)
{
    // Allocate memory to hold the tValue string
    m_tValue = new char[strlen(tValue)+1];
    // Copy the actual tValue string into the m_tValue memory we just
allocated
    strcpy(m_tValue, tValue);
}
```

Now when we allocate a variable of type Storage<char*>, this constructor will get used instead of the default one. As a result, m_tValue will receive its own copy of strString. Consequently, when we delete strString, m_tValue will be unaffected.

However, this class now has a memory leak for type char*, because m_tValue will not be deleted when the Storage variable goes out of scope. As you might have guessed, this can also be solved by specializing the Storage<char*> destructor:

```
1  Storage<char*>::~Storage()
2  {
3      delete[] m_tValue;
4  }
```

Now when variables of type ~Storage<char*> go out of scope, the memory allocated in the specialized constructor will be deleted in the specialized destructor.

14.5 — Class template specialization

In the previous lesson on template specialization, we saw how it was possible to specialize member functions of a template class in order to provide different functionality for specific data types. As it turns out, it is not only possible to specialize member functions of a template class, it is also possible to specialize an entire class!

Consider the case where you want to design a class that stores 8 objects. Here's a simplified class to do so:

```
template <typename T>
class Storage8
{
private:
    T m_tType[8];

public:
    void Set(int nIndex, const T &tType)
    {
        m_tType[nIndex] = tType;
    }

    const T& Get(int nIndex)
    {
        return m_tType[nIndex];
    }
};
```

Because this class is templated, it will work fine for any given type:

```
int main()
{
    // Define a Storage8 for integers
    Storage8<int> cIntStorage;

    for (int nCount=0; nCount<8; nCount++)
        cIntStorage.Set(nCount, nCount);

    for (int nCount=0; nCount<8; nCount++)
        std::cout << cIntStorage.Get(nCount) << std::endl;

    // Define a Storage8 for bool
    Storage8<bool> cBoolStorage;
    for (int nCount=0; nCount<8; nCount++)
        cBoolStorage.Set(nCount, nCount & 3);

    for (int nCount=0; nCount<8; nCount++)
        std::cout << (cBoolStorage.Get(nCount) ? "true" : "false") <<
std::endl;
```

```
17
18      return 0;
19  }
```

This example prints:

```
0
1
2
3
4
5
6
7
false
true
true
true
false
true
true
true
```

While this class is completely functional, it turns out that the implementation of Storage8<bool> is much more inefficient than it needs to be. Because all variables must have an address, and the CPU can't address anything smaller than a byte, all variables must be at least a byte in size. Consequently, a variable of type bool ends up using an entire byte even though technically it only needs a single bit to store its true or false value! Thus, a bool is 1 bit of useful information and 7 bits of wasted space. Our Storage8<bool> class, which contains 8 bools, is 1 byte worth of useful information and 7 types of wasted space.

As it turns out, using some basic bit logic, it's possible to compress all 8 bools into a single byte, eliminating the wasted space altogether. However, in order to do this, we'll effectively need to essentially revamp the class, replacing the array of 8 bools with a variable that is a single byte in size. While we could create an entirely new class to do so, this has one major downside: we have to give it a different name. Then the programmer has to remember that Storage8<T> is meant for non-bool types, whereas Storage8Bool (or whatever we name the new class) is meant for bools. That's needless complexity we'd rather avoid. Fortunately, C++ provides us a better method: class template specialization.

Class template specialization

Class template specialization allows us to specialize a template class for a particular data type (or set of data types, if there are multiple templated parameters). In this case, we're going to use class template specialization to write a customized version of Storage8<bool> that will take precedence over the generic Storage8<T> class.

Class template specializations are treated as completely independent classes, even though they are allocated in the same way as the templated class. This means that we can change anything

and everything about our specialization class, including the way it's implemented and even the functions it makes public, just as if it were an independent class. Here's our specialized class:

```
template <> // the following is a template class with no templated
parameters
class Storage8<bool> // we're specializing Storage8 for bool
{
// What follows is just standard class implementation details
private:
    unsigned char m_tType;

public:
    void Set(int nIndex, bool tType)
    {
        // Figure out which bit we're setting/unsetting
        // This will put a 1 in the bit we're interested in turning on/off
        unsigned char nMask = 1 << nIndex;

        if (tType)  // If we're setting a bit
            m_tType |= nMask;  // Use bitwise-or to turn that bit on
        else // if we're turning a bit off
            m_tType &= ~nMask;  // bitwise-and the inverse mask to turn
that bit off
    }

    bool Get(int nIndex)
    {
        // Figure out which bit we're getting
        unsigned char nMask = 1 << nIndex;
        // bitwise-and to get the value of the bit we're interested in
        // Then implicit cast to boolean
        return m_tType & nMask;
    }
};
```

First, note that we start off with `template<>`. The template keyword tells the compiler that what follows is templated, and the empty angle braces means that there aren't any template parameters. In this case, there aren't any template parameters because we're replacing the only template parameter (typename T) with a specific type (bool).

Next, we add `<bool>` to the class name to denote that we're specializing a bool version of Storage8.

All of the other changes are just class implementation details. You do not need to understand how the bit-logic works in order to use the class (though here's a link to the lesson on bitwise operators if you want to figure it out, but need a refresher on how bitwise operators work).

Note that this specialization class utilizes a single unsigned char (1 byte) instead of an array of 8 bools (8 bytes).

Now, when we declare a class of type Storage8<T>, where T is not a bool, we'll get a version stenciled from the generic templated Storage8<T> class. When we declare a class of type Storage8<bool>, we'll get the specialized version we just created. Note that we have kept the publicly exposed interface of both classes the same — while C++ gives us free reign to add, remove, or change functions of Storage8<bool> as we see fit, keeping a consistent interface means the programmer can use either class in exactly the same manner.

We can use the exact same example as before to show both Storage8<T> and Storage8<bool> being instantiated:

```
1  int main()
2  {
3      // Define a Storage8 for integers (instantiates Storage8<T>, where T =
   int)
4      Storage8<int> cIntStorage;
5
6      for (int nCount=0; nCount<8; nCount++)
7          cIntStorage[nCount] = nCount;
8
9      for (int nCount=0; nCount<8; nCount++)
10         std::cout << cIntStorage[nCount] << std::endl;
11
12     // Define a Storage8 for bool  (instantiates Storage8<bool>
   specialization)
13     Storage8<bool> cBoolStorage;
14     for (int nCount=0; nCount<8; nCount++)
15         cBoolStorage.Set(nCount, nCount & 3);
16
17     for (int nCount=0; nCount<8; nCount++)
18         std::cout << (cBoolStorage.Get(nCount) ? "true" : "false") <<
   std::endl;
19
20     return 0;
21 }
```

As you might expect, this prints the same result as the previous example that used the non-specialized version of Storage8<bool>:

```
0
1
2
3
4
5
6
7
0
1
2
3
4
5
```

```
6
7
false
true
true
true
false
true
true
true
```

It’s worth noting again that keeping the public interface between your template class and all of the specializations identical is generally a good idea, as it makes them easier to use — however, it’s not strictly necessary.

14.6 — Partial template specialization

In the lesson on expression parameters and template specialization, you learned how expression parameters could be used to parametrize template classes.

Let's take another look at the Buffer class we used in the previous example:

```
template <typename T, int nSize> // nSize is the expression parameter
class Buffer
{
private:
    // The expression parameter controls the side of the array
    T m_atBuffer[nSize];

public:
    T* GetBuffer() { return m_atBuffer; }

    T& operator[](int nIndex)
    {
        return m_atBuffer[nIndex];
    }
};

int main()
{
    // declare a char buffer
    Buffer<char, 10> cChar10Buffer;

    // copy a value into the buffer
    strcpy(cChar10Buffer.GetBuffer(), "Ten");

    return 0;
}
```

Now, let's say we wanted to write a function to print out a buffer as a string. Although we could implement this as a member function, we're going to do it as a non-member function instead because it will make the successive examples easier to follow.

Using templates, we might write something like this:

```
template <typename T, int nSize>
void PrintBufferString(Buffer<T, nSize> &rcBuf)
{
    std::cout << rcBuf.GetBuffer() << std::endl;
}
```

This would allow us to do the following:

```
1  int main()
2  {
3      // declare a char buffer
4      Buffer<char, 10> cChar10Buffer;
5
6      // copy a value into the buffer
7      strcpy(cChar10Buffer.GetBuffer(), "Ten");
8
9      // Print the value
10     PrintBufferString(cChar10Buffer);
11     return 0;
12 }
```

and get the following result:

```
Ten
```

Although this works, it has a design flaw. Consider the following:

```
1  int main()
2  {
3      // declare an int buffer
4      Buffer<int, 10> cInt10Buffer;
5
6      // copy values into the buffer
7      for (int nCount=0; nCount < 10; nCount++)
8          cInt10Buffer[nCount] = nCount;
9
10     // Print the value?
11     PrintBufferString(cInt10Buffer); // what does this mean?
12     return 0;
13 }
```

This program will compile, execute, and produce the following value (or one similar):

```
0012FF10
```

What happened? PrintBufferString() has std::cout print the value of `rcBuf.GetBuffer()`, which returns a pointer to m_atBuffer! When the data type is a char, cout will print the array as a C-style character string, but when the data type is non-char (such as in this case), cout will print the address that the pointer is holding!

Obviously this case exposes a misuse of this function (as written). Without explicitly examining the code, the programmer would not have any clue that this function does not handle non-char buffers correctly. This is likely to lead to programming errors.

Template specialization

One seemingly useful way to solve this problem is to use template specialization to ensure that only arrays of type char can be passed to PrintBufferString(). As you learned in the previous

lesson, template specialization allows you to define a function where all of the templated types have been resolved to a specific data type.

Here's an example of how that might work here:

```
1  void PrintBufferString(Buffer<char, 10> &rcBuf)
2  {
3      std::cout << rcBuf.GetBuffer() << std::endl;
4  }
5
6  int main()
7  {
8      // declare a char buffer
9      Buffer<char, 10> cChar10Buffer;
10
11     // copy a value into the buffer
12     strcpy(cChar10Buffer.GetBuffer(), "Ten");
13
14     // Print the value
       PrintBufferString(cChar10Buffer);
       return 0;
   }
```

As you can see, we've now specialized PrintBufferString so it will only accept Buffers of type char and of length 10. This means if we try to call PrintBufferString with an int buffer, the compiler will give us an error.

Although this solves the issue of making sure PrintBufferString can not be called with an int Buffer, it brings up another problem: using full template specialization means we have to explicitly define the length of the buffer this function will accept! Consider the following example:

```
1  int main()
2  {
3      Buffer<char, 10> cChar10Buffer;
4      Buffer<char, 11> cChar11Buffer;
5
6      strcpy(cChar10Buffer.GetBuffer(), "Ten");
7      strcpy(cChar11Buffer.GetBuffer(), "Eleven");
8
9      PrintBufferString(cChar10Buffer);
10     PrintBufferString(cChar11Buffer); // this will not compile
11
       return 0;
   }
```

Trying to call PrintBufferString() with cChar11Buffer will not work, because cChar11Buffer is a class of type Buffer<char, 11>, and PrintBufferString() only accepts classes of type Buffer<char, 10>. Even though Buffer<char, 10> and Buffer<char, 11> are both templated from the generic Buffer class, the different template parameters means they are treated as different classes, and can not be intermixed.

Although we could make a copy of PrintBufferString() that could handle Buffer<char, 11>, what happens when we want to call PrintBufferString() will a buffer of size 5, or 14? We'd have to copy the function for each different Buffer size we wanted to use.

Obviously full template specialization is too restrictive a solution here. The solution we are looking for is partial template specialization.

Partial template specialization

Partial template specialization allows us to write functions where some of the template parameters have been fully or partially resolved. In this case, the ideal solution would be to allow PrintBufferString() to accept char Buffers of any length. That means we have to specialize the templated data type, but leave the length in templated form. Fortunately, partial template specialization allows us to do just that!

```
1  template<int nSize>
2  void PrintBufferString(Buffer<char, nSize> &rcBuf)
3  {
4      std::cout << rcBuf.GetBuffer() << std::endl;
   }
```

As you can see here, we've explicitly declared that this function will only work for Buffers of type char, but nSize is still a templated parameter, so it will work for char buffers of any size. That's all there is to it!

Consider the following example:

```
1  int main()
2  {
3      // declare an integer buffer with room for 12 chars
4      Buffer<char, 10> cChar10Buffer;
5      Buffer<char, 11> cChar11Buffer;
6
7      // strcpy a string into the buffer and print it
8      strcpy(cChar10Buffer.GetBuffer(), "Ten");
9      strcpy(cChar11Buffer.GetBuffer(), "Eleven");
10
11     PrintBufferString(cChar10Buffer);
12     PrintBufferString(cChar11Buffer);
13
14     return 0;
   }
```

This prints:

```
Ten
Eleven
```

Just as we expect.

Partial template specialization for pointers

In the previous lesson on expression parameters and template specialization, we took a look at a simple templated Storage class:

```
1  using namespace std;
2
3  template <typename T>
4  class Storage
5  {
6  private:
7      T m_tValue;
8  public:
9      Storage(T tValue)
10     {
11         m_tValue = tValue;
12     }
13
14     ~Storage()
15     {
16     }
17
18     void Print()
       {
           std::cout << m_tValue << std::endl;;
       }
   };
```

We showed that this class had problems when template parameter T was of type char* because of the shallow copy/pointer assignment that takes place in the constructor. In that lesson, we used full template specialization to create a specialized version of the Storage constructor for type char* that allocated memory and created an actual deep copy of tValue. For reference, here's the fully specialized char* Storage constructor:

```
1  Storage<char*>::Storage(char* tValue)
2  {
3      // Allocate memory to hold the tValue string
4      m_tValue = new char[strlen(tValue)+1];
5      // Copy the actual tValue string into the m_tValue memory we just
   allocated
6      strcpy(m_tValue, tValue);
   }
```

While that worked great for Storage<char*>, what about other pointer types? It's fairly easy to see that if T is any pointer type, then we run into the problem of the constructor doing a pointer assignment instead of making an actual copy of the element being pointed to.

Because full template specialization forces us to fully resolve templated types, in order to fix this issue we'd have to define a new specialized constructor for each and every pointer type we wanted to use Storage with! This leads to lots of duplicate code, which as you well know by now is something we want to avoid as much as possible.

Fortunately, partial template specialization offers us a convenient solution. In this case, we'll use class partial template specialization to define a special version of Storage that works for pointer values:

```
using namespace std;

template <typename T>
class Storage<T*> // this is specialization of Storage that works with pointer types
{
private:
    T* m_tValue;
public:
    Storage(T* tValue) // for pointer type T
    {
         m_tValue = new T(*tValue);
    }

    ~Storage()
    {
        delete m_tValue;
    }

    void Print()
    {
        std::cout << *m_tValue << std::endl;
    }
};
```

And an example of this working:

```
int main()
{
    // Declare a non-pointer Storage to show it works
    Storage<int> cIntStorage(5);

    // Declare a pointer Storage to show it works
    int x = 7;
    Storage<int*> cIntPtrStorage(&x);

    // If cIntPtrStorage did a pointer assignment on x,
    // then changing x will change cIntPtrStorage too
    x = 9;
    cIntPtrStorage.Print();

    return 0;
}
```

This prints the value:

```
7
```

The fact that we got a 7 here shows that cIntPtrStorage used the pointer version of Storage, which allocated it's own copy of the int. If cIntPtrStorage had used the non-pointer version of Storage, it would have done a pointer assignment — and when we changed the value of x, we would have changed cIntPtrStorage's value too.

Using partial template class specialization to create separate pointer and non-pointer implementations of a class is extremely useful when you want a class to handle both differently, but in a way that's completely transparent to the end-user.

Exceptions

15.1 — The need for exceptions

In the previous lesson on handling errors, we talked about ways to use assert(), cerr(), and exit() to handle errors. However, we punted on one further topic that we will now cover: exceptions.

When return codes fail

When writing reusable code, error handling is a necessity. One of the most common ways to handle potential errors is via return codes. For example:

```
int FindFirstChar(const char* strString, char chChar)
{
    // Step through each character in strString
    for (int nIndex=0; nIndex < strlen(strString); nIndex++)
        // If the charater matches chChar, return it's index
        if (strString[nIndex] == chChar)
            return nIndex;

    // If no match was found, return -1
    return -1;
}
```

This function returns the index of the first character matching chChar within strString. If the character can not be found, the function returns -1 as an error indicator.

The primary virtue of this approach is that it is extremely simple. However, using return codes has a number of drawbacks which can quickly become apparent when used in non-trivial cases:

First, return values can be cryptic — if a function returns -1, is it trying to indicate an error, or is that actually a valid return value? It's often hard to tell without digging into the guts of the function.

Second, functions can only return one value, so what happens when you need to return both a function result and an error code? Consider the following function:

```
double Divide(int nX, int nY)
{
    return static_cast<double>(nX)/nY;
}
```

This function is in desperate need of some error handling, because it will crash if the user passes in 0 for nY. However, it also needs to return the result of nX/nY. How can it do both? The most common answer is that either the result or the error handling will have to be passed back as a reference parameter, which makes for ugly code that is less convenient to use. For example:

```
1  double Divide(int nX, int nY, bool &bSuccess)
2  {
3      if (nY == 0)
4      {
5          bSuccess = false;
6          return 0.0;
7      }
8
9      bSuccess = true;
10     return static_cast<double>(nX)/nY;
11 }
12
13 int main()
14 {
15     bool bSuccess; // we must now pass in a bSuccess
16     double dResult = Divide(5, 3, bSuccess);
17
18     if (!bSuccess) // and check it before we use the result
           cerr << "An error occurred" << endl;
       else
           cout << "The answer is " << dResult << endl;
   }
```

Third, in sequences of code where many things can go wrong, error codes have to be checked constantly. Consider the following code that involves parsing a text file for values that are supposed to be there:

```
1  std::ifstream fSetupIni("setup.ini"); // open setup.ini for reading
2  if (!fSetupIni)
3      return ERROR_OPENING_FILE; // Some enum value indicating error
4
5  // Read parameters and return an error if the parameter is missing
6  if (!ReadParameter(fSetupIni, m_nFirstParameter))
7      return ERROR_PARAMETER_MISSING; // Some other enum value indicating
   error
8  if (!ReadParameter(fSetupIni, m_nSecondParameter))
9      return ERROR_PARAMETER_MISSING; // Some other enum value indicating
   error
10 if (!ReadParameter(fSetupIni, m_nThirdParameter))
11     return ERROR_PARAMETER_MISSING; // Some other enum value indicating
   error
```

Now imagine if there were twenty parameters — you're essentially checking for an error and returning ERROR_PARAMETER_MISSING twenty times! This makes the function harder to read.

Fourth, return codes do not mix with constructors very well. What happens if you're creating an object and something inside the constructor goes catastrophically wrong? Constructors have no return type to pass back a status indicator, and passing one back via a reference parameter is messy and must be explicitly checked. Furthermore, even if you do this, the object will still be created and then has to be dealt with or disposed of.

Finally, when an error code is returned to the caller, the caller may not always be equipped to handle the error. If the caller doesn't want to handle the error, it either has to ignore it (in which case it will be lost forever), or return the error up the stack to the function that called it. This can be messy and lead to many of the same issues noted above.

The primary issue with return codes is that the error handling code ends up intricately linked to the normal control flow of the code. This in turns ends up constraining both how the code is laid out, and how errors can be reasonably handled.

Exceptions

Exception handling provides a mechanism to decouple handling of errors or other exceptional circumstances from the typical control flow of your code. This allows more freedom to handle errors when and however is most useful for a given situation, alleviating many (if not all) of the messiness that return codes cause.

In the next lesson, we'll take a look at how exceptions work in C++.

15.2 — Basic exception handling

In the previous lesson on introduction to exceptions, we talked about how using return codes causes you control flow and error flow to be intermingled, constraining both. Exceptions in C++ are implemented using three keywords that work in conjunction with each other: **throw**, **try**, and **catch**.

Throwing exceptions

We use signals all the time in real life to note that particular events have occurred. For example, during American football, if a player has committed a foul, the referee will throw a flag on the ground and whistle the play dead. A penalty is then assessed and executed. Once the penalty has been taken care of, play generally resumes as normal.

In C++, a **throw statement** is used to signal that an exception or error case has occurred (think of throwing a penalty flag). Signaling that an exception has occurred is also commonly called **raising** an exception.

To use a throw statement, simply use the throw keyword, followed by a value of any data type you wish to use to signal that an error has occurred. Typically, this value will be an error code, a description of the problem, or a custom exception class.

Here are some examples:

```
1 throw -1; // throw a literal integer value
2 throw ENUM_INVALID_INDEX; // throw an enum value
3 throw "Can not take square root of negative number"; // throw a literal
  char* string
4 throw dX; // throw a double variable that was previously defined
5 throw MyException("Fatal Error"); // Throw an object of class MyException
```

Each of these statements acts a signal that some kind of problem that needs to be handled has occurred.

Looking for exceptions

Throwing exceptions is only one part of the exception handling process. Let's go back to our American football analogy: once a referee has thrown a penalty flag, what happens next? The players notice that a penalty has occurred and stop play. The normal flow of the football game is disrupted.

In C++, we use the **try** keyword to define a block of statements (called a **try block**). The try block acts as an observer, looking for any exceptions that are thrown by statements within the try block.

Here's an example of an try block:

```
1  try
2  {
3      // Statements that may throw exceptions you want to handle now go here
4      throw -1;
   }
```

Note that the try block doesn't define HOW we're going to handle the exception. It merely tells the program, "Hey, if you see an exception in the following code, grab it!".

Handling exceptions

Finally, the end of our American football analogy: After the penalty has been called and play has stopped, the referee assesses the penalty and executes it. In other words, the penalty must be handled before normal play can resume.

Actually handling exceptions is the job of the catch block(s). The **catch** keyword is used to define a block of code (called a **catch block**) that handles exceptions for a single data type.

Here's an example of a catch block:

```
1  catch (int)
2  {
3      // Handle an exception of type int here
4      cerr << "We caught an exception of type int" << endl;
   }
```

Try blocks and catch blocks work together — A try block detects any exceptions that are thrown by statements within the try block, and routes them to the appropriate catch block for handling. A try block must have at least one catch block attached to it, but may have multiple catch blocks listed in sequence:

```
1  int main()
2  {
3      try
4      {
5          // Statements that may throw exceptions you want to handle now go
   here
6          throw -1;
```

```
    }
    catch (int)
    {
        // Any exceptions of type int thrown within the above try block
get sent here
        cerr << "We caught an exception of type int" << endl;
    }
    catch (double)
    {
        // Any exceptions of type double thrown within the above try block
get sent here
        cerr << "We caught an exception of type double" << endl;
    }

    return 0;
}
```

Putting throw, try, and catch together

Running the above try/catch block would produce the following result:

```
We caught an exception of type int
```

A throw statement was used to raise an exception with the value -1, which is of type int. The try block routed the int exception to the catch block that handles exceptions of type int, which then printed the error message.

Exception handling behind the scenes

Let's talk about what happens behind the scenes in a little more detail. Exception handling is actually quite simple, and the following two paragraphs are all you really need to know about it:

When an exception is raised (using **throw**), execution of the program immediately jumps to the nearest enclosing **try** block (propagating up the stack if necessary). If any of the **catch** handlers attached to the try block handle that type of exception, that handler is executed and the exception is considered handled.

If no appropriate catch handlers exist, execution of the program propagates to the next enclosing try block. If no appropriate catch handlers can be found before the end of the program, the program will fail with an exception error.

That's really all there is to it. The rest of this chapter will be dedicated to showing examples of these principles at work.

Exceptions are handled immediately

Here's a short program that demonstrates how exceptions are handled immediately:

```
int main()
```

```
{
    try
    {
        throw 4.5; // throw exception of type double
        cout << "This never prints" << endl;
    }
    catch(double dX) // handle exception of type double
    {
        cerr << "We caught a double of value: " << dX << endl;
    }
}
```

This program is about as simple as it gets. Here's what happens: the throw statement is the first statement that gets executed — this causes an exception of type double to be raised. Execution immediately moves to the nearest enclosing try block, which is the only try block in this program. The catch handlers are then checked to see if any handlers matche. Our exception is of type double, so we're looking for a catch handler of type double. We have one, so it executes.

Consequently, the result of this program is as follows:

```
We caught a double of value: 4.5
```

Note that the cout statement never executed, because the exception caused the execution path to change to the exception handler for doubles.

A more realistic example

Let's take a look at an example that's not quite so academic:

```
#include "math.h" // for sqrt() function
using namespace std;

int main()
{
    cout << "Enter a number: ";
    double dX;
    cin >> dX;

    try // Look for exceptions that occur within try block and route to
attached catch block(s)
    {
        // If the user entered a negative number, this is an error
condition
        if (dX < 0.0)
            throw "Can not take sqrt of negative number"; // throw
exception of type char*

        // Otherwise, print the answer
        cout << "The sqrt of " << dX << " is " << sqrt(dX) << endl;
    }
    catch (char* strException) // catch exceptions of type char*
    {
```

```
        cerr << "Error: " << strException << endl;
    }
}
```

In this code, the user is asked to enter a number. If they enter a positive number, the if statement does not execute, no exception is thrown, and the square root of the number is printed. Because no exception is thrown in this case, the code inside the catch block never executes. The result is something like this:

```
Enter a number: 9
The sqrt of 9 is 3
```

If the user enters a negative number, we throw an exception of type char*. Because we're within a try block and a matching exception handler is found, control immediately transfers to the char* exception handler. The result is:

```
Enter a number: -4
Error: Can not take sqrt of negative number
```

By now, you should be getting the basic idea behind exceptions. In the next lesson, we'll do quite a few more examples to show how flexible exceptions are.

Multiple statements within a try block

In the lesson on the need for exceptions, we showed an example of one case where return codes don't work very well:

```
std::ifstream fSetupIni("setup.ini"); // open setup.ini for reading
if (!fSetupIni)
    return ERROR_OPENING_FILE; // Some enum value indicating error

// Note that error handling and actual code logic are intermingled

if (!ReadParameter(fSetupIni, m_nFirstParameter))
    return ERROR_PARAMETER_MISSING; // Some other enum value indicating
error
if (!ReadParameter(fSetupIni, m_nSecondParameter))
    return ERROR_PARAMETER_MISSING; // Some other enum value indicating
error
if (!ReadParameter(fSetupIni, m_nThirdParameter))
    return ERROR_PARAMETER_MISSING; // Some other enum value indicating
error
```

In this code, ReadParameter() is returning a boolean value indicating success or failure. We end up having to check the return code from each call to ReadParameter() to ensure that it succeeded before proceeding. This leads to code that is messy and redundant.

Let's rewrite this snippet of code using a new version of ReadParameter() throws an int exception on failure instead of returning a boolean value:

```
1  std::ifstream fSetupIni("setup.ini"); // open setup.ini for reading
2  if (!fSetupIni)
3      return ERROR_OPENING_FILE; // Some enum value indicating error
4
5  // Read parameters and return an error if the parameter is missing
6  try
7  {
8      // Here's all the normal code logic
9      m_nFirstParameter = ReadParameter(fSetupIni);
10     m_nSecondParameter = ReadParameter(fSetupIni);
11     m_nThirdParameter = ReadParameter(fSetupIni);
12 }
13 catch (int) // Here's the error handling, nicely separated
   {
       return ERROR_PARAMETER_MISSING; // Some other enum value indicating
   error
   }
```

Note how much easier this is to read! If any of the calls to ReadParameter() throws an exception, that exception will be caught and routed to the int exception handler, which returns an error enum to the caller.

15.3 — Exceptions, functions, and stack unwinding

In the previous lesson on basic exception handling, we explained how throw, try, and catch work together to enable exception handling. This lesson is dedicated to showing more examples of exception handling at work in various cases.

Exceptions within functions

In all of the examples in the previous lesson, the throw statements were placed directly within a try block. If this were a necessity, exception handling would be of limited use.

One of the most useful properties of exception handling is that the throw statements do NOT have to be placed directly inside a try block due to the way exceptions propagate when thrown. This allows us to use exception handling in a much more modular fashion. We'll demonstrate this by rewriting the square root program from the previous lesson to use a modular function.

```
1  #include "math.h" // for sqrt() function
2  using namespace std;
3
4  // A modular square root function
5  double MySqrt(double dX)
6  {
7      // If the user entered a negative number, this is an error condition
8      if (dX < 0.0)
9          throw "Can not take sqrt of negative number"; // throw exception
   of type char*
10
11     return sqrt(dX);
12 }
13
14 int main()
   {
```

```
15      cout << "Enter a number: ";
16      double dX;
17      cin >> dX;
18
19      try // Look for exceptions that occur within try block and route to
    attached catch block(s)
20      {
21          cout << "The sqrt of " << dX << " is " << MySqrt(dX) << endl;
22      }
23      catch (char* strException) // catch exceptions of type char*
24      {
25          cerr << "Error: " << strException << endl;
        }
    }
```

In this program, we've taken the code that checks for an exception and calculates the square root and put it inside a modular function called MySqrt(). We've then called this MySqrt() function from inside a try block. Let's verify that it still works as expected:

```
Enter a number: -4
Error: Can not take sqrt of negative number
```

It does!

The most interesting part of this program is the MySqrt() function, which potentially raises an exception. However, you will note that this exception is not inside of a try block! This essentially means MySqrt is willing to say, "Hey, there's a problem!", but is unwilling to handle the problem itself. It is, in essence, delegating that responsibility to its caller (the equivalent of how using a return code passes the responsibility of handling an error back to a function's caller).

Let's revisit for a moment what happens when an exception is raised. First, the program looks to see if the exception can be handled immediately (which means it was thrown inside a try block). If not, it immediately terminates the current function and checks to see if the caller will handle the exception. If not, it terminates the caller and checks the caller's caller. Each function is terminated in sequence until a handler for the exception is found, or until main() terminates. This process is called **unwinding the stack** (see the lesson on the stack and the heap if you need a refresher on what the call stack is).

Now, let's take a detailed look at how that applies to this program when MySqrt(-4) is called and an exception is raised.

First, the program checks to see if we're immediately inside a try block within the function. In this case, we are not. Then, the stack begins to unwind. First, MySqrt() terminates, and control returns to main(). The program now checks to see if we're inside a try block. We are, and there's a char* handler, so the exception is handled by the try block within main(). To summarize, MySqrt() raised the exception, but the try/catch block in main() was the one who captured and handled the exception.

At this point, some of you are probably wondering why it's a good idea to pass errors back to the caller. Why not just make MySqrt() handle it's own error? The problem is that different applications may want to handle errors in different ways. A console application may want to print a text message. A windows application may want to pop up an error dialog. In one application, this may be a fatal error, and in another application it may not. By passing the error back up the stack, each application can handle an error from MySqrt() in a way that is the most context appropriate for it! Ultimately, this keeps MySqrt() as modular as possible, and the error handling can be placed in the less-modular parts of the code.

Another stack unwinding example

Here's another example showing stack unwinding in practice, using a larger stack. Although this program is long, it's pretty simple: main() calls First(), First() calls Second(), Second() calls Third(), Third() calls Last(), and Last() throws an exception.

```
1  #include <iostream>
2  using namespace std;
3
4  void Last() // called by Third()
5  {
6      cout << "Start Last" << endl;
7      cout << "Last throwing int exception" << endl;
8      throw -1;
9      cout << "End Last" << endl;
10
11 }
12
13 void Third() // called by Second()
14 {
15     cout << "Start Third" << endl;
16     Last();
17     cout << "End Third" << endl;
18 }
19
20 void Second() // called by First()
21 {
22     cout << "Start Second" << endl;
23     try
24     {
25         Third();
26     }
27     catch(double)
28     {
29          cerr << "Second caught double exception" << endl;
30     }
31     cout << "End Second" << endl;
32 }
33
34 void First() // called by main()
35 {
36     cout << "Start First" << endl;
37     try
```

```
32      {
33          Second();
34      }
35      catch (int)
36      {
37           cerr << "First caught int exception" << endl;
38      }
39      catch (double)
40      {
41           cerr << "First caught double exception" << endl;
42      }
43      cout << "End First" << endl;
44  }
45
46  int main()
47  {
48      cout << "Start main" << endl;
49      try
50      {
51          First();
52      }
53      catch (int)
54      {
55           cerr << "main caught int exception" << endl;
56      }
57      cout << "End main" << endl;
58
59      return 0;
60  }
```

Take a look at this program in more detail, and see if you can figure out what gets printed and what doesn't when it is run. The answer follows:

```
Start main
Start First
Start Second
Start Third
Start Last
Last throwing int exception
First caught int exception
End First
End main
```

Let's examine what happens in this case. The printing of all the start statements is straightforward and doesn't warrant further explanation. Last() prints "Last throwing int exception" and then throws an int exception. This is where things start to get interesting.

Because Last() doesn't handle the exception itself, the stack begins to unwind. Last() terminates immediately and control returns to the caller, which is Third().

Third() doesn't handle any exceptions either, so it terminates immediately and control returns to Second().

Second() has a try block, and the call to Third() is within it, so the program attempts to match the exception with an appropriate catch block. However, there are no handlers for exceptions of type int here, so Second() terminates immediately and control returns to First().

First() also has a try block, and the call to Second() is within it, so the program looks to see if there is a catch handler for int exceptions. There is! Consequently, First() handles the exception, and prints "First caught int exception".

Because the exception has now been handled, control continues normally at the end of the catch block within First(). This means First() prints "End First" and then terminates normally.

Control returns to main(). Although main() has an exception handler for int, our exception has already been handled by First(), so the catch block within main() does not get executed. main() simply prints "End main" and then terminates normally.

There are quite a few interesting principles illustrated by this program:

First, the immediate caller of a function that throws an exception doesn't have to handle the exception if it doesn't want to. In this case, Third() didn't handle the exception thrown by Last(). It delegated that responsibility to one of it's callers up the stack.

Second, if a try block doesn't have a catch handler for the type of exception being thrown, stack unwinding occurs just as if there were no try block at all. In this case, Second() didn't handle the exception either because it didn't have the right kind of catch block.

Third, once an exception is handled, control flow proceeds as normal starting from the end of the catch blocks. This was demonstrated by First() handling the error and then terminating normally. By the time the program got back to main(), the exception had been thrown and handled already — main() had no idea there even was an exception at all!

As you can see, stack unwinding provides us with some very useful behavior — if a function does not want to handle an exception, it doesn't not have to. The exception will propagate up the stack until it finds someone who will! This allows us to decide where in the call stack is the most appropriate place to handle any errors that may occur.

In the next lesson, we'll take a look at what happens when you don't capture an exception, and a method to prevent that from happening.

15.4 — Uncaught exceptions, catch-all handlers, and exception specifiers

By now, you should have a reasonable idea of how exceptions work. In this lesson, we'll cover a few more interesting exception cases.

Uncaught exceptions

In the past few examples, there are quite a few cases where a function assumes its caller (or another function somewhere up the call stack) will handle the exception. In the following example, MySqrt() assumes someone will handle the exception that it throws — but what happens if nobody actually does?

Here's our square root program again, minus the try block in main():

```
1  #include "math.h" // for sqrt() function
2  using namespace std;
3
4  // A modular square root function
5  double MySqrt(double dX)
6  {
7      // If the user entered a negative number, this is an error condition
8      if (dX < 0.0)
9          throw "Can not take sqrt of negative number"; // throw exception
   of type char*
10
11     return sqrt(dX);
12 }
```

```
int main()
{
    cout << "Enter a number: ";
    double dX;
    cin >> dX;

    // Look ma, no exception handler!
    cout << "The sqrt of " << dX << " is " << MySqrt(dX) << endl;
}
```

Now, let's say the user enters -4, and MySqrt(-4) raises an exception. MySqrt() doesn't handle the exception, so the program stack unwinds and control returns to main(). But there's no exception handler here either, so main() terminates. At this point, we just terminated our application!

When main() terminates with an unhandled exception, the operating system will generally notify you that an unhandled exception error has occurred. How it does this depends on the operating system, but possibilities include printing an error message, popping up an error dialog, or simply crashing. Some OS's are less graceful than others. Generally this is something you want to avoid altogether!

Catch-all handlers

And now we find ourselves in a condundrum: functions can potentially throw exceptions of any data type, and if an exception is not caught, it will propagate to the top of your program and cause it to terminate. Since it's possible to call functions without knowing how they are even implemented, how can we possibly prevent this from happening?

Fortunately, C++ provides us with a mechanism to catch all types of exceptions. This is known as a **catch-all handler**. A catch-all handler works just like a normal catch block, except that instead of using a specific type to catch, it uses the ellipses operator (…) as the type to catch. If you recall from lesson 7.14 on ellipses and why to avoid them, ellipses were previously used to pass arguments of any type to a function. In this context, they represent exceptions of any data type. Here's an simple example:

```
try
{
    throw 5; // throw an int exception
}
catch (double dX)
{
    cout << "We caught an exception of type double: " << dX << endl;
}
catch (...) // catch-all handler
{
    cout << "We caught an exception of an undetermined type" << endl;
}
```

Because there is no specific exception handler for type int, the catch-all handler catches this exception. This example produces the following result:

```
We caught an exception of an undetermined type
```

The catch-all handler should be placed last in the catch block chain. This is to ensure that exceptions can be caught by exception handlers tailored to specific data types if those handlers exist. Visual Studio enforces this constraint — I am unsure if other compilers do.

Often, the catch-all handler block is left empty:

```
1  catch(...) {} // ignore any unanticipated exceptions
```

This will catch any unanticipated exceptions and prevent them from stack unwinding to the top of your program, but does no specific error handling.

Using the catch-all handler to wrap main()

One interesting use for the catch-all handler is to wrap the contents of main():

```
1  int main()
2  {
3
4      try
5      {
6          RunGame();
7      }
8      catch(...)
9      {
10         cerr << "Abnormal termination" << endl;
11     }
12
13     SaveState(); // Save user's game
14     return 1;
15 }
```

In this case, if RunGame() or any of the functions it calls throws an exception that is not caught, that exception will unwind up the stack and eventually get caught by this catch-all handler. This will prevent main() from terminating, and gives us a chance to print an error of our choosing and then save the user's state before exiting. This can be useful to catch and handle problems that may be unanticipated.

Exception specifiers

This subsection should be considered optional reading because exception specifiers are rarely used in practice, are not well supported by compilers, and Bjarne Stroustrup (the creator of C++) considers them a failed experiment.

Exception specifiers are a mechanism that allows us to use a function declaration to specify whether a function may or will not throw exceptions. This can be useful in determining whether a function call needs to be put inside a try block or not.

There are three types of exception specifiers, all of which use what is called the **throw (…)** syntax.

First, we can use an empty throw statement to denote that a function does not throw any exceptions outside of itself:

```
1  int DoSomething() throw(); // does not throw exceptions
```

Note that DoSomething() can still use exceptions as long as they are handled internally. Any function that is declared with throw() is supposed to cause the program to terminate immediately if it does try to throw an exception outside of itself, but implementation is spotty.

Second, we can use a specific throw statement to denote that a function may throw a particular type of exception:

```
1  int DoSomething() throw(double); // may throw a double
```

Finally, we can use a catch-all throw statement to denote that a function may throw an unspecified type of exception:

```
1  int DoSomething() throw(...); // may throw anything
```

Due to the incomplete compiler implementation, the fact that exception specifiers are more like statements of intent than guarantees, some incompatibility with template functions, and the fact that most C++ programmers are unaware of their existence, I recommend you do not bother using exception specifiers.

15.5 — Exceptions, classes, and inheritance

Exceptions and member functions

Up to this point in the tutorial, you've only seen exceptions used in non-member functions. However, exceptions are equally useful in member functions, and even moreso in overloaded operators. Consider the following overloaded [] operator as part of a simple integer array class:

```
1 int IntArray::operator[](const int nIndex)
2 {
3     return m_nData[nIndex];
  }
```

Although this function will work great as long as nIndex is a valid array index, this function is sorely lacking in some good error checking. We could add an assert statement to ensure the index is valid:

```
1 int IntArray::operator[](const int nIndex)
2 {
3     assert (nIndex >= 0 && nIndex < GetLength());
4     return m_nData[nIndex];
  }
```

Now if the user passes in an invalid index, the program will cause an assertion error. While this is useful to indicate to the user that something went wrong, sometimes the better course of action is to fail silently and let the caller know something went wrong so they can deal with it as appropriate.

Unfortunately, because overloaded operators have specific requirements as to the number and type of parameter(s) they can take and return, there is no flexibility for passing back error codes

or boolean values to the caller. However, since exceptions do not change the signature of a function, they can be put to great use here. Here's an example:

```
int IntArray::operator[](const int nIndex)
{
    if (nIndex < 0 || nIndex >= GetLength())
        throw nIndex;

    return m_nData[nIndex];
}
```

Now, if the user passes in an invalid exception, operator[] will throw an int exception.

When constructors fail

Constructors are another area of classes in which exceptions can be very useful. If a constructor fails, simply throw an exception to indicate the object failed to create. The object's construction is aborted and its destructor is never executed.

Exception classes

One of the major problem with using basic data types (such as int) as exception types is that they are inherently vague. An even bigger problem is disambiguation of what an exception means when there are multiple statements or function calls within a try block.

```
try
{
    int *nValue = new int(anArray[nIndex1] + anArray[nIndex2]);
}
catch (int nValue)
{
    // What are we catching here?
}
```

In this example, if we were to catch an int exception, what does that really tell us? Was one of the array indexes out of bounds? Did operator+ cause integer overflow? Did operator new fail because it ran out of memory? Unfortunately, in this case, there's just no easy way to disambiguate. While we can throw char* exceptions to solve the problem of identifying WHAT went wrong, this still does not provide us the ability to handle exceptions from various sources differently.

One way to solve this problem is to use exception classes. An **exception class** is just a normal class that is designed specifically to be thrown as an exception. Let's design a simple exception class to be used with our IntArray class:

```
class ArrayException
{
private:
    std::string m_strError;

```

```
5
6       ArrayException() {}; // not meant to be called
7   public:
8       ArrayException(std::string strError)
9           : m_strError(strError)
10      {
11      }

        std::string GetError() { return m_strError; }
    }
```

Here's our overloaded operator[] throwing this class:

```
1   int IntArray::operator[](const int nIndex)
2   {
3       if (nIndex < 0 || nIndex >= GetLength())
4           throw ArrayException("Invalid index");
5
        return m_nData[nIndex];
    }
```

And a sample usage of this class:

```
1   try
2   {
3       int nValue = anArray[5];
    }
4   catch (ArrayException &cException)
5   {
6       cerr << "An array exception occurred (" << cException.GetError() << ")"
7   << endl;
    }
```

Using such a class, we can have the exception return a description of the problem that occurred, which provides context for what went wrong. And since ArrayException is it's own unique type, we can specifically catch exceptions thrown by the array class and treat them differently from other exceptions if we wish.

Note that exception handlers should catch class exception objects by reference instead of by value. This prevents the compiler from make a copy of the exception, which can be expensive when the exception is a class object. Catching exceptions by pointer should generally be avoided unless you have a specific reason to do so.

std::exception

The C++ standard library comes with an exception class that is used by many of the other standard library classes. The class is almost identical to the ArrayException class above, except the GetError() function is named what():

```
1   try
    {
```

```
    // do some stuff with the standard library here
}
catch (std::exception &cException)
{
    cerr << "Standard exception: " << cException.what() << endl;
}
```

We'll talk more about std::exception in a moment.

Exceptions and inheritance

Since it's possible to throw classes as exceptions, and classes can be derived from other classes, we need to consider what happens when we use inherited classes as exceptions. As it turns out, exception handlers will not only match classes of a specific type, they'll also match classes derived from that specific type as well! Consider the following example:

```
class Base
{
public:
    Base() {}
};

class Derived: public Base
{
public:
    Derived() {}
};

int main()
{
    try
    {
        throw Derived();
    }
    catch (Base &cBase)
    {
        cerr << "caught Base";
    }
    catch (Derived &cDerived)
    {
        cerr << "caught Derived";
    }

    return 0;
}
```

In the above example we throw an exception of type Derived. However, the output of this program is:

```
caught Base
```

What happened?

First, as mentioned above, derived classes will be caught by handlers for the base type. Because Derived is derived from Base, Derived is-a Base (they have an is-a relationship). Second, when C++ is attempting to find a handler for a raised exception, it does so sequentially. Consequently, the first thing C++ does is check whether the exception handler for Base matches the Derived exception. Because Derived is-a Base, the answer is yes, and it executes the catch block for type Base! The catch block for Derived is never even tested in this case.

In order to make this example work as expected, we need to flip the order of the catch blocks:

```
1  class Base
2  {
3  public:
4      Base() {}
5  };
6
7  class Derived: public Base
8  {
9  public:
10     Derived() {}
11 };
12
13 int main()
14 {
15     try
16     {
17         throw Derived();
18     }
19     catch (Derived &cDerived)
20     {
21         cerr << "caught Derived";
22     }
23     catch (Base &cBase)
24     {
25         cerr << "caught Base";
26     }
27
28     return 0;
29 }
```

This way, the Derived handler will get first shot at catching objects of type Derived (before the handler for Base can). Objects of type Base will not match the Derived handler (Derived is-a Base, but Base is not a Derived), and thus will "fall through" to the Base handler.

Rule: Handlers for derived exception classes should be listed before those for base classes.

The ability to use a handler to catch exceptions of derived types using a handler for the base class turns out to be exceedingly useful.

Let's take a look at this using std::exception. There are many classes derived from std::exception, such as std::bad_alloc, std::bad_cast, std::runtime_error, and others. When the standard library

has an error, it can throw a derived exception correlating to the appropriate specific problem it has encountered.

Most of the time, we probably won't care whether the problem was a bad allocation, a bad cast, or something else. We just care that we got an exception from the standard library. In this case, we just set up an exception handler to catch std::exception, and we'll end up catching std::exception and all of the derived exceptions together in one place. Easy!

```
1  try
2  {
3       // code using standard library goes here
   }
4  // This handler will catch std::exception and all the derived exceptions
5  too
6  catch (std::exception &cException)
7  {
8      cerr << "Standard exception: " << cException.what() << endl;
   }
```

However, sometimes we'll want to handle a specific type of exception differently. In this case, we can add a handler for that specific type, and let all the others "fall through" to the base handler. Consider:

```
1  try
2  {
3       // code using standard library goes here
   }
4  // This handler will catch std::bad_alloc (and any exceptions derived from
5  it) here
6  catch (std::bad_alloc &cException)
7  {
8      cerr << "You ran out of memory!" << endl;
   }
9  // This handler will catch std::exception (and any exception derived from
10 it) that fall
11 // through here
12 catch (std::exception &cException)
13 {
       cerr << "Standard exception: " << cException.what() << endl;
   }
```

In this example, exceptions of type std::bad_alloc will be caught by the first handler and handled there. Exceptions of type std::exception and all of the other derived classes will be caught by the second handler.

Such inheritance hierarchies allow us to use specific handlers to target specific derived exception classes, or to use base class handlers to catch the whole hierarchy of exceptions. This allows us a fine degree of control over what kind of exceptions we want to handle while ensuring we don't have to do too much work to catch "everything else" in a hierarchy.

15.6 — Exception dangers and downsides

As with almost everything that has benefits, there are some potential downsides to exceptions as well. This article is not meant to be comprehensive, but just to point out some of the major issues that should be considered when using exceptions (or deciding whether to use them).

Cleaning up resources

One of the biggest problems that new programmers run into when using exceptions is the issue of cleaning up resources when an exception occurs. Consider the following example:

```
try
{
    OpenFile(strFilename);
    WriteFile(strFilename, strData);
    CloseFile(strFilename);
}
catch (FileException &cException)
{
    cerr << "Failed to write to file: " << cException.what() << endl;
}
```

What happens if WriteFile() fails and throws a FileException? At this point, we've already opened the file, and now control flow jumps to the FileException handler, which prints an error and exits. Note that the file was never closed! This example should be rewritten as follows:

```
try
{
    OpenFile(strFilename);
    WriteFile(strFilename, strData);
    CloseFile(strFilename);
```

```
}
catch (FileException &cException)
{
    // Make sure file is closed
    CloseFile(strFilename);
    // Then write error
    cerr << "Failed to write to file: " << cException.what() << endl;
}
```

This kind of error often crops up in another form when dealing with dynamically allocated memory:

```
try
{
    Person *pJohn = new Person("John", 18, E_MALE);
    ProcessPerson(pJohn);
    delete pJohn;
}
catch (PersonException &cException)
{
    cerr << "Failed to process person: " << cException.what() << endl;
}
```

If ProcessPerson() throws an exception, control flow jumps to the catch handler. As a result, pJohn is never deallocated! This example is a little more tricky than the previous one — because pJohn is local to the try block, it goes out of scope when the try block exits. That means the exception handler can not access pJohn at all (its been destroyed already), so there's no way for it to deallocate the memory.

However, there are two relatively easy ways to fix this. First, declare pJohn outside of the try block so it does not go out of scope when the try block exits:

```
Person *pJohn = NULL;
try
{
    pJohn = new Person("John", 18, E_MALE);
    ProcessPerson(pJohn );
    delete pJohn;
}
catch (PersonException &cException)
{
    delete pJohn;
    cerr << "Failed to process person: " << cException.what() << endl;
}
```

Because pJohn is declared outside the try block, it is accessible both within the try block and the catch handlers. This means the catch handler can do cleanup properly.

The second way is to use a local variable of a class that knows how to cleanup itself when it goes out of scope. The standard library provides a class called std::auto_ptr that can be used for this

purpose. **std::auto_ptr** is a template class that holds a pointer, and deallocates it when it goes out of scope.

```
#include <memory> // for std::auto_ptr
try
{
    pJohn = new Person("John", 18, E_MALE);
    auto_ptr<Person> pxJohn(pJohn); // pxJohn now owns pJohn

    ProcessPerson(pJohn);

    // when pxJohn goes out of scope, it will delete pJohn
}
catch (PersonException &cException)
{
    cerr << "Failed to process person: " << cException.what() << endl;
}
```

Note that std::auto_ptr should not be set to point to arrays. This is because it uses the delete operator to clean up, not the delete[] operator. In fact, there is no array version of std::auto_ptr! It turns out, there isn't really a need for one. In the standard library, if you want to do dynamically allocated arrays, you're supposed to use the std::vector class, which will deallocate itself when it goes out of scope.

Exceptions and destructors

Unlike constructors, where throwing exceptions can be a useful way to indicate that object creation succeeded, exceptions should *not* be thrown in destructors.

The problem occurs when an exception is thrown from a destructor during the stack unwinding process. If that happens, the compiler is put in a situation where it doesn't know whether to continue the stack unwinding process or handle the new exception. The end result is that your program will be terminated immediately.

Consequently, the best course of action is just to abstain from using exceptions in destructors altogether. Write a message to a log file instead.

Performance concerns

Exceptions do come with a small performance price to pay. They increase the size of your executable, and they will also cause it to run slower due to the additional checking that has to be performed. However, the main performance penalty for exceptions happens when an exception is actually thrown. In this case, the stack must be unwound and an appropriate exception handler found, which is a relatively an expensive operation. Consequently, exception handling should only be used for truly exceptional cases and catastrophic errors.

The Standard Template Library

16.1 — The Standard Template Library (STL)

Congratulations! You made it all the way through the primary portion of the tutorial! In the preceding lessons, we covered all the principal C++ language features (excluding those in the C++11 extension to the language).

So the obvious question is, "what next?". One thing you've probably noticed is that an awful lot of programs use the same concepts over and over again: loops, strings, arrays, sorting, etc… You've probably also noticed that writing programs using non-class versions of containers and common algorithms are error-prone. The good news is that C++ comes with a library that is chock full of reusable classes for you to build programs out of. This library is called The C++ Standard Template Library (STL).

Although you have not have known it, you've actually been using the STL since your very first program back in lesson 0.6, when you included iostream. iostream (and our friend cout) are part of the STL!

The Standard Template Library

The Standard Template Library is a collection of classes that provide templated containers, algorithms, and iterators. If you need a common class or algorithm, odds are the STL has it. The

upside is that you can take advantage of these classes without having to write and debug the classes yourself, and the STL does a good job providing reasonably efficient versions of these classes. The downside is that the STL is complex, and can be a little intimidating since everything is templated.

Fortunately, you can bite off the STL in tiny pieces, using only what you need from it, and ignore the rest until you're ready to tackle it.

In the next few lessons, we'll take a high-level look at the types of containers, algorithms, and iterators that the STL provides. Then in subsequent lessons, we'll dig into some of the specific classes.

16.2 — STL containers overview

By far the most commonly used functionality of the STL library are the STL container classes. If you need a quick refresher on container classes, check out lesson 10.4 — Container Classes.

The STL contains many different container classes that can be used in different situations. Generally speaking, the container classes fall into three basic categories: Sequence containers, Associative containers, and Container adapters. We'll just do a quick overview of the containers here.

Sequence Containers

Sequence contains are container classes that maintain the ordering of elements in the container. A defining characteristic of sequence containers is that you can choose where to insert your element by position. The most common example of a sequence container is the array: if you insert four elements into an array, the elements will be in the exact order you inserted them.

The STL contains 3 sequence containers: vector, deque, and list.

- If you've ever taken physics, you probably are thinking of a vector as an entity with both magnitude and direction. The unfortunately named **vector** class in the STL is a dynamic array capable of growing as needed to contain its elements. The vector class allows

random access to its elements via operator[], and inserting and removing elements from the end of the vector is generally fast.

The following program inserts 6 numbers into a vector and uses the overloaded [] operator to access them in order to print them.

```
1  #include <vector>
2  #include <iostream>
3  int main()
4  {
5      using namespace std;
6
7      vector<int> vect;
8      for (int nCount=0; nCount < 6; nCount++)
9          vect.push_back(10 - nCount); // insert at end of array
10
11     for (int nIndex=0; nIndex < vect.size(); nIndex++)
12         cout << vect[nIndex] << " ";
13
14     cout << endl;
15 }
```

This program produces the result:
10 9 8 7 6 5

- The **deque** class (pronounced "deck") is a double-ended queue class, implemented as a dynamic array that can grow from both ends.

```
1  #include <iostream>
2  #include <deque>
3  int main()
4  {
5      using namespace std;
6
7      deque<int> deq;
8      for (int nCount=0; nCount < 3; nCount++)
9      {
10         deq.push_back(nCount); // insert at end of array
11         deq.push_front(10 - nCount); // insert at front of array
12     }
13
14     for (int nIndex=0; nIndex < deq.size(); nIndex++)
15         cout << deq[nIndex] << " ";
16
17     cout << endl;
18 }
```

- This program produces the result:
- 8 9 10 0 1 2
- A **list** is a special type of sequence container called a doubly linked list where each element in the container contains pointers that point at the next and previous elements in

the list. Lists only provide access to the start and end of the list — there is no random access provided. If you want to find a value in the middle, you have to start at one end and "walk the list" until you reach the element you want to find. The advantage of lists is that inserting elements into a list is very fast if you already know where you want to insert them. Generally iterators are used to walk through the list.

We'll talk more about both linked lists and iterators in future lessons.

- Although the STL **string** (and wstring) class aren't generally included as a type of sequence container, they essentially are, as they can be thought of as a vector with data elements of type char (or wchar).

Associative Containers

Associative contains are containers that automatically sort their inputs when those inputs are inserted into the container. By default, associative containers compare elements using operator<.

- A **set** is a container that stores unique elements, with duplicate elements disallowed. The elements are sorted according to their values.
- A **multiset** is a set where duplicate elements are allowed.
- A **map** (also called an associative array) is a set where each element is a pair, called a key/value pair. The key is used for sorting and indexing the data, and must be unique. The value is the actual data.
- A **multimap** (also called a dictionary) is a map that allows duplicate keys. Real-life dictionaries are multimaps: the key is the word, and the value is the meaning of the word. All the keys are sorted in ascending order, and you can look up the value by key. Some words can have multiple meanings, which is why the dictionary is a multimap rather than a map.

Container Adapters

Container adapters are special predefined containers that are adapted to specific uses. The interesting part about container adapters is that you can choose which sequence container you want them to use.

- A **stack** is a container where elements operate in a LIFO (Last In, First Out) context, where elements are inserted (pushed) and removed (popped) from the end of the container. Stacks default to using deque as their default sequence container (which seems odd, since vector seems like a more natural fit), but can use vector or list as well.
- A **queue** is a container where elements operate in a FIFO (First In, First Out) context, where elements are inserted (pushed) to the back of the container and removed (popped) from the front. Queues default to using deque, but can also use list.
- A **priority queue** is a type of queue where the elements are kept sorted (via operator<). When elements are pushed, the element is sorted in the queue. Removing an element from the front returns the highest priority item in the priority queue.

16.3 — STL iterators overview

An **Iterator** is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented. With many classes (particularly lists and the associative classes), iterators are the primary way elements of these classes are accessed.

An iterator is best visualized as a pointer to a given element in the container, with a set of overloaded operators to provide a set of well-defined functions:

- **Operator*** — Dereferencing the iterator returns the element that the iterator is currently pointing at.
- **Operator++** — Moves the iterator to the next element in the container. Most iterators also provide Operator– to move to the previous element.
- **Operator== and Operator!=** — Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, deference the iterators first, and then use a comparison operator.
- **Operator=** — Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator is point at, deference the iterator first, then use the assign operator.

Each container includes four basic member functions for use with Operator=:

- **begin()** returns an iterator representing the beginning of the elements in the container.

- **end()** returns an iterator representing the element just past the end of the elements.
- **cbegin()** returns a const (read-only) iterator representing the beginning of the elements in the container.
- **cend()** returns a const (read-only) iterator representing the element just past the end of the elements.

It might seem weird that end() doesn't point to the last element in the list, but this is done primarily to make looping easy: iterating over the elements can continue until the iterator reaches end(), and then you know you're done.

Finally, all containers provide (at least) two types of iterators:

- **container::iterator** provides a read/write iterator
- **container::const_iterator** provides a read-only iterator

Lets take a look at some examples of using iterators.

Iterating through a vector

```
1  #include <iostream>
2  #include <vector>
3  int main()
4  {
5      using namespace std;
6
7      vector<int> vect;
8      for (int nCount=0; nCount < 6; nCount++)
9          vect.push_back(nCount);
10
11     vector<int>::const_iterator it; // declare an read-only iterator
12     it = vect.begin(); // assign it to the start of the vector
13     while (it != vect.end()) // while it hasn't reach the end
14         {
15         cout << *it << " "; // print the value of the element it points to
16         it++; // and iterate to the next element
           }

       cout << endl;
   }
```

This prints the following:

0 1 2 3 4 5

Iterating through a list

Now let's do the same thing with a list:

```
1  #include <iostream>
   #include <list>
```

```
int main()
{
    using namespace std;

    list<int> li;
    for (int nCount=0; nCount < 6; nCount++)
        li.push_back(nCount);

    list<int>::const_iterator it; // declare an iterator
    it = li.begin(); // assign it to the start of the list
    while (it != li.end()) // while it hasn't reach the end
    {
        cout << *it << " "; // print the value of the element it points to
        it++; // and iterate to the next element
    }

    cout << endl;
}
```

This prints:

0 1 2 3 4 5

Note the code is almost identical to the vector case, even though vectors and lists have almost completely different internal implementations!

Iterating through a set

In the following example, we're going to create a set from 6 numbers and use an iterator to print the values in the set:

```
#include <iostream>
#include <set>
int main()
{
    using namespace std;

    set<int> myset;
    myset.insert(7);
    myset.insert(2);
    myset.insert(-6);
    myset.insert(8);
    myset.insert(1);
    myset.insert(-4);

    set<int>::const_iterator it; // declare an iterator
    it = myset.begin(); // assign it to the start of the set
    while (it != myset.end()) // while it hasn't reach the end
    {
        cout << *it << " "; // print the value of the element it points to
        it++; // and iterate to the next element
    }
```

```
19
20      cout << endl;
    }
```

This program produces the following result:

-6 -4 1 2 7 8

Note that although populating the set differs from the way we populate the vector and list, the code used to iterate through the elements of the set was essentially identical.

Iterating through a map

This one is a little trickier. Maps and multimaps take pairs of elements (defined as an stl::pair). We use the make_pair() helper function to easily create pairs. std::pair allows access to the elements of the pair via the first() and second() member functions. In our map, we use first() as the key, and second() as the value.

```
1   #include <iostream>
2   #include <map>
3   #include <string>
4   int main()
5   {
6       using namespace std;
7
8       map<int, string> mymap;
9       mymap.insert(make_pair(4, "apple"));
10      mymap.insert(make_pair(2, "orange"));
11      mymap.insert(make_pair(1, "banana"));
12      mymap.insert(make_pair(3, "grapes"));
13      mymap.insert(make_pair(6, "mango"));
14      mymap.insert(make_pair(5, "peach"));
15
16      map<int, string>::const_iterator it; // declare an iterator
17      it = mymap.begin(); // assign it to the start of the vector
18      while (it != mymap.end()) // while it hasn't reach the end
19      {
20          cout << it->first << "=" << it->second << " "; // print the value
    of the element it points to
21          it++; // and iterate to the next element
        }

        cout << endl;
    }
```

This program produces the result:

1=banana 2=orange 3=grapes 4=apple 5=peach 6=mango

Notice here how easy iterators make it to step through each of the elements of the container. You don’t have to care at all how map stores its data!

Conclusion

Iterators provide an easy way to step through the elements of a container class without having to understand how the container class is implemented. When combined with STL's algorithms and the member functions of the container classes, iterators become even more powerful. In the next lesson, you'll see an example of using an iterator to insert elements into a list (which doesn't provide an overloaded operator[] to access it's elements directly).

One point worth noting: Iterators must be implemented on a per-class basis, because the iterator does need to know how a class is implemented. Thus iterators are always tied to specific container classes.

16.4 — STL algorithms overview

In addition to container classes and iterators, STL also provides a number of generic algorithms for working with the elements of the container classes. These allow you to do things like search, sort, insert, reorder, remove, and copy elements of the container class.

Note that algorithms are implemented as global functions that operate using iterators. This means that each algorithm only needs to be implemented once, and it will generally automatically work for all containers that provides a set of iterators (including your custom container classes). While this is very powerful and can lead to the ability to write complex code very quickly, it's also got a dark side: some combination of algorithms and container types may not work, may cause infinite loops, or may work but be extremely poor performing. So use these at your risk.

STL provides quite a few algorithms — we will only touch on some of the more common and easy to use ones here. The rest (and the full details) will be saved for a chapter on STL algorithms.

To use any of the STL algorithms, simply include the algorithm header file.

min_element and max_element

The min_element and max_element algorithms find the min and max element in a container class:

```
#include <iostream>
#include <list>
#include <algorithm>
int main()
{
    using namespace std;

    list<int> li;
    for (int nCount=0; nCount < 6; nCount++)
        li.push_back(nCount);

    list<int>::const_iterator it; // declare an iterator
    it = min_element(li.begin(), li.end());
        cout << *it << " ";
    it = max_element(li.begin(), li.end());
        cout << *it << " ";

    cout << endl;
}
```

Prints:

0 5

find (and list::insert)

In this example, we'll use the find() algorithm to find a value in the list class, and then use the list::insert() function to add a new value into the list at that point.

```
#include <iostream>
#include <list>
#include <algorithm>
int main()
{
    using namespace std;

    list<int> li;
    for (int nCount=0; nCount < 6; nCount++)
        li.push_back(nCount);

    list<int>::const_iterator it; // declare an iterator
    it = find(li.begin(), li.end(), 3); // find the value 3 in the list
    li.insert(it, 8); // use list::insert to insert the value 8 before it

    for (it = li.begin(); it != li.end(); it++) // for loop with iterators
        cout << *it << " ";

    cout << endl;
}
```

This prints the value

0 1 2 8 3 4 5

sort and reverse

In this example, we'll sort a vector and then reverse it.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  int main()
5  {
6      using namespace std;
7
8      vector<int> vect;
9      vect.push_back(7);
10     vect.push_back(-3);
11     vect.push_back(6);
12     vect.push_back(2);
13     vect.push_back(-5);
14     vect.push_back(0);
15     vect.push_back(4);
16
17     sort(vect.begin(), vect.end()); // sort the list
18
19     vector<int>::const_iterator it; // declare an iterator
20     for (it = vect.begin(); it != vect.end(); it++) // for loop with
21 iterators
22         cout << *it << " ";
23
24     cout << endl;
25
26     reverse(vect.begin(), vect.end()); // reverse the list
27
28     for (it = vect.begin(); it != vect.end(); it++) // for loop with
29 iterators
30         cout << *it << " ";
31
32     cout << endl;
33 }
```

This produces the result:

-5 -3 0 2 4 6 7
7 6 4 2 0 -3 -5

Note that sort() doesn't work on list container classes — the list class provides it's own sort() member function, which is much more efficient than the generic version would be.

Conclusion

Although this is just a taste of the algorithms that STL provides, it should suffice to show how easy these are to use in conjunction with iterators and the basic container classes. There are enough other algorithms to fill up a whole chapter!

std::string

17.1 — std::string and std::wstring

The standard library contains many useful classes — but perhaps the most useful is std::string. std::string (and std::wstring) is a string class that provides many operations to assign, compare, and modify strings. In this chapter, we'll look into these string classes in depth.

Note: C-style strings will be referred to as "C-style strings", whereas std::strings (and std::wstring) will be referred to simply as "strings".

Motivation for a string class

In a previous lesson, we covered C-style strings, which using char arrays to store a string of characters. If you've tried to do anything with C-style strings, you'll very quickly come to the conclusion that they are a pain to work with, easy to mess up, and hard to debug.

C-style strings have many shortcomings, primarily revolving around the fact that you have to do all the memory management yourself. For example, if you want to assign the string "hello!" into a buffer, you have to first dynamically allocate a buffer of the correct length:

```
1 char *strHello = new char[7];
```

Don't forget to account for an extra character for the null terminator!

Then you have to actually copy the value in:

```
1 strcpy(strHello, "hello!");
```

Hopefully you made your buffer large enough so there's no buffer overflow!

And of course, because the string is dynamically allocated, you have to remember to deallocate it properly when you're done with it:

```
1 delete[] strHello;
```

Don't forget to use array delete instead of normal delete!

Furthermore, many of the intuitive operators that C provides to work with numbers, such as assignment and comparisons, simply don't work with C-style strings. Sometimes these will appear to work but actually produce incorrect results — for example, comparing two C-style strings using == will actually do a pointer comparison, not a string comparison. Assigning one C-style string to another using operator= will appear to work at first, but is actually doing a pointer copy (shallow copy), which is not generally what you want. These kinds of things can lead to program crashes that are very hard to find and debug!

The bottom line is that working with C-style strings requires remembering a lot of nit-picky rules about what is safe/unsafe, memorizing a bunch of functions that have funny names like strcat() and strcmp() instead of using intuitive operators, and doing lots of manual memory management.

Fortunately, C++ and the standard library provide a much better way to deal with strings: the std::string and std::wstring classes. By making use of C++ concepts such as constructors, destructors, and operator overloading, std::string allows you to create and manipulate strings in an intuitive and safe manner! No more memory management, no more weird function names, and a much reduced potential for disaster.

Sign me up!

String overview

All string functionality in the standard library lives in the <string> header file. To use it, simply include the string header:

```
1  #include <string>
```

There are actually 3 different string classes in the string header. The first is a templated base class named basic_string<>:

```
1  namespace std
2  {
3      template<class charT, class traits = char_traits<charT>, class Allocator
   = allocator<charT> >
4          class basic_string;
5  }
```

You won't be working with this class directly, so don't worry about what traits or an Allocator is for the time being. The default values will suffice in almost every imaginable case.

There are two specializations of basic_string<> provided by the standard library:

```
1  namespace std
2  {
3      typedef basic_string<char> string;
4      typedef basic_string<wchar_t> wstring;
   }
```

These are the two classes that you will actually use. std::string is used for standard ascii (utf-8) strings. std::wstring is used for wide-character/unicode (utf-16) strings. There is no built-in class for utf-32 strings (though you should be able to extend your own from basic_string<> if you need one).

Although you will directly use std::string and std::wstring, it turns out that all of the string functionality is actually implemented in the basic_string<> class and then inherited by string and wstring. Consequently, all of the functions presented will work for both string and wstring.

However, because basic_string is a templated class, it also means the compiler will produce horrible looking template errors when you do something syntactically incorrect with a string or wstring. Don't be intimidated by these errors; they look far worse than they are!

Here's a list of all the functions in the string class. Most of these functions have multiple flavors to handle different types of inputs, which we will cover in more depth in the next lessons.

Function	Effect
Creation and destruction	
(constructor) (destructor)	Create or copy a string Destroy a string
Size and capacity	
capacity() empty() length(), size() max_size() reserve()	Returns the number of characters that can be held without reallocation Returns a boolean indicating whether the string is empty Returns the number of characters in string Returns the maximum string size that can be allocated Expand or shrink the capacity of the string
Element access	
[], at()	Accesses the character at a particular index
Modification	
=, assign() +=, append(), push_back() insert() clear() erase() replace() resize() swap()	Assigns a new value to the string Concatenates characters to end of the string Inserts characters at an arbitrary index in string Delete all characters in the string Erase characters at an arbitrary index in string Replace characters at an arbitrary index with other characters Expand or shrink the string (truncates or adds characters at end of string) Swaps the value of two strings
Input and Output	
>>, getline() << c_str copy() data()	Reads values from the input stream into the string Writes string value to the output stream Returns the contents of the string as a NULL-terminated C-style string Copies contents (not NULL-terminated) to a character array Returns the contents of the string as a non-NULL-terminated character array
String comparison	
==, != <, <=, > >= compare()	Compares whether two strings are equal/unequal (returns bool) Compares whether two strings are less than / greater than each other (returns bool) Compares whether two strings are equal/unequal (returns -1, 0, or 1)

Substrings and concatenation	
+	Concatenates two strings
substr()	Returns a substring
Searching	
find	Find index of first character/substring
find_first_of	Find index of first character from a set of characters
find_first_not_of	Find index of first character not from a set of characters
find_last_of	Find index of last character from a set of characters
find_last_not_of	Find index of last character not from a set of characters
rfind	Find index of last character/substring
Iterator and allocator support	
begin(), end()	Forward-direction iterator support for beginning/end of string
get_allocator()	Returns the allocator
rbegin(), rend()	Reverse-direction iterator support for beginning/end of string

Note: The above table will look funny if your browser is too narrow

While the standard library string classes provide a lot of functionality, there are a few notable omissions:

- Regular expression support
- Constructors for creating strings from numbers
- Capitalization / upper case / lower case functions
- Case-insensitive comparisons
- Tokenization / splitting string into array
- Easy functions for getting the left or right hand portion of string
- Whitespace trimming
- Formatting a string sprintf style
- Conversion from utf-8 to utf-16 or vice-versa

For most of these, you will have to either write your own functions, or convert your string to a C-style string (using c_str()) and use the C functions that offer this functionality.

In the next lessons, we will look at the various functions of the string class in more depth. Although we will use string for our examples, everything is equally applicable to wstring.

17.2 — std::string construction and destruction

In this lesson, we'll take a look at how to construct objects of std::string, as well as how to create strings from numbers and vice-versa.

String construction

The string classes have a number of constructors that can be used to create strings. We'll take a look at each of them here.

Note: string::size_type resolves to size_t, which is the same unsigned integral type that is returned by the sizeof operator. It's actual size varies depending on environment. For purposes of this tutorial, envision it as an unsigned int.

string::string()

- This is the default constructor. It creates an empty string.

Sample code:

```
1 std::string sSource;
2 cout << sSource;
```

Output:

string::string(const string& strString)

- This is the copy constructor. This constructor creates a new string as a copy of strString.

Sample code:

```
1 string sSource("my string");
2 string sOutput(sSource);
3 cout << sOutput;
```

Output:

```
my string
```

string::string(const string& strString, size_type unIndex)
string::string(const string& strString, size_type unIndex, size_type unLength)

- This constructor creates a new string that contains at most unLength characters from strString, starting with index unIndex. If a NULL is encountered, the string copy will end, even if unLength has not been reached.
- If no unLength is supplied, all characters starting from unIndex will be used.
- If unIndex is larger than the size of the string, the out_of_range exception will be thrown.

Sample code:

```
1  string sSource("my string");
2  string sOutput(sSource, 3);
3  cout << sOutput<< endl;
4  string sOutput2(sSource, 3, 4);
   cout << sOutput2 << endl;
```

Output:

```
string
stri
```

string::string(const char *szCString)

- This constructor creates a new string from the C-style string szCString, up to but not including the NULL terminator.
- If the resulting size exceeds the maximum string length, the length_error exception will be thrown.
- **Warning:** szCString must not be NULL.

Sample code:

```
1  const char *szSource("my string");
2  string sOutput(szSource);
3  cout << sOutput << endl;
```

Output:

```
my string
```

string::string(const char *szCString, size_type unLength)

- This constructor creates a new string from the first unLength chars from the C-style string szCString.
- If the resulting size exceeds the maximum string length, the length_error exception will be thrown.
- **Warning:** For this function only, NULLs are not treated as end-of-string characters in szCString! This means it is possible to read off the end of your string if unLength is too big. Be careful not to overflow your string buffer!

Sample code:

```
1  const char *szSource("my string");
2  string sOutput(szSource, 4);
3  cout << sOutput << endl;
```

Output:

```
my s
```

string::string(size_type nNum, char chChar)

- This constructor creates a new string initialized by nNum occurances of the character chChar.
- If the resulting size exceeds the maximum string length, the length_error exception will be thrown.

Sample code:

```
1  string sOutput(4, 'Q');
2  cout << sOutput << endl;
```

Output:

```
QQQQ
```

template<class InputIterator> string::string(InputIterator itBeg, InputIterator itEnd)

- This constructor creates a new string initialized by the characters of range [itBeg, itEnd).
- If the resulting size exceeds the maximum string length, the length_error exception will be thrown.

No sample code for this one. It's obscure enough you'll probably never use it.

string::~string()

String construction

- This is the destructor. It destroys the string and frees the memory.

No sample code here either since the destructor isn't called explicitly.

Constructing strings from numbers

One notable omission in the std::string class is the lack of ability to create strings from numbers. For example:

```
1  string sFour(4);
```

Produces the following error:

```
c:\vcprojects\test2\test2\test.cpp(10) : error C2664:
'std::basic_string<_Elem,_Traits,_Ax>::basic_string(std::basic_string<_Elem,_
Traits,_Ax>::_Has_debug_it)' : cannot convert parameter 1 from 'int' to
'std::basic_string<_Elem,_Traits,_Ax>::_Has_debug_it'
```

Remember what I said about the string classes producing horrible looking errors? The relevant bit of information here is:

```
cannot convert parameter 1 from 'int' to 'std::basic_string
```

In other words, it tried to convert your int into a string but failed.

The easiest way to convert numbers into strings is to involve the std::ostringstream class. std::ostringstream is already set up to accept input from a variety of sources, including characters, numbers, strings, etc… It is also capable of outputting strings (either via the extraction operator>>, or via the str() function). For more information on std::ostringstream, see Lesson 13.4 — Stream classes for strings.

Here's a simple solution for creating std::string from various types of inputs:

```
1  #include <iostream>
2  #include <sstream>
3  #include <string>
4
5  template <typename T>
6  inline std::string ToString(T tX)
7  {
8      std::ostringstream oStream;
9      oStream << tX;
10     return oStream.str();
11 }
```

Here's some sample code to test it:

```
1  int main()
2  {
3      string sFour(ToString(7));
4      string sSixPointSeven(ToString(6.7));
5      string sA(ToString('A'));
6      cout << sFour << endl;
7      cout << sSixPointSeven << endl;
8      cout << sA << endl;
9  }
```

And the output:

```
4
```

```
6.7
A
```

Note that this solution omits any error checking. It is possible that inserting tX into oStream could fail. An appropriate response would be to throw an exception if the conversation fails.

Converting strings to numbers

Similar to the solution above:

```
1  #include <iostream>
2  #include <sstream>
3  #include <string>
4
5  template <typename T>
6  inline bool FromString(const std::string& sString, T &tX)
7  {
8      std::istringstream iStream(sString);
9      return (iStream >> tX) ? true : false;
10 }
```

Here's some sample code to test it:

```
1  int main()
2  {
3      double dX;
4      if (FromString("3.4", dX))
5          cout << dX << endl;
6      if (FromString("ABC", dX))
7          cout << dX << endl;
8  }
```

And the output:

```
3.4
```

Note that the second conversation failed and returned false.

17.3 — std::string length and capacity

Once you've created strings, it's often useful to know how long they are. This is where length and capacity operations come into play. We'll also discuss various ways to convert std::string back into C-style strings, so you can use them with functions that expect strings of type char*.

Length of a string

The length of the string is quite simple — it's the number of characters in the string. There are two identical functions for determining string length:

size_type string::length() const
size_type string::size() const

- Both of these functions return the current number of characters in the string, excluding the null terminator.

Sample code:

```
1 string sSource("012345678");
2 cout << sSource.length() << endl;
```

Output:

```
9
```

Although it's possible to use length() to determine whether a string has any characters or not, it's more efficient to use the empty() function:

bool string::empty() const

- Returns true if the string has no characters, false otherwise.

Sample code:

```
1 string sString1("Not Empty");
2 cout << (sString1.empty() ? "true" : "false") << endl;
3 string sString2; // empty
  cout << (sString2.empty() ? "true" : "false")  << endl;
```

Output:

```
false
```

```
true
```

There is one more size-related function that you will probably never use, but we'll include it here for completeness:

size_type string::max_size() const

- Returns the maximum number of characters that a string is allowed to have.
- This value will vary depending on operating system and system architecture.

Sample code:

```
1  string sString("MyString");
2  cout << sString.max_size() << endl;
```

Output:

```
4294967294
```

Capacity of a string

The capacity of a string reflects how much memory the string allocated to hold it's contents. This value is measured in string characters, excluding the NULL terminator. For example, a string with capacity 8 could hold 8 characters.

size_type string::capacity() const

- Returns the number of characters a string can hold without reallocation.

Sample code:

```
1  string sString("01234567");
2  cout << "Length: " << sString.length() << endl;
3  cout << "Capacity: " << sString.capacity() << endl;
```

Output:

```
Length: 8
Capacity: 15
```

Note that the capacity is higher than the length of the string! Although our string was length 8, the string actually allocated enough memory for 15 characters! Why was this done?

The important thing to recognize here is that if a user wants to put more characters into a string than the string has capacity for, the string has to be reallocated to a larger capacity. For example,

if a string had both length and capacity of 8, then adding any characters to the string would force a reallocation. By making the capacity larger than the actual string, this gives the user some buffer room to expand the string before reallocation needs to be done.

As it turns out, reallocation is bad for several reasons:

First, reallocating a string is comparatively expensive. First, new memory has to be allocated. Then each character in the string has to be copied to the new memory. This can take a long time if the string is big. Finally, the old memory has to be deallocated. If you are doing many reallocations, this process can slow your program down significantly.

Second, whenever a string is reallocated, the contents of the string change to a new memory address. This means all references, pointers, and iterators to the string become invalid!

Note that it's not always the case that strings will be allocated with capacity greater than length. Consider the following program:

```
1  string sString("0123456789abcde");
2  cout << "Length: " << sString.length() << endl;
3  cout << "Capacity: " << sString.capacity() << endl;
```

This program outputs:

```
Length: 15
Capacity: 15
```

(Results may vary depending on compiler).

Let's add one character to the string and watch the capacity change:

```
1  string sString("0123456789abcde");
2  cout << "Length: " << sString.length() << endl;
3  cout << "Capacity: " << sString.capacity() << endl;
4
5  // Now add a new character
6  sString += "f";
7  cout << "Length: " << sString.length() << endl;
8  cout << "Capacity: " << sString.capacity() << endl;
```

This produces the result:

```
Length: 15
Capacity: 15
Length: 16
Capacity: 31
```

void string::reserve()
void string::reserve(size_type unSize)

- The second flavor of this function sets the capacity of the string to at least unSize (it can be greater). Note that this may require a reallocation to occur.
- If the first flavor of the function is called, or the second flavor is called with unSize less than the current capacity, the function will try to shrink the capacity to match the length. This is a non-binding request.

Sample code:

```
1  string sString("01234567");
2  cout << "Length: " << sString.length() << endl;
3  cout << "Capacity: " << sString.capacity() << endl;
4
5  sString.reserve(200);
6  cout << "Length: " << sString.length() << endl;
7  cout << "Capacity: " << sString.capacity() << endl;
8
9  sString.reserve();
   cout << "Length: " << sString.length() << endl;
   cout << "Capacity: " << sString.capacity() << endl;
```

Output:

```
Length: 8
Capacity: 15
Length: 8
Capacity: 207
Length: 8
Capacity: 207
```

This example shows two interesting things. First, although we requested a capacity of 200, we actually got a capacity of 207. The capacity is always guaranteed to be at least as large as your request, but may be larger. We then requested the capacity change to fit the string. This request was ignored, as the capacity did not change.

If you know in advance that you're going to be constructing a large string by doing lots of string operations that will add to the size of the string, you can avoid having the string reallocated multiple times by immediately setting the string to it's final capacity:

```
1  #include <iostream>
2  #include <string>
3  #include <cstdlib> // for rand() and srand()
4  #include <ctime> // for time()
5
6  using namespace std;
7
8  int main()
9  {
       srand(time(0)); // seed random number generator
```

```
10      string sString; // length 0
11      sString.reserve(64); // reserve 64 characters
12
13      // Fill string up with random lower case characters
14      for (int nCount=0; nCount < 64; ++nCount)
15          sString += 'a' + rand() % 26;
16
17      cout << sString;
}
```

The result of this program will change each time, but here's the output from one execution:

```
wzpzujwuaokbakgijqdawvzjqlgcipiiuuxhyfkdppxpyycvytvyxwqsbtielxpy
```

Rather than having to reallocate sString multiple times, we set the capacity once and then fill the string up. This can make a huge difference in performance when constructing large strings via concatenation.

17.4 — std::string character access and conversion to C-style arrays

Character access

There are two almost identical ways to access characters in a string. The easier to use and faster version is the overloaded operator[]:

char& string::operator[] (size_type nIndex)
const char& string::operator[] (size_type nIndex) const

- Both of these functions return the character with index nIndex
- Passing an invalid index results in undefined behavior
- Using length() as the index is valid for const strings only, and returns the value generated by the string's default constructor. It is not recommended that you do this.
- Because char& is the return type, you can use this to edit characters in the array

Sample code:

```
1  string sSource("abcdefg");
2  cout << sSource[5] << endl;
3  sSource[5] = 'X';
4  cout << sSource << endl;
```

Output:

```
f
abcdeXg
```

There is also a non-operator version. This version is slower since it uses exceptions to check if the nIndex is valid. If you are not sure whether nIndex is valid, you should use this version to access the array:

char& string::at (size_type nIndex)
const char& string::at (size_type nIndex) const

- Both of these functions return the character with index nIndex
- Passing an invalid index results in an out_of_range exception
- Because char& is the return type, you can use this to edit characters in the array

Sample code:

```
1  string sSource("abcdefg");
2  cout << sSource.at(5) << endl;
3  sSource.at(5) = 'X';
4  cout << sSource << endl;
```

Output:

```
f
abcdeXg
```

Conversion to C-style arrays

Many functions (including all C functions) expect strings to be formatted as C-style strings rather than std::string. For this reason, std::string provides 3 different ways to convert std::string to C-style strings.

const char* string::c_str () const

- Returns the contents of the string as a const C-style string
- A null terminator is appended
- The C-style string is owned by the std::string and should not be deleted

Sample code:

```
1  string sSource("abcdefg");
2  cout << strlen(sSource.c_str());
```

Output:

```
7
```

const char* string::data () const

- Returns the contents of the string as a const C-style string
- A null terminator is **not** appended
- The C-style string is owned by the std::string and should not be deleted

Sample code:

```
1  string sSource("abcdefg");
2  char *szString = "abcdefg";
3  // memcmp compares the first n characters of two C-style strings and
   returns 0 if they are equal
4  if (memcmp(sSource.data(), szString, sSource.length()) == 0)
5      cout << "The strings are equal";
```

```
else
    cout << "The strings are not equal";
```

Output:

```
The strings are equal
```

size_type string::copy(char *szBuf, size_type nLength) const
size_type string::copy(char *szBuf, size_type nLength, size_type nIndex) const

- Both flavors copy at most nLength characters of the string to szBuf, beginning with character nIndex
- The number of characters copied is returned
- No null is appended. It is up to the caller to ensure szBuf is initialized to NULL or terminate the string using the returned length
- The caller is responsible for not overflowing szBuf

Sample code:

```
string sSource("sphinx of black quartz, judge my vow");

char szBuf[20];
int nLength = sSource.copy(szBuf, 5, 10);
szBuf[nLength]='\0';  // Make sure we terminate the string in the buffer

cout << szBuf << endl;
```

Output:

```
black
```

Unless you need every bit of efficiency, c_str() is the easiest and safest of the three functions to use.

17.5 — std::string assignment and swapping

String assignment

The easiest way to assign a value to a string is to the use the overloaded operator= function. There is also an assign() member function that duplicates some of this functionality.

string& string::operator= (const string& str)
string& string::assign (const string& str)
string& string::operator= (const char* str)
string& string::assign (const char* str)
string& string::operator= (char c)

- These functions assign values of various types to the string.
- These functions return *this so they can be "chained".
- Note that there is no assign() function that takes a single char.

Sample code:

```
1  string sString;
2
3  // Assign a string value
4  sString = string("One");
5  cout << sString << endl;
6
7  const string sTwo("Two");
8  sString.assign(sTwo);
9  cout << sString << endl;
10
11 // Assign a C-style string
12 sString = "Three";
13 cout << sString << endl;
14
15 sString.assign("Four");
16 cout << sString << endl;
17
18 // Assign a char
19 sString = '5';
20 cout << sString << endl;
21
22 // Chain assignment
23 string sOther;
24 sString = sOther = "Six";
25 cout << sString << " " << sOther << endl;
```

Output:

```
One
Two
Three
Four
5
Six Six
```

The assign() member function also comes in a few other flavors:

string& string::assign (const string& str, size_type index, size_type len)

- Assigns a substring of str, starting from index, and of length len
- Throws an out_of_range exception if the index is out of bounds
- Returns *this so it can be "chained".

Sample code:

```
1  const string sSource("abcdefg");
2  string sDest;
3
4  sDest.assign(sSource, 2, 4); // assign a substring of source from index 2
   of length 4
5  cout << sDest << endl;
```

Output:

```
cdef
```

string& string::assign (const char* chars, size_type len)

- Assigns len characters from the C-style array chars
- Ignores special characters (including '\0')
- Throws an length_error exception if the result exceeds the maximum number of characters
- Returns *this so it can be "chained".

Sample code:

```
1  string sDest;
2
3  sDest.assign("abcdefg", 4);
4  cout << sDest << endl;
```

Output:

```
abcd
```

This function is potentially dangerous and it's use is not recommended.

string& string::assign (size_type len, char c)

- Assigns len occurrences of the character c
- Throws an length_error exception if the result exceeds the maximum number of characters
- Returns *this so it can be "chained".

Sample code:

```
1  string sDest;
2
3  sDest.assign(4, 'g');
4  cout << sDest << endl;
```

Output:

```
gggg
```

Swapping

If you have two strings and want to swap their values, there are two functions both named swap() that you can use.

void string::swap (string &str)
void swap (string &str1, string &str2)

- Both functions swap the value of the two strings. The member function swaps *this and str, the global function swaps str1 and str2.
- These functions are efficient and should be used instead of assignments to perform a string swap.

Sample code:

```
1  string sStr1("red");
2  string sStr2("blue");
3
4  cout << sStr1 << " " << sStr2 << endl;
5  swap(sStr1, sStr2);
6  cout << sStr1 << " " << sStr2 << endl;
7  sStr1.swap(sStr2);
8  cout << sStr1 << " " << sStr2 << endl;
```

Output:

```
red blue
blue red
red blue
```

17.6 — std::string appending

Appending

Appending strings to the end of an existing string is easy using either operator+=, append(), or push_back() function.

string& string::operator+= (const string& str)
string& string::append (const string& str)

- Both functions append the characters of str to the string.
- Both function return *this so they can be "chained".
- Both functions throw a length_error exception if the result exceeds the maximum number of characters.

Sample code:

```
1  string sString("one");
2
3  sString += string(" two");
4
5  string sThree(" three");
6  sString.append(sThree);
7
8  cout << sString << endl;
```

Output:

```
one two three
```

There's also a flavor of append() that can append a substring:

string& string::append (const string& str, size_type index, size_type num)

- This function appends num characters from str, starting at index, to the string.
- Returns *this so it can be "chained".
- Throws an out_of_range if index is out of bounds
- Throws a length_error exception if the result exceeds the maximum number of characters.

Sample code:

```
1  string sString("one ");
2
3  const string sTemp("twothreefour");
4  sString.append(sTemp, 3, 5); // append substring of sTemp starting at
   index 3 of length 5
5  cout << sString << endl;
```

Output:

```
one three
```

Operator+= and append() also have versions that work on C-style strings:

string& string::operator+= (const char* str)
string& string::append (const char* str)

- Both functions append the characters of str to the string.
- Both function return *this so they can be "chained".
- Both functions throw a length_error exception if the result exceeds the maximum number of characters.
- str should not be NULL.

Sample code:

```
1  string sString("one");
2
3  sString += " two";
4  sString.append(" three");
   cout << sString << endl;
```

Output:

```
one two three
```

There is an additional flavor of append() that works on C-style strings:

string& string::append (const char* str, size_type len)

- Appends the first len characters of str to the string.
- Returns *this so they can be "chained".
- Throw a length_error exception if the result exceeds the maximum number of characters.
- Ignores special characters (including '\0')

Sample code:

```
1  string sString("one");
```

```
2 sString.append("threefour", 5);
3 cout << sString << endl;
4
```

Output:

```
one three
```

This function is dangerous and its use is not recommended.

There is also a set of functions that append characters. Note that the name of the non-operator function to append a character is push_back(), not append()!

string& string::operator+= (char c)
void string::push_back (char c)

- Both functions append the the character c to the string.
- Operator += returns *this so it can be "chained".
- Both functions throw a length_error exception if the result exceeds the maximum number of characters.

Sample code:

```
1 string sString("one");
2
3 sString += ' ';
4 sString.append('2');
5 cout << sString << endl;
```

Output:

```
one 2
```

Now you might be wondering why the name of the function is push_back() and not append(). This follows a naming convention used for stacks, where push_back() is the function that adds a single item to the end of the stack. If you envision a string as a stack of characters, using push_back() to add a single character to the end makes sense. However, the lack of an append() function is inconsistent in my view!

It turns out there is an append() function for characters, that looks like this:

string& string::append (size_type num, char c)

- Adds num occurrences of the character c to the string
- Returns *this so it can be "chained".

- Throws a length_error exception if the result exceeds the maximum number of characters.

Sample code:

```
1  string sString("aaa");
2
3  sString.append(4, 'b');
4  cout << sString << endl;
```

Output:

```
aaabbbb
```

There's one final flavor of append() that you won't understand unless you know what iterators are. If you're not familiar with iterators, you can ignore this function.

string& string::append (InputIterator start, InputIterator end)

- Appends all characters from the range [start, end) (including start up to but not including end)
- Returns *this so it can be "chained".
- Throws a length_error exception if the result exceeds the maximum number of characters.

17.7 — std::string inserting

Inserting

Inserting characters into an existing string can be done via the insert() function.

string& string::insert (size_type index, const string& str)
string& string::insert (size_type index, const char* str)

- Both functions insert the characters of str into the string at index
- Both function return *this so they can be "chained".
- Both functions throw out_of_range if index is invalid
- Both functions throw a length_error exception if the result exceeds the maximum number of characters.
- In the C-style string version, str must not be NULL.

Sample code:

```
1  string sString("aaaa");
2  cout << sString << endl;
3
4  sString.insert(2, string("bbbb"));
5  cout << sString << endl;
6
7  sString.insert(4, "cccc");
   cout << sString << endl;
```

Output:

```
aaaa
aabbbbaa
aabbccccbbaa
```

Here's a crazy version of insert() that allows you to insert a substring into a string at an arbitrary index:

string& string::insert (size_type index, const string& str, size_type startindex, size_type num)

- This function inserts num characters str, starting from startindex, into the string at index.
- Returns *this so it can be "chained".
- Throws an out_of_range if index or startindex is out of bounds
- Throws a length_error exception if the result exceeds the maximum number of

characters.

Sample code:

```
1  string sString("aaaa");
2
3  const string sInsert("01234567");
4  sString.insert(2, sInsert, 3, 4); // insert substring of sInsert from
   index [3,7) into sString at index 2
5  cout << sString << endl;
```

Output:

```
aa3456aa
```

There is a flavor of insert() that inserts the first portion of a C-style string:

string& string::insert(size_type index, const char* str, size_type len)

- Inserts len characters of str into the string at index
- Returns *this so it can be "chained".
- Throws an out_of_range exception if the index is invalid
- Throws a length_error exception if the result exceeds the maximum number of characters.
- Ignores special characters (such as '\0')

Sample code:

```
1  string sString("aaaa");
2
3  sString.insert(2, "bcdef", 3);
4  cout << sString << endl;
```

Output:

```
aabcdaa
```

There's also a flavor of insert() that inserts the same character multiple times:

string& string::insert(size_type index, size_type num, char c)

- Inserts num instances of char c into the string at index
- Returns *this so it can be "chained".
- Throws an out_of_range exception if the index is invalid
- Throws a length_error exception if the result exceeds the maximum number of

characters.

Sample code:

```
1  string sString("aaaa");
2
3  sString.insert(2, 4, 'c');
4  cout << sString << endl;
```

Output:

```
aaccccaa
```

And finally, the insert() function also has three different versions that use iterators:

void insert(iterator it, size_type num, char c)
iterator string::insert(iterator it, char c)
void string::insert(iterator it, InputIterator begin, InputIterator end)

- The first function inserts num instances of the character c before the iterator it.
- The second inserts a single character c before the iterator it, and returns an iterator to the position of the character inserted.
- The third inserts all characters between [begin,end) before the iterator it.
- All functions throw a length_error exception if the result exceeds the maximum number of characters.

Author : Hossein Hezami (3v1l)

E-mail : B.Devils.B@Gmail.com , Black_Devils.B0ys@Yahoo.com

Y! : Teacher_3v1l

Black_Devils B0ys Digital Network Security Group

Date : 2013

Special Tnx to Best My Friend Alex. :x

For More Information Visit : www.LearnCpp.com

www.SoftGozar.Com