Introduction to C++

Objectives:

- Learn why C++ is the emerging standard in software development.
- Learn the steps to develop a C++ program.
- Learn how to enter, compile, and link your first working C++ program.

A Brief History of C++

Computer languages have undergone dramatic evolution since the first electronic computers were built to assist in telemetry calculations during World War II. Early on, programmers worked with the most primitive computer instructions: machine language. These instructions were represented by long strings of ones and zeroes. Soon, assemblers were invented to map machine instructions to human-readable and -manageable mnemonics, such as `ADD` and `MOV`.

In time, higher-level languages evolved, such as BASIC and COBOL. These languages let people work with something approximating words and sentences, such as `Let I = 100`. These instructions were translated back into machine language by interpreters and compilers. An interpreter translates a program as it reads it, turning the program instructions, or code, directly into actions. A compiler translates the code into an intermediary form. This step is called compiling, and produces an object file. The compiler then invokes a linker, which turns the object file into an executable program.

Because interpreters read the code as it is written and execute the code on the spot, interpreters are easy for the programmer to work with. Compilers, however, introduce the extra steps of compiling and linking the code, which is inconvenient. Compilers produce a program that is very fast each time it is run. However, the time-consuming task of translating the source code into machine language has already been accomplished.

Another advantage of many compiled languages like C++ is that you can distribute the executable program to people who don't have the compiler. With an interpretive language, you must have the language to run the program.

For many years, the principle goal of computer programmers was to write short pieces of code that would execute quickly. The program needed to be small, because memory was expensive, and it needed to be fast, because processing power was also expensive. As computers have become smaller, cheaper, and faster, and as the cost of memory has fallen, these priorities have changed. Today the cost of a programmer's time far outweighs the cost of most of the computers in use by businesses. Well-written, easy-to-maintain code is at a premium. Easy- to-maintain means that as business requirements change, the program can be extended and enhanced without great expense.

Today, there are 5 generations of programming language. Refer to the following table:

Generation	Type	Examples
5GL	Logic & Intelligence Languages	Prolog, LISP
4GL	Interpreted Languages	Visual Basic, most of the DBMS-related languages
3GL	Structure Languages	Fortran, C, C++, Pascal, Modula-2
2GL	Assembly Languages	Intel x86 and Motorola 68000 assembly languages
1GL	Machine Language	Intel x86 and Motorola 68000 machine codes

Among all these languages, the computers can understand only the 1GL directly. The code is called ***Native Code***.

Programs

The word program is used in two ways: to describe individual instructions, or source code, created by the programmer, and to describe an entire piece of executable software. This distinction can cause enormous confusion, so we will try to distinguish between the source code on one hand, and the executable on the other.

New Term: A *program* can be defined as either a set of written instructions created by a programmer or an executable piece of software.

Source code can be turned into an executable program in two ways: Interpreters translate the source code into computer instructions, and the computer acts on those instructions immediately. Alternatively, compilers translate source code into a program, which you can run at a later time. While interpreters are easier to work with, most serious programming is done with compilers because compiled code runs much faster. C++ is a compiled language.

Solving Problems

The problems programmers are asked to solve have been changing. Twenty years ago, programs were created to manage large amounts of raw data. The people writing the code and the people using the program were all computer professionals. Today, computers are in use by far more people, and most know very little about how computers and programs work. Computers are tools used by people who are more interested in solving their business problems than struggling with the computer.

Ironically, in order to become easier to use for this new audience, programs have become far more sophisticated. Gone are the days when users typed in cryptic commands at esoteric prompts, only to see a stream of raw data. Today's programs use sophisticated "user-friendly interfaces," involving multiple windows, menus, dialog boxes, and the myriad of metaphors with which we've all become familiar. The programs written to support this new approach are far more complex than those written just ten years ago.

As programming requirements have changed, both languages and the techniques used for writing programs have evolved. While the complete history is fascinating, this course will focus on the transformation from procedural programming to object-oriented programming.

Procedural, Structured, and Object-Oriented Programming

Until recently, programs were thought of as a series of procedures that acted upon data. A procedure, or function, is a set of specific instructions executed one after the other. The data was quite separate from the procedures, and the trick in programming was to keep track of which functions called which other functions, and what data was changed. To make sense of this potentially confusing situation, structured programming was created.

The principle idea behind structured programming is as simple as the idea of divide and conquer. A computer program can be thought of as consisting of a set of tasks. Any task that is too complex to be described simply would be broken down into a set of smaller component tasks, until the tasks were sufficiently small and self-contained enough that they were easily understood.

As an example, computing the average salary of every employee of a company is a rather complex task. You can, however, break it down into these subtasks:

1. Find out what each person earns.
2. Count how many people you have.
3. Total all the salaries.
4. Divide the total by the number of people you have.

Totaling the salaries can be broken down into

1. Get each employee's record.
2. Access the salary.
3. Add the salary to the running total.
4. Get the next employee's record.

In turn, obtaining each employee's record can be broken down into

1. Open the file of employees.
2. Go to the correct record.
3. Read the data from disk.

Structured programming remains an enormously successful approach for dealing with complex problems. By the late 1980s, however, some of the deficiencies of structured programming had become all too clear.

First, it is natural to think of your data (employee records, for example) and what you can do with your data (sort, edit, and so on) as related ideas.

Second, programmers found themselves constantly reinventing new solutions to old problems. This is often called "*reinventing the wheel*", and is the opposite of reusability. The idea behind reusability is to build components that have known properties, and then to be able to plug them into your program, as you need them. This is modeled after the hardware world--when an engineer needs a new transistor, he doesn't usually invent one, he goes to the big bin of transistors and finds one that works the way he needs it to, or perhaps modifies it. There was no similar option for a software engineer.

New Term: The way we are now using computers--with menus and buttons and windows--fosters a more interactive, event-driven approach to computer programming. *Event-driven* means that an event happens--the user presses a button or chooses from a menu--and the program must respond. Programs are becoming increasingly interactive, and it has became important to design for that kind of functionality.

Old-fashioned programs forced the user to proceed step-by-step through a series of screens. Modern event-driven programs present all the choices at once and respond to the user's actions.

Object-oriented programming attempts to respond to these needs, providing techniques for managing enormous complexity, achieving reuse of software components, and coupling data with the tasks that manipulate that data.

The essence of object-oriented programming is to treat data and the procedures that act upon the data as a single ***object***--a self-contained entity with an identity and certain characteristics of its own.

C++ and Object-Oriented Programming

C++ fully supports object-oriented programming, including the four pillars of object-oriented development: encapsulation, data hiding, inheritance, and polymorphism. Encapsulation and Data Hiding When an engineer needs to add a resistor to the device he is creating, he doesn't typically build a new one from scratch. He walks over to a bin of resistors, examines the colored bands that indicate the properties, and picks the one he needs. The resistor is a "*black box*" as far as the engineer is concerned—he doesn't much care how it does its work as long as it conforms to his specifications; he doesn't need to look inside the box to use it in his design.

The property of being a self-contained unit is called encapsulation. With encapsulation, we can accomplish data hiding. Data hiding is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally. Just as you can use a refrigerator without knowing how the compressor works, you can use a well-designed object without knowing about its internal data members.

Similarly, when the engineer uses the resistor, he need not know anything about the internal state of the resistor. All the properties of the resistor are encapsulated in the resistor object; they are not spread out through the circuitry. It is not necessary to understand how the resistor works in order to use it effectively. Its data is hidden inside the resistor's casing.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. Once created, a well-defined class act as a fully encapsulated entity--it is used as a whole unit. The actual inner workings of the class should be hidden. Users of a well-defined class do not need to know how the class works; they just need to know how to use it. Inheritance and Reuse When the engineers at Acme Motors want to build a new car, they have two choices: They can start from scratch, or they can modify an existing model. Perhaps their Star model is nearly perfect, but they'd like to add a turbocharger and a six-speed transmission. The chief engineer would prefer not to start from the ground up, but rather to say, "*Let's build another Star, but let's add these additional*

capabilities. We'll call the new model a Quasar." A Quasar is a kind of Star, but one with new features.

C++ supports the idea of reuse through inheritance. A new type, which is an extension of an existing type, can be declared. This new subclass is said to derive from the existing type and is sometimes called a derived type. The Quasar is derived from the Star and thus inherits all its qualities, but can add to them as needed. Polymorphism - The new Quasar might respond differently than a Star does when you press down on the accelerator. The Quasar might engage fuel injection and a turbocharger, while the Star would simply let gasoline into its carburetor. A user, however, does not have to know about these differences. He can just "floor it" and the right thing will happen, depending on which car he's driving.

C++ supports the idea that different objects do "*the right thing*" through what is called function polymorphism and class polymorphism. Poly means many, and morph means form. Polymorphism refers to the same name taking many forms

Please refer to the appendices for more explanation of Object-oriented terminologies.

How C++ Evolved

As object-oriented analysis, design, and programming began to catch on, Bjarne Stroustrup took the most popular language for commercial software development, C, and extended it to provide the features needed to facilitate object-oriented programming. He created C++, and in less than a decade it has gone from being used by only a handful of developers at AT&T to being the programming language of choice for an estimated one million developers worldwide.

While it is true that C++ is a superset of C, and that virtually any legal C program is a legal C++ program, the leap from C to C++ is very significant. C++ benefited from its relationship to C for many years, as C programmers could ease into their use of C++. To really get the full benefit of C++, however, many programmers found they had to unlearn much of what they knew and learn a whole new way of conceptualizing and solving programming problems.

The ANSI Standard

The Accredited Standards Committee, operating under the procedures of the American National Standards Institute (ANSI), is working to create an international standard for C++.

The draft of this standard has been published, and a link is available at `www.libertyassociates.com`.

The ANSI standard is an attempt to ensure that C++ is portable--that code you write for Microsoft's compiler will compile without errors, using a compiler from any other vendor. Further, because the code in this course is ANSI compliant, it should compile without errors on a Mac, a Windows box, or an Alpha.

For most students of C++, the ANSI standard will be invisible. The standard has been stable for a while, and all the major manufacturers support the ANSI standard. We have endeavored to ensure that all the code in this edition of this course is ANSI compliant.

Should I Learn C First?

The question inevitably arises: "*Since C++ is a superset of C, should I learn C first?*" Stroustrup and most other C++ programmers agree. Not only is it unnecessary to learn C first, it may be advantageous not to do so. This is due to the fact that, unlearn something is more difficult than learning new thing. The OOP programming paradigm is very different from the traditional Structured Programming.

Preparing to Program

C++, perhaps more than other languages, demands that the programmer design the program before writing it. Trivial problems don't require much design. Complex problems, however, such as the ones professional programmers are challenged with every day, do require design, and the more thorough the design, the more likely it is that the program will solve the problems it is designed to solve, on time and on budget. A good design also makes for a program that is relatively bug-free and easy to maintain. It has been estimated that fully 90 percent of the cost of software is the combined cost of debugging and maintenance. To the extent that good design can reduce those costs, it can have a significant impact on the bottom-line cost of the project.

The first question you need to ask when preparing to design any program is, "*What is the problem I'm trying to solve?*" Every program should have a clear, well-articulated goal, and you'll find that even the simplest programs in this course do so.

The second question every good programmer asks is, "*Can this be accomplished without resorting to writing custom software?*" Reusing an old program, using pen and paper, or buying software off the shelf is often a better solution to a problem than writing something new. The programmer who can offer these alternatives will never suffer from lack of work; finding less-expensive solutions to today's problems will always generate new opportunities later.

Assuming you understand the problem, and it requires writing a new program, you are ready to begin your design.

Your Development Environment

Assume that your computer has a mode in which you can write directly to the screen, without worrying about a graphical environment, such as the ones in Windows or on the Macintosh.

Your compiler may have its own built-in text editor, or you may be using a commercial text editor or word processor that can produce text files. The important thing is that whatever you write your program in, it must save simple, plain-text files, with no word processing commands embedded in the text. Examples of safe editors include Windows Notepad, the DOS Edit command, Brief, Epsilon, EMACS, and vi. Many commercial word processors, such as WordPerfect, Word, and dozens of others, also offer a method for saving simple text files.

The files you create with your editor are called source files, and for C++ they typically are named with the extension `.CPP`, `.CP`, or `.C`. For this course, we'll name all the source code files with the `.CPP` extension, but check your compiler for what it needs.

NOTE: Most C++ compilers don't care what extension you give your source code, but if you don't specify otherwise, many will use `.CPP` by default.

DO use a simple text editor to create your source code, or use the built-in editor that comes with your compiler. DON'T use a word processor that saves special formatting characters. If you do use a word processor, save the file as ASCII text. DO save your files with the `.C`, `.CP`, or `.CPP` extension. DO check your documentation for specifics about your compiler and linker to ensure that you know how to compile and link your programs.

Compiling the Source Code

Although the source code in your file is somewhat cryptic, and anyone who doesn't know C++ will struggle to understand what it is for, it is still in what we call human-readable form. Your source code file is not a program, and it can't be executed, or run, as a program can.

To turn your source code into a program, you use a compiler. How you invoke your compiler, and how you tell it where to find your source code, will vary from compiler to compiler; check your documentation. In GNU C++ you pick the `RUN` menu command or type

```
gcc <filename>
```

from the command line, where <filename> is the name of your source code file (for example, `test.cpp`). Other compilers may do things slightly differently.

NOTE: If you compile the source code from the operating system's command line, you should type the following:

GNU/Dev C++ compiler: `gcc` <filename>
For the Borland C++ compiler: `bcc` <filename>
For the Borland C++ for Windows compiler: `bcc` <filename>
For the Borland Turbo C++ compiler: `tc` <filename>
For the Microsoft compilers: `cl` <filename>

After your source code is compiled, an object file is produced. This file is often named with the extension `.OBJ or .O`. This is still not an executable program, however. To turn this into an executable program, you must run your linker.

Creating an Executable File with the Linker

C++ programs are typically created by linking together one or more OBJ files with one or more libraries. A library is a collection of linkable files that were supplied with your compiler, that you purchased separately, or that you created and compiled. All C++ compilers come with a library of useful functions (or procedures) and classes that you can include in your program. A function is a block of code that performs a service, such as adding two numbers or printing to the screen. A class is a collection of data and related functions; we'll be talking about classes a lot.

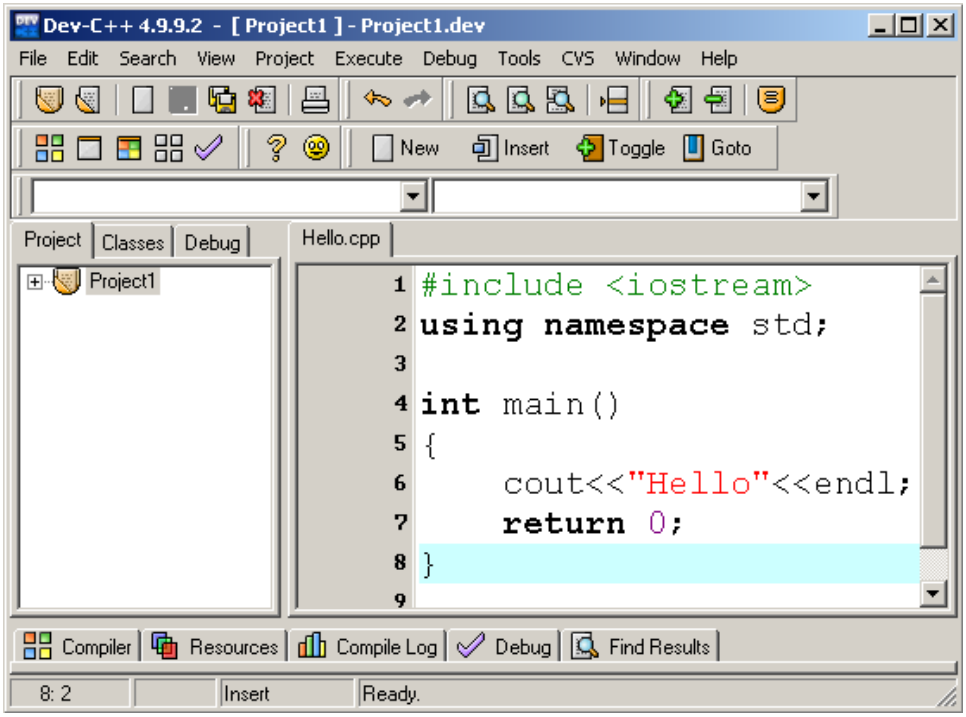
The steps to create an executable file are

1. Create a source code file, with a `.CPP` extension.
2. Compile the source code into a file with the `.OBJ` extension.

Link your OBJ file with any needed libraries to produce an executable program.

Integrated Development Environment (IDE)

Today, many compiler vendors combine all the development tools into a single environment. Some the single interface, the programmer can do editing, compiling, linking, and also debugging. More advanced features might be added to assist the development process, such as "*Watch*" and "*Step*".

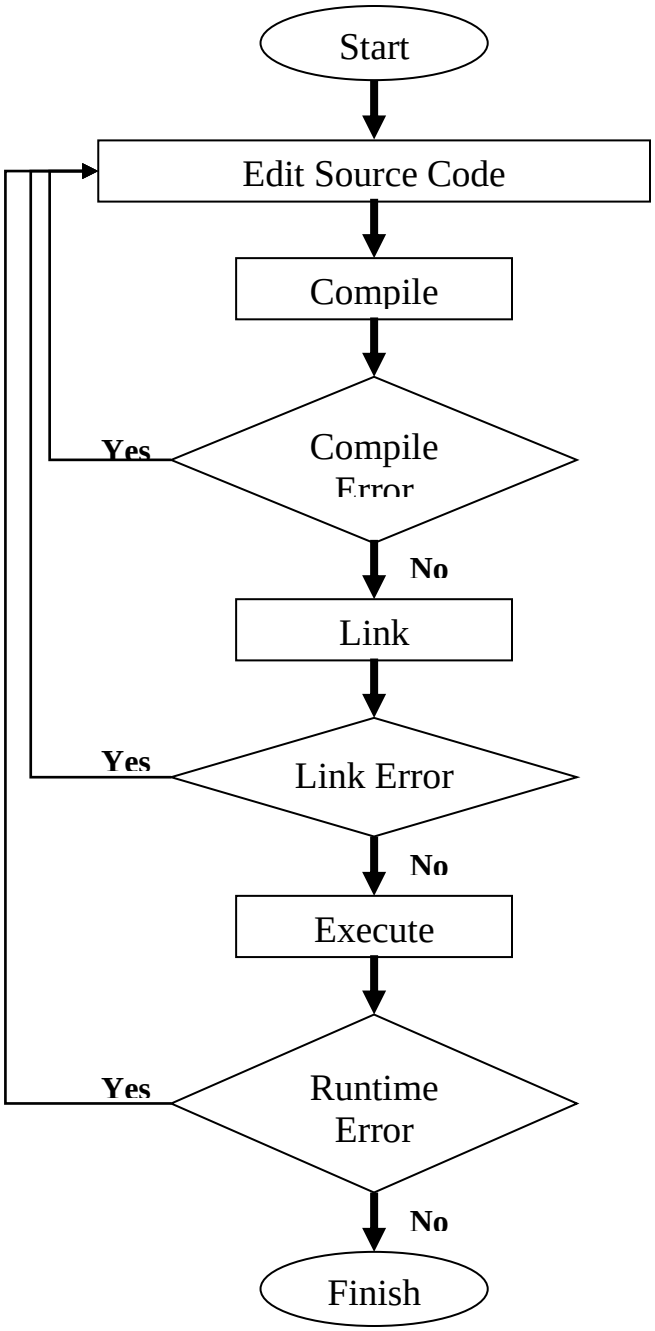


Dev C++ 4.9.9.2 IDE

The Development Cycle

If every program worked the first time you tried it that would be the complete development cycle: Write the program, compile the source code, link the program, and run it. Unfortunately, almost every program, no matter how trivial, can and will have errors, or bugs, in the program. Some bugs will cause the compile to fail, some will cause the link to fail, and some will only show up when you run the program.

Whatever type of bug you find, you must fix it, and that involves editing your source code, recompiling and re-linking, and then rerunning the program. This cycle is represented in diagram below to shows the steps in the development cycle.



HELLO.CPP: *Your First C++ Program*

Type the first program directly into your editor, exactly as shown. Once you are certain it is correct, save the file, compile it, link it, and run it. It will print the words `Hello World` to your screen. Don't worry too much about how it works, this is really just to get you comfortable with the development cycle. Every aspect of this program will be covered over the next couple of parts.

Your first C++ Program

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

> **WARNING:** The following listing contains line numbers on the left. These numbers are for reference only. They should not be typed in to your editor. For example, in line 1 of previous listing, you should enter:
>
> ```
> #include <iostream>
> ```

Make certain you enter this exactly as shown. Pay careful attention to the punctuation. The `<<` in line 5 is the redirection symbol, produced on most keyboards by holding the Shift key and pressing the comma key twice. Line 5 ends with a semicolon; don't leave this off!

Also check to make sure you are following your compiler directions properly. Most compilers will link automatically, but check your documentation. If you get errors, look over your code carefully and determine how it is different from the above. If you see an error on line 1, such as `cannot find file iostream.h`, check your compiler documentation for directions on setting up your `include` path or environment variables. If you receive an error that there is no prototype for `main`, add the line `int main();` just before line 3. You will need to add this line before the beginning of the `main` function in every program in this course. Most compilers don't require this, but a few do.

Your finished program will look like this:

Try running `HELLO.EXE`; it should write

```
Hello World!
```

directly to your screen. If so, congratulations! You've just entered, compiled, and run your first C++ program. It may not look like much, but almost every professional C++ programmer started out with this exact program.

Compile Errors

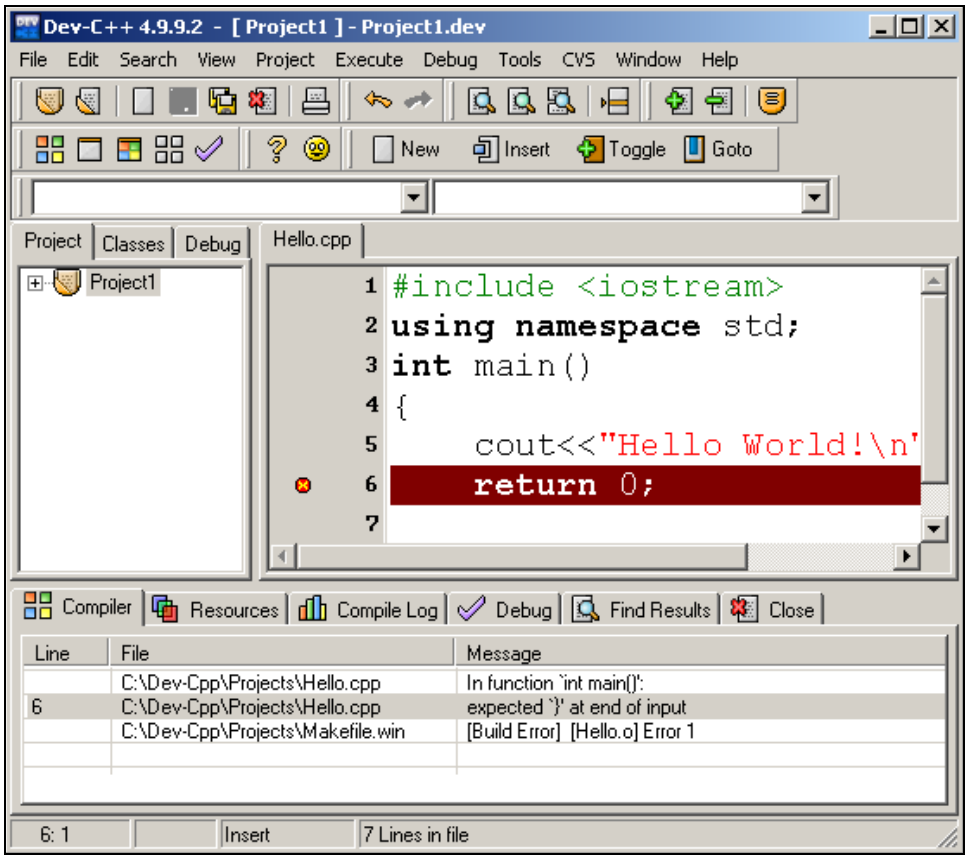
Compile-time errors can occur for any number of reasons. Usually they are a result of a typo or other inadvertent minor error. Good compilers will not only tell you what you did wrong, they'll point you to the exact place in your code where you made the mistake. The great ones will even suggest a remedy!

You can see this by intentionally putting an error into your program. If `HELLO.CPP` ran smoothly, edit it now and remove the closing brace on line 6. Your program will now look like the following:

Hello.cpp

```
1: #include <iostream>
2: using namespace std;
3: int main()
4: {
5:      cout << "Hello World!\n";
6:      return 0;
```

Recompile your program and you should see an error that looks similar to the following:



This error tells you the file and line number of the problem, and what the problem is (although I admit it is somewhat cryptic). Note that the error message points you to line 6. Beware that in other programs, sometime the errors might be caused by the codes before the indicated line.

The Parts of a C++ Program

C++ programs consist of objects, functions, variables, and other component parts. We will consider these parts in depth, but to get a sense of how a program fits together you must see a complete working program. Today you learn

- The parts of a C++ program.
- How the parts work together.
- What a function is and what it does.

A Simple Program

Even the simple program such as HELLO.CPP as discussed, had many interesting parts. This section will review this program in more detail. Following listing reproduces the original version of HELLO.CPP for your convenience.

Hello.cpp demonstrates the parts of a C++ program.

Source Code: *Hello.cpp*

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

Output

```
Hello World!
```

On line 1, the file iostream.h is included in the file. The first character is the # symbol, which is a signal to the preprocessor. Each time you start your compiler, the preprocessor is run. The preprocessor reads through your source code, looking for lines that begin with the pound symbol (#), and acts on those lines before the compiler runs.

include is a preprocessor instruction that says, "*What follows is a filename. Find that file and read it in right here*". The angle brackets around the filename tell the preprocessor to look in all the usual places for this file. If your compiler is set up correctly, the angle brackets will cause the preprocessor to look for the file iostream.h in the directory that holds all the H files for your compiler. The file iostream.h (Input-Output-Stream) is used by cout, which assists with writing to the screen. The effect of line 1 is to include the file iostream.h into this program as if you had typed it in yourself.

New Term: The preprocessor runs before your compiler each time the compiler is invoked. The preprocessor translates any line that begins with a pound symbol (#) into a special command, getting your code file ready for the compiler.

On line 2, indicate the reference to **std** namespace. This is new standard in C++. Namespace is for the purpose of qualifying unique reference to an ambiguous class name and to organize classes in more manageable manner.

Line 3 begins the actual program with a function named main(). Every C++ program has a main() function. In general, a function is a block of code that performs one or more actions. Usually functions are invoked or called by other functions, but main() is special. When your program starts, main() is called automatically.

main(), like all functions, must state what kind of value it will return. The return value type for main() in HELLO.CPP is void, which means that this function will not return any value at all. Returning values from functions is discussed in detail on the topic of "*Expressions and Statements*".

All functions begin with an opening brace ({) and end with a closing brace (}). The braces for the main() function are on lines 4 and 7. Everything between the opening and closing braces is considered a part of the function.

The meat and potatoes of this program is on line 5. The object cout is used to print a message to the screen. We'll cover objects in general on topic "*Basic Classes*" and cout and its related object cin in detail on topic "*The Preprocessor*". These two objects, cout and cin, are used in C++ to print strings and values to the screen. A string is just a set of characters.

Here's how cout is used: type the word cout, followed by the output redirection operator (<<). Whatever follows the output redirection operator is written to the screen. If you want a string of characters written, be sure to enclose them in double quotes ("), as shown on line 5.

New Term: A text string is a series of printable characters.

The final two characters, \n, tell cout to put a new line after the words Hello World! This special code is explained in detail later.

All ANSI-compliant programs declare main() to return an int. This value is "*returned*" to the operating system when your program completes. Some programmers signal an error by returning the value 1. In this course, main() will always return 0.

The main() function ends on line 7 with the closing brace.

A Brief Look at cout

On topic "*Streams*" you will see how to use cout to print data to the screen. For now, you can use cout without fully understanding how it works. To print a value to the screen, write the word cout, followed by the insertion operator (<<), which you create by typing the less-than character (<) twice. Even though this is two characters, C++ treats it as one.

Follow the insertion character with your data. Listing below illustrates how this is used. Type in the example exactly as written, except substitute your own name where you see Jesse Liberty (unless your name is Jesse Liberty, in which case leave it just the way it is; it's perfect-- but I'm still not splitting royalties!).

Using cout.

Source Code: *cout.cpp*

```
 1  // Example: using cout
 2  #include <iostream>
 3  using namespace std;
 4  int main()
 5  {
 6    cout<<"Hello there.\n";
 7    cout<<"Here is 5: " << 5 << "\n";
 8    cout<<"The manipulator endl writes a new line to the screen."<< endl;
 9    cout << "Here is a very big number:\t" << 70000 << endl;
10    cout << "Here is the sum of 8 and 5:\t" << 8+5 << endl;
11    cout << "Here's a fraction:\t\t" << (float) 5/8 << endl;
12    cout << "And a very very big number:\t" << (double) 7000*7000 << endl;
13    cout << "Don't forget to replace Jesse Liberty with your name...\n";
14    cout << "Jesse Liberty is a C++ programmer!\n";
15    return 0;
16  }
```

Output

```
Hello there.
Here is 5: 5
The manipulator endl writes a new line to the screen.
Here is a very big number:      70000
Here is the sum of 8 and 5:     13
Here's a fraction:              0.625
And a very very big number:     4.9e+07
Don't forget to replace Jesse Liberty with your name...
Jesse Liberty is a C++ programmer!
```

On line 2, the statement #include <iostream.h> causes the iostream.h file to be added to your source code. This is required if you use cout and its related functions.

On line 6 is the simplest use of cout, printing a string or series of characters. The symbol \n is a special formatting character. It tells cout to print a newline character to the screen.

Three values are passed to cout on line 7, and each value is separated by the insertion operator. The first value is the string "*Here is 5:* ". Note the space after the colon. The space is part of the string. Next, the value 5 is passed to the insertion operator and the newline character (always in double quotes or single quotes). This causes the line

```
Here is 5: 5
```

to be printed to the screen. Because there is no newline character after the first string, the next value is printed immediately afterwards. This is called concatenating the two values.

On line 8, an informative message is printed, and then the manipulator endl is used. The purpose of endl is to write a new line to the screen.

On line 9, a new formatting character, \t, is introduced. This inserts a tab character and is used on lines 8-12 to line up the output. Line 9 shows that not only integers, but long integers as well can be printed.

Line 10 demonstrates that cout will do simple addition. The value of 8+5 is passed to cout, but 13 is printed.

On line 11, the value 5/8 is inserted into cout. The term (float) tells cout that you want this value evaluated as a decimal equivalent, and so a fraction is printed. On line 12 the value 7000 * 7000 is given to cout, and the term (double) is used to tell cout that you want this to be printed using scientific notation. All of this will be explained on topic "*Variables and Constants*", when data types are discussed.

On line 14, you substituted your name, and the output confirmed that you are indeed a C++ programmer. It must be true, because the computer said so!

Comments

When you are writing a program, it is always clear and self-evident what you are trying to do. Funny thing, though--a month later, when you return to the program, it can be quite confusing and unclear. I'm not sure how that confusion creeps into your program, but it always does.

To fight the onset of confusion, and to help others understand your code, you'll want to use comments. Comments are simply text that is ignored by the compiler, but that may inform the reader of what you are doing at any particular point in your program.

Types of Comments

C++ comments come in two flavors: the double-slash (//) comment, and the slash-star (/*) comment. The double-slash comment, which will be referred to as a C++-style comment, tells the compiler to ignore everything that follows this comment, until the end of the line.

The slash-star comment mark tells the compiler to ignore everything that follows until it finds a star-slash (*/) comment mark. These marks will be referred to as C-style comments. Every /* must be matched with a closing */.

As you might guess, C-style comments are used in the C language as well, but C++-style comments are not part of the official definition of C.

Many C++ programmers use the C++-style comment most of the time, and reserve C-style comments for blocking out large blocks of a program. You can include C++-style comments within a block "commented out" by C-style comments; everything, including the C++-style comments, is ignored between the C-style comment marks.

Using Comments

As a general rule, the overall program should have comments at the beginning, telling you what the program does. Each function should also have comments explaining what the function does and what values it returns. Finally, any statement in your program that is obscure or less than obvious should be commented as well.

Listing below demonstrates the use of comments, showing that they do not affect the processing of the program or its output.

Comments.cpp demonstrates comments.

Source Code: *Comments.cpp*

```
 1  #include <iostream>
 2  using namespace std;
 3  int main()
 4  {
 5    /*    this is a comment
 6          and it extends until the closing
 7          star-slash comment mark */
 8    cout << "Hello World!\n";
 9    // this comment ends at the end of the line
10    cout << "That comment ended!\n";
11
12    // double slash comments can be alone on a line
13    /* as can slash-star comments */
14    return 0;
15  }
```

Output

```
Hello World!
That comment ended!
```

The comments on lines 5 through 7 are completely ignored by the compiler, as are the comments on lines 9, 12, and 13. The comment on line 9 ended with the end of the line, however, while the comments on lines 5 and 13 required a closing comment mark.

Comments at the Top of Each File

It is a good idea to put a comment block at the top of every file you write. The exact style of this block of comments is a matter of individual taste, but every such header should include at least the following information:

- The name of the function or program.
- The name of the file.
- What the function or program does.
- A description of how the program works.
- The author's name.
- A revision history (notes on each change made).
- Additional notes as needed.

For example, the following block of comments might appear at the top of the Hello World program.

```
/************************************************************
Program:      Hello World
File:         Hello.cpp
Function:     Main (complete program listing in this file)
Description:  Prints the words "Hello world" to the screen
Author:       Thong Sam Pah
```

```
Notes:        This is an introductory, sample program.
Revisions:    1.00  10/1/94 (jl) First release
              1.01  10/2/94 (jl) Capitalized "World"

**********************************************************/
```

It is very important that you keep the notes and descriptions up-to-date. A common problem with headers like this is that they are neglected after their initial creation, and over time they become increasingly misleading. When properly maintained, however, they can be invaluable guides to the overall program.

The listings in the rest of this course will leave off the headings in an attempt to save room. That does not diminish their importance, however, so they will appear in the programs provided at the end of each week.

A Final Word of Caution About Comments

Comments that state the obvious are less than useful. In fact, they can be counterproductive, because the code may change and the programmer may neglect to update the comment. What is obvious to one person may be obscure to another, however, so judgment is required.

The bottom line is that comments should not say what is happening, they should say why it is happening.

DO add comments to your code. **DO** keep comments up-to-date. **DO** use comments to tell what a section of code does. **DON'T** use comments for self-explanatory code.

Functions

While main() is a function, it is an unusual one. Typical functions are called, or invoked, during the course of your program. A program is executed line by line in the order it appears in your source code, until a function is reached. Then the program branches off to execute the function. When the function finishes, it returns control to the line of code immediately following the call to the function.

A good analogy for this is sharpening your pencil. If you are drawing a picture, and your pencil breaks, you might stop drawing, go sharpen the pencil, and then return to what you were doing. When a program needs a service performed, it can call a function to perform the service and then pick up where it left off when the function is finished running. Listing below demonstrates this idea.

Demonstrating a call to a function.

Source Code: *FunctionCall.cpp*

```
 1  #include <iostream>
 2  using namespace std;
 3  // function Demonstration Function
 4  // prints out a useful message
 5  void DemonstrationFunction()
 6  {
 7    cout << "In Demonstration Function\n";
 8  }
 9
10  // function main - prints out a message, then
11  // calls DemonstrationFunction, then prints out
12  // a second message.
13  int main()
14  {
15    cout << "In main\n" ;
16    DemonstrationFunction();
17    cout << "Back in main\n";
18    return 0;
19  }
```

Output

```
In main
In Demonstration Function
Back in main
```

The function DemonstrationFunction() is defined on lines 5-7. When it is called, it prints a message to the screen and then returns.

Line 13 is the beginning of the actual program. On line 15, main() prints out a message saying it is in main(). After printing the message, line 16 calls DemonstrationFunction(). This call causes the commands in DemonstrationFunction() to execute. In this case, the entire function consists of the code on line 7, which prints another message. When DemonstrationFunction() completes (line 8), it returns back to where it was called from. In this case the program returns to line 17, where main() prints its final line.

Using Functions

Functions either return a value or they return void, meaning they return nothing. A function that adds two integers might return the sum, and thus would be defined to return an integer value. A function that just prints a message has nothing to return and would be declared to return void.

Functions consist of a header and a body. The header consists, in turn, of the return type, the function name, and the parameters to that function. The parameters to a function allow values to be passed into the function. Thus, if the function were to add two numbers, the numbers would be the parameters to the function. Here's a typical function header:

```
int Sum(int a, int b)
```

A parameter is a declaration of what type of value will be passed in; the actual value passed in by the calling function is called the argument. Many programmers use these two terms, parameters and arguments, as synonyms. Others are careful about the technical distinction. This course will use the terms interchangeably.

The body of a function consists of an opening brace, zero or more statements, and a closing brace. The statements constitute the work of the function. A function may return a value, using a return statement. This statement will also cause the function to exit. If you don't put a return statement into your function, it will automatically return void at the end of the function. The value returned must be of the type declared in the function header.

Listing below demonstrates a function that takes two integer parameters and returns an integer value. Don't worry about the syntax or the specifics of how to work with integer values (for example, int x) for now; that is covered in detail on next part.

Func.cpp demonstrates a simple function.

Source Code: *Func.cpp*

```
 1  #include <iostream>
 2  using namespace std;
 3  int Add (int x, int y)
 4  {
 5    cout << "In Add(), received " << x << " and " << y << "\n";
 6    return (x+y);
 7  }
 8
 9  int main()
10  {
11    cout << "I'm in main()!\n";
12    int a, b, c;
13    cout << "Enter two numbers: ";
14    cin >> a;
15    cin >> b;
16    cout << "\nCalling Add()\n";
17    c=Add(a,b);
18    cout << "\nBack in main().\n";
19    cout << "c was set to " << c;
20    cout << "\nExiting...\n\n";
21    return 0;
22  }
```

Output

```
I'm in main()!
Enter two numbers: 3 5

Calling Add()
In Add(), received 3 and 5

Back in main().
c was set to 8

Exiting...
```

The function Add() is defined on line 3. It takes two integer parameters and returns an integer value. The program itself begins on line 9 and on line 11, where it prints a message. The program prompts the user for two numbers (lines 13 to 15). The user types each number, separated by a space, and then presses the Enter key. main() passes the two numbers typed in by the user as arguments to the Add() function on line 17.

Processing branches to the Add() function, which starts on line 3. The parameters ***a*** and ***b*** are printed and then added together. The result is returned on line 6, and the function returns.

In lines 14 and 15, the cin object is used to obtain a number for the variables ***a*** and ***b***, and cout is used to write the values to the screen. Variables and other aspects of this program are explored in depth in the next few parts.

Variables and Constants

Programs need a way to store the data they use. Variables and constants offer various ways to represent and manipulate that data.

Today you will learn

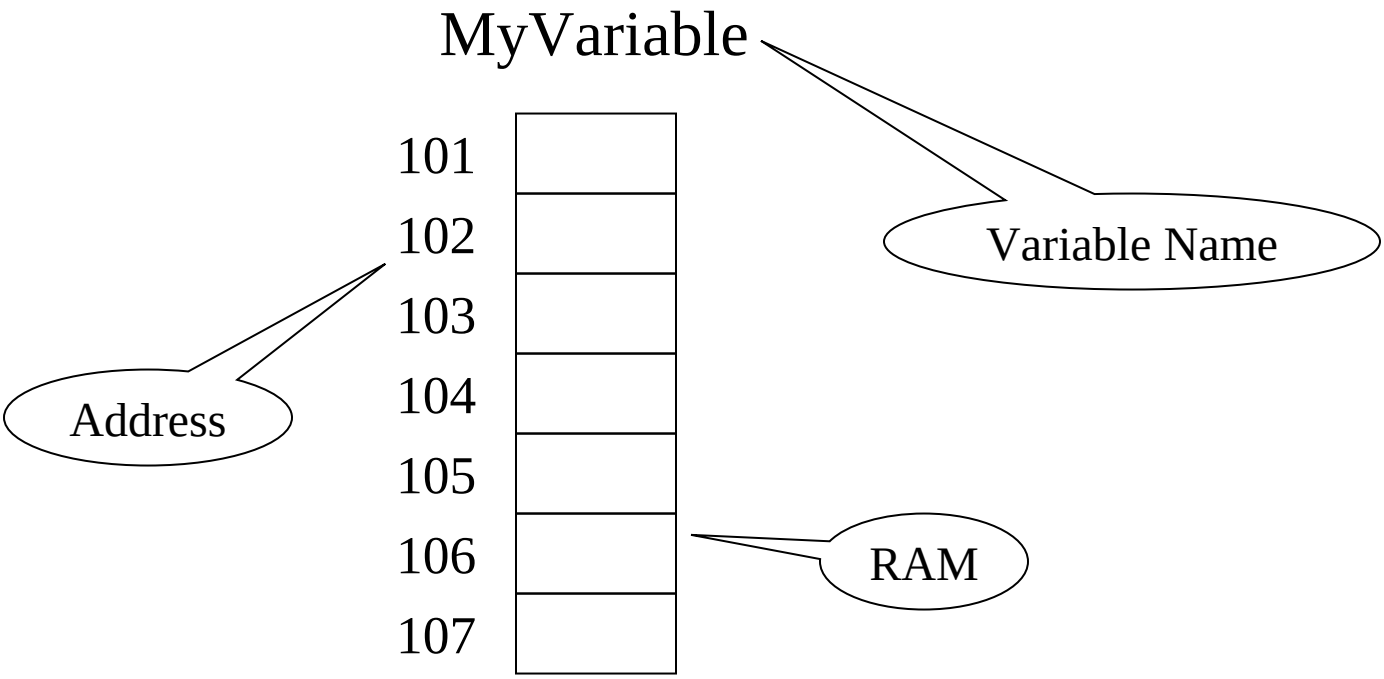
- How to declare and define variables and constants.
- How to assign values to variables and manipulate those values.
- How to write the value of a variable to the screen.

What Is a Variable?

In C++ a variable is a place to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value.

Your computer's memory can be viewed as a series of cubbyholes. Each cubbyhole is one of many, many such holes all lined up. Each cubbyhole--or memory location--is numbered sequentially. These numbers are known as memory addresses. A variable reserves one or more cubbyholes in which you may store a value.

Your variable's name (for example, `myVariable`) is a label on one of these cubbyholes, so that you can find it easily without knowing its actual memory address. Figure 3.1 is a schematic representation of this idea. As you can see from the figure, `myVariable` starts at memory address `103`. Depending on the size of `myVariable`, it can take up one or more memory addresses.



A schematic representation of memory.

Setting Aside Memory

When you define a variable in C++, you must tell the compiler what kind of variable it is: an integer, a character, and so forth. This information tells the compiler how much room to set aside and what kind of value you want to store in your variable.

Each cubbyhole is one byte large. If the type of variable you create is two bytes in size, it needs two bytes of memory, or two cubbyholes. The type of the variable (for example, integer) tells the compiler how much memory (how many cubbyholes) to set aside for the variable.

Because computers use bits and bytes to represent values, and because memory is measured in bytes, it is important that you understand and are comfortable with these concepts.

Size of Integers

On any one computer, each variable type takes up a single, unchanging amount of room. That is, an integer might be two bytes on one machine, and four on another, but on either computer it is always the same, day in and day out.

A `char` variable (used to hold characters) is most often one byte long. A `short` integer is two bytes on most computers, a `long` integer is usually four bytes, and an integer (without the keyword `short` or `long`) can be two or four bytes. Listing below should help you determine the exact size of these types on your computer.

Determining the size of variable types on your computer.

Source Code: *sizeof.cpp*

```
 1  #include <iostream>
 2  using namespace std;
 3  int main()
 4  {
 5    cout << "The size of an int is:\t\t"    << sizeof(int)    << " bytes.\n";
 6    cout << "The size of a short int is:\t" << sizeof(short)  << " bytes.\n";
 7    cout << "The size of a long int is:\t"  << sizeof(long)   << " bytes.\n";
 8    cout << "The size of a char is:\t\t"    << sizeof(char)   << " bytes.\n";
 9    cout << "The size of a float is:\t\t"   << sizeof(float)  << " bytes.\n";
10    cout << "The size of a double is:\t"    << sizeof(double) << " bytes.\n";
11
12    return 0;
13  }
```

Output

```
The size of an int is:          2 bytes.
The size of a short int is:     2 bytes.
The size of a long int is:      8 bytes.
The size of a char is:          1 bytes.
The size of a float is:         4 bytes.
The size of a double is:        8 bytes.
```

NOTE: On your computer, the number of bytes presented might be different.

Analysis: Most of Listing above should be pretty familiar. The one new feature is the use of the `sizeof()` function in lines 5 through 10. `sizeof()` is provided by your compiler, and it tells you the size of the object you pass in as a parameter. For example, on line 5 the keyword `int` is passed into `sizeof()`. Using `sizeof()`, I was able to determine that on my computer an `int` is equal to a `short int`, which is 2 bytes.

signed and unsigned

In addition, all integer types come in two varieties: `signed` and `unsigned`. The idea here is that sometimes you need negative numbers, and sometimes you don't. Integers (`short` and `long`) without the word "unsigned" are assumed to be `signed`. `Signed` integers are either negative or positive. `Unsigned` integers are always positive.

Because you have the same number of bytes for both `signed` and `unsigned` integers, the largest number you can store in an `unsigned` integer is twice as big as the largest positive number you can store in a `signed` integer. An `unsigned short` integer can handle numbers from 0 to 65,535. Half the numbers represented by a `signed short` are negative, thus a `signed short` can only represent numbers from -32,768 to 32,767. If this is confusing, be sure to read Appendix A, "Operator Precedence."

Fundamental Variable Types

Several other variable types are built into C++. They can be conveniently divided into integer variables (the type discussed so far), floating-point variables, and character variables.

Floating-point variables have values that can be expressed as fractions--that is, they are real numbers. Character variables hold a single byte and are used for holding the 256 characters and symbols of the ASCII and extended ASCII character sets.

New Term: *The ASCII character set* is the set of characters standardized for use on computers. ASCII is an acronym for American Standard Code for Information Interchange. Nearly every computer operating system supports ASCII, though many support other international character sets as well.

The types of variables used in C++ programs are described in Table below. This table shows the variable type, how much room this course assumes it takes in memory, and what kinds of values can be stored in these variables. The values that can be stored are determined by the size of the variable types, so check your output from listing sizeof.cpp.

Table: *Variable Types*

Type	*Size*	*Values*
`unsigned short int`	2 bytes	0 to 65,535
`short int`	2 bytes	-32,768 to 32,767
`unsigned long int`	4 bytes	0 to 4,294,967,295
`long int`	4 bytes	-2,147,483,648 to 2,147,483,647
`int (16 bit)`	2 bytes	-32,768 to 32,767
`int (32 bit)`	4 bytes	-2,147,483,648 to 2,147,483,647
`unsigned int (16 bit)`	2 bytes	0 to 65,535
`unsigned int (32 bit)`	2 bytes	0 to 4,294,967,295
`char`	1 byte	256 character values
`float`	4 bytes	1.2e-38 to 3.4e38
`double`	8 bytes	2.2e-308 to 1.8e308

NOTE: The sizes of variables might be different from those shown in table above, depending on the compiler and the computer you are using. If your computer had the same output as was presented in listing sizeof.cpp, Table above should apply to your compiler. If your output from listing sizeof.cpp was different, you should consult your compiler's manual for the values that your variable types can hold.

Defining a Variable

You create or define a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon. The variable name can be virtually any combination of letters, but cannot contain spaces. Legal variable names include `x`, `J23qrsnf`, and `myAge`. Good variable names tell you what the variables are for; using good names makes it easier to understand the flow of your program. The following statement defines an integer variable called `myAge`:

```
int myAge;
```

As a general programming practice, avoid such horrific names as `J23qrsnf`, and restrict single-letter variable names (such as `x` or `i`) to variables that are used only very briefly. Try to use expressive names such as `myAge` or `howMany`. Such names are easier to understand three weeks later when you are scratching your head trying to figure out what you meant when you wrote that line of code.

Try this experiment: Guess what these pieces of programs do, based on the first few lines of code:

Example 1

```
main()
{
     unsigned short x;
     unsigned short y;
     ULONG z;
     z = x * y;
}
```

Example 2

```
main ()
{
     unsigned short Width;
     unsigned short Length;
     unsigned short Area;
     Area = Width * Length;
}
```

Clearly, the second program is easier to understand, and the inconvenience of having to type the longer variable names is more than made up for by how much easier it is to maintain the second program.

Case Sensitivity

C++ is case-sensitive. In other words, uppercase and lowercase letters are considered to be different. A variable named `age` is different from `Age`, which is different from `AGE`.

NOTE: Some compilers allow you to turn case sensitivity off. Don't be tempted to do this; your programs won't work with other compilers, and other C++ programmers will be very confused by your code.

There are various conventions for how to name variables, and although it doesn't much matter which method you adopt, it is important to be consistent throughout your program.

Many programmers prefer to use all lowercase letters for their variable names. If the name requires two words (for example, my car), there are two popular conventions: `my_car` or `myCar`. The latter form is called camel-notation, because the capitalization looks something like a camel's hump.

Some people find the underscore character (`my_car`) to be easier to read, while others prefer to avoid the underscore, because it is more difficult to type. This course uses camel-notation, in which the second and all subsequent words are capitalized: `myCar`, `theQuickBrownFox`, and so forth.

NOTE: Many advanced programmers employ a notation style that is often referred to as Hungarian notation. The idea behind Hungarian notation is to prefix every variable with a set of characters that describes its type. Integer variables might begin with a lowercase letter i, longs might begin with a lowercase l. Other notations indicate constants, globals, pointers, and so forth. Most of this is much more important in C programming, because C++ supports the creation of user-defined types and because C++ is strongly typed.

Keywords

Some words are reserved by C++, and you may not use them as variable names. These are keywords used by the compiler to control your program. Keywords include `if`, `while`, `for`, and `main`. Your compiler manual should provide a complete list, but generally, any reasonable name for a variable is almost certainly not a keyword.

DO define a variable by writing the type, then the variable name. **DO** use meaningful variable names. DO remember that C++ is case sensitive. **DON'T** use C++ keywords as variable names. DO understand the number of bytes each variable type consumes in memory, and what values can be stored in variables of that type. **DON'T** use `unsigned` variables for negative numbers.

Creating More Than One Variable at a Time

You can create more than one variable of the same type in one statement by writing the type and then the variable names, separated by commas. For example:

```
unsigned int myAge, myWeight;   // two unsigned int variables
long area, width, length;       // three longs
```

As you can see, `myAge` and `myWeight` are each declared as `unsigned` integer variables. The second line declares three individual `long` variables named `area`, `width`, and `length`. The type (`long`) is assigned to all the variables, so you cannot mix types in one definition statement.

Assigning Values to Your Variables

You assign a value to a variable by using the assignment operator (`=`). Thus, you would assign `5` to `Width` by writing

```
unsigned short Width;
Width = 5;
```

You can combine these steps and initialize `Width` when you define it by writing

```
unsigned short Width = 5;
```

Initialization looks very much like assignment, and with integer variables, the difference is minor. Later, when constants are covered, you will see that some values must be initialized because they cannot be assigned to. The essential difference is that initialization takes place at the moment you create the variable.

Just as you can define more than one variable at a time, you can initialize more than one variable at creation. For example:

```
// create two long variables and initialize them
Âlong width = 5, length = 7;
```

This example initializes the `long` integer variable `width` to the value `5` and the `long` integer variable `length` to the value `7`. You can even mix definitions and initializations:

```
int myAge = 39, yourAge, hisAge = 40;
```

This example creates three type `int` variables, and it initializes the first and third.

Listing below shows a complete program, ready to compile, that computes the area of a rectangle and writes the answer to the screen.

A demonstration of the use of variables.

Source Code:*Variables.cpp*

```
 1  // Demonstration of variables
 2  #include <iostream>
 3  using namespace std;
 4  int main()
 5  {
 6    unsigned short int Width = 5, Length;
 7    Length = 10;
 8
 9    // create  an unsigned short and initialize with result
10    // of multiplying Width by Length
11    unsigned short int Area  = Width * Length;
12
13    cout << "Width:" << Width << "\n";
14    cout << "Length:"  << Length << endl;
15    cout << "Area:" << Area << endl;
16    return 0;
17  }
```

Output

```
Width:5
Length:10
Area:50
```

Analysis

Line	*Description*
2	Includes the required `include` statement for the `iostream`'s library so that `cout` will work.
4	Begins the program.
6	`Width` is defined as an `unsigned short` integer, and its value is initialized to `5`. Another `unsigned short` integer, `Length`, is also defined, but it is not initialized. On line 7, the value `10` is assigned to `Length`
11	An `unsigned short` integer, `Area`, is defined, and it is initialized with the value obtained by multiplying `Width` times `Length`.
13-15	The variables' value are printed to the screen. Note that the special word `endl` creates a new line

typedef

It can become tedious, repetitious, and, most important, error-prone to keep writing `unsigned short int`. C++ enables you to create an alias for this phrase by using the keyword `typedef`, which stands for type definition.

In effect, you are creating a synonym, and it is important to distinguish this from creating a new type (which you will do later). `typedef` is used by writing the keyword `typedef`, followed by the existing type and then the new name. For example

```
typedef unsigned short int USHORT
```

creates the new name `USHORT` that you can use anywhere you might have written `unsigned short int`. Listing below is a replay of Variables.cpp listing, using the type definition `USHORT` rather than `unsigned short int`.

A demonstration of typedef.

Source Code:*Typeof.cpp*

```
 1  // Demonstrates typedef keyword
 2  #include <iostream>
 3  using namespace std;
 4
 5  typedef unsigned short int USHORT;       //typedef defined
 6
 7  void main()
 8  {
 9    USHORT  Width = 5;
10    USHORT Length;
11    Length = 10;
12    USHORT Area  = Width * Length;
13    cout << "Width:" << Width << "\n";
14    cout << "Length:"  << Length << endl;
15    cout << "Area:" << Area <<endl;
16  }
```

```
Output
Width:5
Length:10
Area:50
```

Analysis: On line 5, `USHORT` is type defined as a synonym for `unsigned short int`. The program is very much like Listing Variables.cpp, and the output is the same.

When to Use short and When to Use long

One source of confusion for new novice is when to declare a variable to be type `long` and when to declare it to be type `short`. The rule, when understood, is fairly straightforward: If there is any chance that the value you'll want to put into your variable will be too big for its type, use a larger type.

As seen in Table Variable Types, `unsigned short` integers, assuming that they are two bytes, can hold a value only up to 65,535. `Signed short` integers can hold only half that. Although `unsigned long` integers can hold an extremely large number (4,294,967,295) that is still quite finite. If you need a larger number, you'll have to go to `float` or `double`, and then you lose some precision. Floats and doubles can hold extremely large numbers, but only the first 7 or 19 digits are significant on most computers. That means that the number is rounded off after that many digits.

Wrapping Around an unsigned Integer

The fact that `unsigned long` integers have a limit to the values they can hold is only rarely a problem, but what happens if you do run out of room?

When an `unsigned` integer reaches its maximum value, it wraps around and starts over. Listing below shows what happens if you try to put too large a value into a `short` integer.

A demonstration of putting too large a value in an unsigned integer.

Source Code: *Overflow.cpp*

```
 1 #include <iostream>
 2 using namespace std;
 3 int main()
 4 {
 5   unsigned short int smallNumber;
 6   smallNumber = 65535;
 7   cout << "small number:" << smallNumber << endl;
 8   smallNumber++;
 9   cout << "small number:" << smallNumber << endl;
10   smallNumber++;
11   cout << "small number:" << smallNumber << endl;
12   return 0;
13 }
```

Output

```
small number:65535
small number:0
small number:1
```

Analysis

Line	*Description*
5	`The variable smallNumber` is declared to be an `unsigned short int`, which on my computer is a two-byte variable, able to hold a value between 0 and 65,535
6	The maximum value is assigned to `smallNumber`
7	`The value of smallNumber is printed`
8	`smallNumber` is incremented; that is, 1 is added to it. The symbol for incrementing is `++`. Thus, the value in `smallNumber` would be `65,536`. However, `unsigned short` integers can't hold a number larger than 65,535, so the value is wrapped around to `0`
9	`The value of smallNumber is printed again`
10	`The variable smallNumber` is incremented again, and then its new value, `1`, is printed.

Wrapping Around a signed Integer

A `signed` integer is different from an `unsigned` integer, in that half of the values you can represent are negative. Instead of picturing a traditional car odometer, you might picture one that rotates up for positive numbers and down for negative numbers. One mile from 0 is either 1 or -1. When run out of positive numbers, run right into the largest negative numbers and then count back down to 0. Listing below shows what happens when add 1 to the maximum positive number in an `unsigned short` integer.

A demonstration of adding too large a number to a signed integer.

Source Code:*Overflow2.cpp*

```
 1  #include <iostream>
 2  using namespace std;
 3  int main()
 4  {
 5    short int smallNumber;
 6    smallNumber = 32767;
 7    cout << "small number:" << smallNumber << endl;
 8    smallNumber++;
 9    cout << "small number:" << smallNumber << endl;
10    smallNumber++;
11    cout << "small number:" << smallNumber << endl;
12    return 0;
13  }
```

Output

```
small number:32767
small number:-32768
small number:-32767
```

Analysis: On line 5, `smallNumber` is declared this time to be a `signed short` integer (if you don't explicitly say that it is `unsigned`, it is assumed to be `signed`). The program proceeds much as the preceding one, but the output is quite different. To fully understand this output, you must be comfortable with how `signed` numbers are represented as bits in a two-byte integer. The bottom line, however, is that just like an `unsigned` integer, the `signed` integer wraps around from its highest positive value to its highest negative value.

Characters

Character variables (`type char`) are typically 1 byte, enough to hold 256 values (see Appendix C). A `char` can be interpreted as a small number (0-255) or as a member of the ASCII set. ASCII stands for the American Standard Code for Information Interchange. The ASCII character set and its ISO (International Standards Organization) equivalent are a way to encode all the letters, numerals, and punctuation marks. Computers do not know about letters, punctuation, or sentences. All they understand are numbers. In fact, all they really know about is whether or not a sufficient amount of electricity is at a particular junction of wires. If so, it is represented internally as a `1`; if not, it is represented as a `0`. By grouping ones and zeros, the computer is able to generate patterns that can be interpreted as numbers, and these in turn can be assigned to letters and punctuation.

In the ASCII code, the lowercase letter "a" is assigned the value `97`. All the lower- and uppercase letters, all the numerals, and all the punctuation marks are assigned values between 1 and 128. Another 128 marks and symbols are reserved for use by the computer maker, although the IBM extended character set has become something of a standard.

Characters and Numbers

When you put a character, for example, `` `a' ``, into a `char` variable, what is really there is just a number between 0 and 255. The compiler knows, however, how to translate back and forth between characters (represented by a single quotation mark and then a letter, numeral, or punctuation mark, followed by a closing single quotation mark) and one of the ASCII values.

The value/letter relationship is arbitrary; there is no particular reason that the lowercase "a" is assigned the value `97`. As long as everyone (your keyboard, compiler, and screen) agrees, there is no problem. It is important to realize, however, that there is a big difference between the value `5` and the character `` `5' ``. The latter is actually valued at `53`, much as the letter `` `a' `` is valued at `97`.

Printing characters based on numbers

Source Code:*PrintChar.cpp*

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    for (int i = 32; i<128; i++)
6          cout << (char) i;
7    return 0;
8  }
```

```
Output
!"#$%G'()*+,./0123456789:;<>?@ABCDEFGHIJKLMNOP_QRSTUVWXYZ[\]^'
abcdefghijklmnopqrstuvwxyz<|>~s
```

This simple program prints the character values for the integers 32 through 127.

Special Printing Characters

The C++ compiler recognizes some special characters for formatting. Table below shows the most common ones. You put these into your code by typing the backslash (called the escape character), followed by the character. Thus, to put a tab character into your code, you would enter a single quotation mark, the slash, the letter t, and then a closing single quotation mark:

```
char tabCharacter = `\t';
```

This example declares a `char` variable (`tabCharacter`) and initializes it with the character value `\t`, which is recognized as a tab. The special printing characters are used when printing either to the screen or to a file or other output device.

> **New Term:** An *escape character* changes the meaning of the character that follows it. For example, normally the character `n` means the letter n, but when it is preceded by the escape character (`\`) it means new line.

Table: ***The Escape Characters***

Character	*What it means*
`\n`	New line
`\t`	Tab
`\b`	Backspace
`\"`	Double quote
`\'`	Single quote
`\\`	Backslash

Constants

Like variables, constants are data storage locations. Unlike variables, and as the name implies, constants don't change. You must initialize a constant when you create it, and you cannot assign a new value later.

Literal Constants

C++ has two types of constants: literal and symbolic.

A literal constant is a value typed directly into your program wherever it is needed. For example

```
int myAge = 39;
```

`myAge` is a variable of type `int`; `39` is a literal constant. You can't assign a value to `39`, and its value can't be changed.

Symbolic Constants

A symbolic constant is a constant that is represented by a name, just as a variable is represented. Unlike a variable, however, after a constant is initialized, its value can't be changed.

If your program has one integer variable named `students` and another named `classes`, you could compute how many students you have, given a known number of classes, if you knew there were 15 students per class:

```
students = classes * 15;
```

NOTE: `*` indicates multiplication.

In this example, `15` is a literal constant. Your code would be easier to read, and easier to maintain, if you substituted a symbolic constant for this value:

```
students = classes * studentsPerClass
```

If you later decided to change the number of students in each class, you could do so where you define the constant `studentsPerClass` without having to make a change every place you used that value.

There are two ways to declare a symbolic constant in C++. The old, traditional, and now obsolete way is with a preprocessor directive, `#define`. Defining Constants with #define to define a constant the traditional way, you would enter this:

```
#define studentsPerClass 15
```

Note that `studentsPerClass` is of no particular type (`int`, `char`, and so on). `#define` does a simple text substitution. Every time the preprocessor sees the word `studentsPerClass`, it puts in the text `15`.

Because the preprocessor runs before the compiler, your compiler never sees your constant; it sees the number `15`. Defining Constants with const Although `#define` works, there is a new, much better way to define constants in C++:

```
const unsigned short int studentsPerClass = 15;
```

This example also declares a symbolic constant named `studentsPerClass`, but this time `studentsPerClass` is typed as an `unsigned short int`. This method has several advantages in making your code easier to maintain and in preventing bugs. The biggest difference is that this constant has a type, and the compiler can enforce that it is used according to its type.

NOTE: Constants cannot be changed while the program is running. If you need to change `studentsPerClass`, for example, you need to change the code and recompile.

DON'T use the term `int`. Use `short` and `long` to make it clear which size number you intended. **DO** watch for numbers overrunning the size of the integer and wrapping around incorrect values. **DO** give your variables meaningful names that reflect their use. **DON'T** use keywords as variable names.

Enumerated Constants

Enumerated constants enable you to create new types and then to define variables of those types whose values are restricted to a set of possible values. For example, you can declare `COLOR` to be an enumeration, and you can define that there are five values for `COLOR`: `RED`, `BLUE`, `GREEN`, `WHITE`, and `BLACK`.

The syntax for enumerated constants is to write the keyword `enum`, followed by the type name, an open brace, each of the legal values separated by a comma, and finally a closing brace and a semicolon. Here's an example:

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

This statement performs two tasks:

1. It makes `COLOR` the name of an enumeration, that is, a new type.
2. It makes `RED` a symbolic constant with the value `0`, `BLUE` a symbolic constant with the value `1`, `GREEN` a symbolic constant with the value `2`, and so forth.

Every enumerated constant has an integer value. If you don't specify otherwise, the first constant will have the value `0`, and the rest will count up from there. Any one of the constants can be initialized with a particular value, however, and those that are not initialized will count upward from the ones before them. Thus, if you write

```
enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };
```

then `RED` will have the value `100`; `BLUE`, the value `101`; `GREEN`, the value `500`; `WHITE`, the value `501`; and `BLACK`, the value `700`.

You can define variables of type `COLOR`, but they can be assigned only one of the enumerated values (in this case, `RED`, `BLUE`, `GREEN`, `WHITE`, or `BLACK`, or else `100`, `101`, `500`, `501`, or `700`). You can assign any color value to your `COLOR` variable. In fact, you can assign any integer value, even if it is not a legal color, although a good compiler will issue a warning if you do. It is important to realize that enumerator variables actually are of type `unsigned int`, and that the enumerated constants equate to integer variables. It is, however, very convenient to be able to name these values when working with colors, days of the week, or similar sets of values. Listing Enum.cpp presents a program that uses an enumerated type.

A demonstration of enumerated constants.

Source Code:*Enum.cpp*

```
 1  #include <iostream>
 2  using namespace std;
 3  int main()
 4  {
 5    enum Days{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
 6
 7    Days DayOff;
 8    int x;
 9
10    cout << "What day would you like off (0-6)? ";
11    cin  >> x;
12    DayOff = Days(x);
13
14    if (DayOff == Sunday || DayOff == Saturday)
15          cout << "\nYou're already off on weekends!\n";
16    else
17          cout << "\nOkay, I'll put in the vacation day.\n";
18    return 0;
19  }
```

Output

```
What day would you like off (0-6)?  1

Okay, I'll put in the vacation day.

What day would you like off (0-6)?  0

You're already off on weekends!
```

Analysis: On line 5, the enumerated constant `DAYS` is defined, with seven values counting upward from 0. The user is prompted for a day on line 10. The chosen value, a number between 0 and 6, is compared on line 14 to the enumerated values for Sunday and Saturday, and action is taken accordingly.

The `if` statement will be covered in more detail on topic "*Expressions and Statements*".

You cannot type the word "Sunday" when prompted for a day; the program does not know how to translate the characters in `Sunday` into one of the enumerated values.

NOTE: For this and all the small programs in this course, I've left out all the code you would normally write to deal with what happens when the user types inappropriate data. For example, this program doesn't check, as it would in a real program, to make sure that the user types a number between 0 and 6. This detail has been left out to keep these programs small and simple, and to focus on the issue at hand.

Expression and Statements

At its heart, a program is a set of commands executed in sequence. The power in a program comes from its capability to execute one or another set of commands, based on whether a particular condition is true or false. Today you will learn

- What statements are.
- What blocks are.
- What expressions are.
- How to branch your code based on conditions.
- What truth is, and how to act on it.

Statements

In C++ a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement). All C++ statements end with a semicolon, even the null statement, which is just the semicolon and nothing else. One of the most common statements is the following assignment statement:

```
x = a + b;
```

Unlike in algebra, this statement does not mean that `x` equals `a+b`. This is read, "Assign the value of the sum of `a` and `b` to `x`," or "Assign to `x`, `a+b`." Even though this statement is doing two things, it is one statement and thus has one semicolon. The assignment operator assigns whatever is on the right side of the equal sign to whatever is on the left side.

New Term: A *null statement* is a statement that does nothing.

Whitespace

Whitespace (tabs, spaces, and newlines) is generally ignored in statements. *Whitespace characters* (spaces, tabs, and newlines) cannot be seen. If these characters are printed, you see only the white of the paper The assignment statement previously discussed could be written as

```
x=a+b;
```

or as

```
x                       =a

+           b           ;
```

Although this last variation is perfectly legal, it is also perfectly foolish. Whitespace can be used to make your programs more readable and easier to maintain, or it can be used to create horrific and indecipherable code. In this, as in all things, C++ provides the power; you supply the judgment.

Blocks and Compound Statements

Any place you can put a single statement, you can put a compound statement, also called a block. A block begins with an opening brace ({) and ends with a closing brace (}). Although every statement in the block must end with a semicolon, the block itself does not end with a semicolon. For example

```
{
      temp = a;
      a = b;
      b = temp;
}
```

This block of code acts as one statement and swaps the values in the variables `a` and `b`.

DO use a closing brace any time you have an opening brace. **DO** end your statements with a semicolon. **DO** use whitespace judiciously to make your code clearer.

Expressions

Anything that evaluates to a value is an expression in C++. An expression is said to return a value. Thus, `3+2;` returns the value `5` and so is an expression. All expressions are statements.

The myriad pieces of code that qualify as expressions might surprise you. Here are three examples:

```
3.2                         // returns the value 3.2
PI                          // float const that returns the value 3.14
SecondsPerMinute            // int const that returns 60
```

Assuming that `PI` is a constant equal to `3.14` and `SecondsPerMinute` is a constant equal to `60`, all three of these statements are expressions.

The complicated expression

```
x = a + b;
```

not only adds `a` and `b` and assigns the result to `x`, but returns the value of that assignment (the value of `x`) as well. Thus, this statement is also an expression. Because it is an expression, it can be on the right side of an assignment operator:

```
y = x = a + b;
```

This line is evaluated in the following order: Add `a` to `b`.

Assign the result of the expression `a + b` to `x`.

Assign the result of the assignment expression `x = a + b` to `y`.

If a, b, x, and y are all integers, and if a has the value 2 and b has the value 5, both x and y will be assigned the value 7.

Evaluating complex expressions.

Source Code:*ComplexExpression.cpp*

```
 1  #include <iostream>
 2  using namespace std;
 3  int main()
 4  {
 5    int a=0, b=0, x=0, y=35;
 6    cout << "a: " << a << " b: " << b;
 7    cout << " x: " << x << " y: " << y << endl;
 8    a = 9;
 9    b = 7;
10    y = x = a+b;
11    cout << "a: " << a << " b: " << b;
12    cout << " x: " << x << " y: " << y << endl;
13    return 0;
14  }
```

Output

```
a: 0 b: 0 x: 0 y: 35
a: 9 b: 7 x: 16 y: 16
```

Analysis: On line 5, the four variables are declared and initialized. Their values are printed on lines 6 and 7. On line 8, a is assigned the value 9. One line 9, b is assigned the value 7. On line 10, the values of a and b are summed and the result is assigned to x. This expression (x = a+b) evaluates to a value (the sum of a + b), and that value is in turn assigned to y.

Operators

An operator is a symbol that causes the compiler to take an action. Operators act on operands, and in C++ all operands are expressions. In C++ there are several different categories of operators. Two of these categories are

- Assignment operators.
- Mathematical operators.

Assignment Operator

The assignment operator (=) causes the operand on the left side of the assignment operator to have its value changed to the value on the right side of the assignment operator. The expression

```
x = a + b;
```

assigns the value that is the result of adding a and b to the operand x.

An operand that legally can be on the left side of an assignment operator is called an lvalue. That which can be on the right side is called (you guessed it) an rvalue.

Constants are r-values. They cannot be l-values. Thus, you can write

```
x = 35;          // ok
```

but you can't legally write

```
35 = x;          // error, not an lvalue!
```

New Term: An *lvalue* is an operand that can be on the left side of an expression. An rvalue is an operand that can be on the right side of an expression. Note that all l-values are r-values, but not all r-values are l-values. An example of an rvalue that is not an lvalue is a literal. Thus, you can write `x = 5;`, but you cannot write `5 = x;`.

Mathematical Operators

There are five mathematical operators: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulus (`%`).

Addition and subtraction work as you would expect, although subtraction with `unsigned` integers can lead to surprising results, if the result is a negative number. You saw something much like this before, when variable overflow was described. Listing below shows what happens when you subtract a large `unsigned` number from a small `unsigned` number.

A demonstration of subtraction and integer overflow.

Source Code: *Substraction.cpp*

```
 1  // Listing demonstrates subtraction and
 2  // integer overflow
 3  #include <iostream>
 4  using namespace std;
 5  int main()
 6  {
 7    unsigned int difference;
 8    unsigned int bigNumber = 100;
 9    unsigned int smallNumber = 50;
10    difference = bigNumber - smallNumber;
11    cout << "Difference is: " << difference;
12    difference = smallNumber - bigNumber;
13    cout << "\nNow difference is: " << difference <<endl;
14    return 0;
15  }
```

Output

```
Difference is: 50
Now difference is: 4294967246
```

Analysis: The subtraction operator is invoked on line 10, and the result is printed on line 11, much as we might expect. The subtraction operator is called again on line 12, but this time a large `unsigned` number is subtracted from a small `unsigned` number. The result would be negative, but because it is evaluated (and printed) as an `unsigned` number, the result is an overflow, as described before. This topic is reviewed in detail in "*Operator Precedence*".

Integer Division and Modulus

Integer division is somewhat different from everyday division. When you divide 21 by 4, the result is a real number (a number with a fraction). Integers don't have fractions, and so the "remainder" is lopped off. The answer is therefore 5. To get the remainder, you take 21 modulus 4 (21 % 4) and the result is 1. The modulus operator tells you the remainder after an integer division.

Finding the modulus can be very useful. For example, you might want to print a statement on every 10th action. Any number whose value is 0 when you modulus 10 with that number is an exact multiple of 10. Thus 1 % 10 is 1, 2 % 10 is 2, and so forth, until 10 % 10, whose result is 0. 11 % 10 is back to 1, and this pattern continues until the next multiple of 10, which is 20. We'll use this technique when looping is discussed on topic "*More Program Flow*".

WARNING: Many novice C++ programmers inadvertently put a semicolon after their `if` statements:

```
if(SomeValue < 10);
SomeValue = 10;
```

What was intended here was to test whether `SomeValue` is less than 10, and if so, to set it to `10`, making `10` the minimum value for `SomeValue`. Running this code snippet will show that `SomeValue` is always set to `10`! Why? The `if` statement terminates with the semicolon (the do-nothing operator). Remember that indentation has no meaning to the compiler. This snippet could more accurately have been written as:

```
if (SomeValue < 10)  // test
;  // do nothing
SomeValue = 10;  // assign
```

Removing the semicolon will make the final line part of the `if` statement and will make this code do what was intended.

Combining the Assignment and Mathematical Operators

It is not uncommon to want to add a value to a variable, and then to assign the result back into the variable. If you have a variable `myAge` and you want to increase the value by two, you can write

```
int myAge = 5;
int temp;
temp = myAge + 2;  // add 5 + 2 and put it in temp
myAge = temp;              // put it back in myAge
```

This method, however, is terribly convoluted and wasteful. In C++, you can put the same variable on both sides of the assignment operator, and thus the preceding becomes

```
myAge = myAge + 2;
```

which is much better. In algebra this expression would be meaningless, but in C++ it is read as "*add two to the value in `myAge` and assign the result to `myAge`.*"

Even simpler to write, but perhaps a bit harder to read is

```
myAge += 2;
```

The self-assigned addition operator (`+=`) adds the rvalue to the lvalue and then reassigns the result into the lvalue. This operator is pronounced "plus-equals." The statement would be read "`myAge` plus-equals two." If `myAge` had the value `4` to start, it would have `6` after this statement.

There are self-assigned subtraction (`-=`), division (`/=`), multiplication (`*=`), and modulus (`%=`) operators as well.

Increment and Decrement

The most common value to add (or subtract) and then reassign into a variable is `1`. In C++, increasing a value by 1 is called incrementing, and decreasing by 1 is called decrementing. There are special operators to perform these actions.

The increment operator (`++`) increases the value of the variable by 1, and the decrement operator (`--`) decreases it by 1. Thus, if you have a variable, `C`, and you want to increment it, you would use this statement:

```
C++;                // Start with C and increment it.
```

This statement is equivalent to the more verbose statement

```
C = C + 1;
```

which you learned is also equivalent to the moderately verbose statement

```
C += 1;
```

Prefix and Postfix

Both the increment operator (`++`) and the decrement operator(`--`) come in two varieties: prefix and postfix. The prefix variety is written before the variable name (`++myAge`); the postfix variety is written after (`myAge++`).

In a simple statement, it doesn't much matter which you use, but in a complex statement, when you are incrementing (or decrementing) a variable and then assigning the result to another variable, it matters very much. The prefix operator is evaluated before the assignment, the postfix is evaluated after.

The semantics of prefix is this: Increment the value and then fetch it. The semantics of postfix is different: Fetch the value and then increment the original.

This can be confusing at first, but if `x` is an integer whose value is `5` and you write

```
int a = ++x;
```

you have told the compiler to increment `x` (making it `6`) and then fetch that value and assign it to `a`. Thus, `a` is now `6` and `x` is now `6`.

If, after doing this, you write

```
int b = x++;
```

you have now told the compiler to fetch the value in `x` (`6`) and assign it to `b`, and then go back and increment `x`. Thus, `b` is now `6`, but `x` is now `7`. Listing below shows the use and implications of both types.

A demonstration of prefix and postfix operators.

Source Code:*PrePost.cpp*

```
 1  // Listing demonstrates use of prefix and
 2  //postfix increment and decrement operators
 3  #include <iostream>
 4  using namespace std;
 5  int main()
 6  {
 7    int myAge = 39;      // initialize two integers
 8    int yourAge = 39;
 9    cout << "I am: " << myAge << " years old.\n";
10    cout << "You are: " << yourAge << " years old\n";
11    myAge++;          // postfix increment
12    ++yourAge;        // prefix increment
13    cout << "One year passes...\n";
14    cout << "I am: " << myAge << " years old.\n";
15    cout << "You are: " << yourAge << " years old\n";
16    cout << "Another year passes\n";
17    cout << "I am: " << myAge++ << " years old.\n";
18    cout << "You are: " << ++yourAge << " years old\n";
19    cout << "Let's print it again.\n";
20    cout << "I am: " << myAge << " years old.\n";
21    cout << "You are: " << yourAge << " years old\n";
22    return 0;
23  }
```

```
Output
I am      39 years old
You are   39 years old
One year passes
I am      40 years old
You are   40 years old
Another year passes
I am      40 years old
You are   41 years old
Let's print it again
I am      41 years old
You are   41 years old
```

Analysis: On lines 7 and 8, two integer variables are declared, and each is initialized with the value `39`. Their values are printed on lines 9 and 10.

On line 11, `myAge` is incremented using the postfix increment operator, and on line 12, `yourAge` is incremented using the prefix increment operator. The results are printed on lines 14 and 15, and they are identical (both `40`).

On line 17, `myAge` is incremented as part of the printing statement, using the postfix increment operator. Because it is postfix, the increment happens after the print, and so the value `40` is printed again. In contrast, on line 18, `yourAge` is incremented using the prefix increment operator. Thus, it is incremented before being printed, and the value displays as `41`.

Finally, on lines 20 and 21, the values are printed again. Because the increment statement has completed, the value in `myAge` is now `41`, as is the value in `yourAge`.

Precedence

In the complex statement

```
x = 5 + 3 * 8;
```

which is performed first, the addition or the multiplication? If the addition is performed first, the answer is 8 * 8, or 64. If the multiplication is performed first, the answer is 5 + 24, or 29.

Every operator has a precedence value, and the complete list is shown in Appendix A, "Operator Precedence." Multiplication has higher precedence than addition, and thus the value of the expression is 29.

When two mathematical operators have the same precedence, they are performed in left-to-right order. Thus

```
x = 5 + 3 + 8 * 9 + 6 * 4;
```

is evaluated multiplication first, left to right. Thus, 8*9 = 72, and 6*4 = 24. Now the expression is essentially

```
x = 5 + 3 + 72 + 24;
```

Now the addition, left to right, is 5 + 3 = 8; 8 + 72 = 80; 80 + 24 = 104.

Be careful with this. Some operators, such as assignment, are evaluated in right-to-left order! In any case, what if the precedence order doesn't meet your needs? Consider the expression

```
TotalSeconds = NumMinutesToThink + NumMinutesToType * 60
```

In this expression, you do not want to multiply the `NumMinutesToType` variable by 60 and then add it to `NumMinutesToThink`. You want to add the two variables to get the total number of minutes, and then you want to multiply that number by 60 to get the total seconds.

In this case, you use parentheses to change the precedence order. Items in parentheses are evaluated at a higher precedence than any of the mathematical operators. Thus

```
TotalSeconds = (NumMinutesToThink + NumMinutesToType) * 60
```

will accomplish what you want.

Nesting Parentheses

For complex expressions, you might need to nest parentheses one within another. For example, you might need to compute the total seconds and then compute the total number of people who are involved before multiplying seconds times people:

```
TotalPersonSeconds = ( ( (NumMinutesToThink + NumMinutesToType) * 60) *
      (PeopleInTheOffice + PeopleOnVacation) )
```

This complicated expression is read from the inside out. First, `NumMinutesToThink` is added to `NumMinutesToType`, because these are in the innermost parentheses. Then this sum is multiplied by 60. Next, `PeopleInTheOffice` is added to `PeopleOnVacation`. Finally, the total number of people found is multiplied by the total number of seconds.

This example raises an important related issue. This expression is easy for a computer to understand, but very difficult for a human to read, understand, or modify. Here is the same expression rewritten, using some temporary integer variables:

```
TotalMinutes = NumMinutesToThink + NumMinutesToType;
TotalSeconds = TotalMinutes * 60;
TotalPeople = PeopleInTheOffice + PeopleOnVacation;
TotalPersonSeconds = TotalPeople * TotalSeconds;
```

This example takes longer to write and uses more temporary variables than the preceding example, but it is far easier to understand. Add a comment at the top to explain what this code does, and change the `60` to a symbolic constant. You then will have code that is easy to understand and maintain.

DO remember that expressions have a value. **DO** use the prefix operator (`++`variable) to increment or decrement the variable before it is used in the expression. **DO** use the postfix operator (variable`++`) to increment or decrement the variable after it is used. **DO** use parentheses to change the order of precedence. **DON'T** nest too deeply, because the expression becomes hard to understand and maintain.

The Nature of Truth

In C++, zero is considered false, and all other values are considered true, although true is usually represented by 1. Thus, if an expression is false, it is equal to zero, and if an expression is equal to zero, it is false. If a statement is true, all you know is that it is nonzero, and any nonzero statement is true.

Relational Operators

The relational operators are used to determine whether two numbers are equal, or if one is greater or less than the other. Every relational statement evaluates to either `1` (`TRUE`) or `0` (`FALSE`). The relational operators are presented later, in Table (The Relational Operators).

If the integer variable `myAge` has the value `39`, and the integer variable `yourAge` has the value `40`, you can determine whether they are equal by using the relational "equals" operator:

```
myAge == yourAge;  // is the value in myAge the same as in yourAge?
```

This expression evaluates to `0`, or `false`, because the variables are not equal. The expression

```
myAge > yourAge;  // is myAge greater than yourAge?
```

evaluates to `0` or `false`.

WARNING: Many novice C++ programmers confuse the assignment operator (`=`) with the equals operator (`==`). This can create a nasty bug in your program.

There are six relational operators: equals (==), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), and not equals (!=). Table below shows each relational operator, its use, and a sample code use.

Table: ***The Relational Operators.***

Name	*Operator*	*Sample*	*Evaluates*
Equals	==	`100 == 50;`	`False`
		`50 == 50;`	`True`
Not Equals	!=	`100 != 50;`	`True`
		`50 != 50;`	`False`
Greater Than	>	`100 > 50;`	`True`
		`50 > 50;`	`False`
Greater Than or Equals	>=	`100 >= 50;`	`True`
		`50 >= 50;`	`True`
Less Than	<	`100 < 50;`	`False`
		`50 < 50;`	`False`
Less Than or Equals	<=	`100 <= 50;`	`False`
		`50 <= 50;`	`True`

DO remember that relational operators return the value `1` (`true`) or `0` (`false`). **DON'T** confuse the assignment operator (=) with the equals relational operator (==). This is one of the most common C++ programming mistakes--be on guard for it.

The if Statement

Normally, your program flows along line by line in the order in which it appears in your source code. The `if` statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result.

The simplest form of an `if` statement is this:

```
if (expression)
    statement;
```

The expression in the parentheses can be any expression at all, but it usually contains one of the relational expressions. If the expression has the value `0`, it is considered false, and the statement is skipped. If it has any nonzero value, it is considered true, and the statement is executed. Consider the following example:

```
if (bigNumber > smallNumber)
     bigNumber = smallNumber;
```

This code compares `bigNumber` and `smallNumber`. If `bigNumber` is larger, the second line sets its value to the value of `smallNumber`.

Because a block of statements surrounded by braces is exactly equivalent to a single statement, the following type of branch can be quite large and powerful:

```
if (expression)
{
     statement1;
     statement2;
     statement3;
}
```

Here's a simple example of this usage:

```
if (bigNumber > smallNumber)
{
     bigNumber = smallNumber;
     cout << "bigNumber: " << bigNumber << "\n";
     cout << "smallNumber: " << smallNumber << "\n";
}
```

This time, if `bigNumber` is larger than `smallNumber`, not only is it set to the value of `smallNumber`, but an informational message is printed. Listing below shows a more detailed example of branching based on relational operators.

A demonstration of branching based on relational operators.

Source Code:*Else.cpp*

```
 1  // Listing demonstrates if statement used with relational operators
 2  #include <iostream>
 3  using namespace std;
 4  int main()
 5  {
 6    int RedSoxScore, YankeesScore;
 7    cout << "Enter the score for the Red Sox: ";
 8    cin >> RedSoxScore;
 9
10    cout << "Enter the score for the Yankees: ";
11    cin >> YankeesScore;
12
13    if (RedSoxScore > YankeesScore)
14          cout << "Go Sox!\n";
15
16    if (RedSoxScore < YankeesScore)
17    {
18          cout << "Go Yankees!\n";
19          cout << "Happy days in New York!\n";
20    }
21
```

```
22    if (RedSoxScore == YankeesScore)
23    {
24          cout << "A tie? Naah, can't be.\n";
25          cout << "Give me the real score for the Yanks: ";
26          cin >> YankeesScore;
27
28          if (RedSoxScore > YankeesScore)
29                cout << "Knew it! Go Sox!";
30
31          if (YankeesScore > RedSoxScore)
32                cout << "Knew it! Go Yanks!";
33
34          if (YankeesScore == RedSoxScore)
35                cout << "Wow, it really was a tie!";
36    }
37    cout << "\nThanks for telling me.\n";
38    return 0;
39  }
```

Output

```
Enter the score for the Red Sox: 10
Enter the score for the Yankees: 10
A tie? Naah, can't be
Give me the real score for the Yanks: 8
Knew it! Go Sox!
Thanks for telling me.
```

Analysis: This program asks for user input of scores for two baseball teams, which are stored in integer variables. The variables are compared in the `if` statement on lines 13, 16, and 22. If one score is higher than the other, an informational message is printed. If the scores are equal, the block of code that begins on line 22 and ends on line 36 is entered. The second score is requested again, and then the scores are compared again.

Note that if the initial Yankees score was higher than the Red Sox score, the `if` statement on line 13 would evaluate as `FALSE`, and line 14 would not be invoked. The test on line 16 would evaluate as `true`, and the statements on lines 18 and 19 would be invoked. Then the `if` statement on line 22 would be tested, and this would be false (if line 16 was true). Thus, the program would skip the entire block, falling through to line 37.

In this example, getting a true result in one `if` statement does not stop other `if` statements from being tested.

Indentation Styles

Previous listing shows one style of indenting `if` statements. Nothing is more likely to create a religious war, however, than to ask a group of programmers what is the best style for brace alignment. Although there are dozens of variations, these appear to be the favorite three:

- Putting the initial brace after the condition and aligning the closing brace under the `if` to close the statement block.

```
if (expression){
     statements
}
```

- Aligning the braces under the `if` and indenting the statements.

```
if (expression)
{
    statements
}
```

- Indenting the braces and statements.

```
if (expression)
   {
   statements
   }
```

This course uses the middle alternative, because I find it easier to understand where blocks of statements begin and end if the braces line up with each other and with the condition being tested. Again, it doesn't matter much which style you choose, as long as you are consistent with it.

else

Often your program will want to take one branch if your condition is true, another if it is false. In Previous listing, you wanted to print one message (`Go Sox!`) if the first test (`RedSoxScore > Yankees`) evaluated `TRUE`, and another message (`Go Yanks!`) if it evaluated `FALSE`.

The method shown so far, testing first one condition and then the other, works fine but is a bit cumbersome. The keyword `else` can make for far more readable code:

```
if (expression)
     statement;
else
     statement;
```

Listing below demonstrates the use of the keyword `else`.

Demonstrating the else keyword.

Source Code:*Else.cpp*

```
 1  // Listing demonstrates if statement with else clause
 2  #include <iostream>
 3  using namespace std;
 4  int main()
 5  {
 6    int firstNumber, secondNumber;
 7    cout << "Please enter a big number: ";
 8    cin >> firstNumber;
 9    cout << "\nPlease enter a smaller number: ";
10    cin >> secondNumber;
```

```
11    if (firstNumber > secondNumber)
12          cout << "\nThanks!\n";
13    else
14          cout << "\nOops. The second is bigger!";
15
16    return 0;
17 }
```

Output

```
Please enter a big number: 10

Please enter a smaller number: 12

Oops. The second is bigger!
```

Analysis: The `if` statement on line 11 is evaluated. If the condition is true, the statement on line 12 is run; if it is false, the statement on line 14 is run. If the `else` clause on line 13 were removed, the statement on line 14 would run whether or not the `if` statement was true. Remember, the `if` statement ends after line 12. If the `else` was not there, line 14 would just be the next line in the program. Remember that either or both of these statements could be replaced with a block of code in braces.

The if Statement

The syntax for the `if` statement is as follows: Form 1

```
if (expression)
    statement;
next statement;
```

If the expression is evaluated as `TRUE`, the statement is executed and the program continues with the next statement. If the expression is not true, the statement is ignored and the program jumps to the next statement. Remember that the statement can be a single statement ending with a semicolon or a block enclosed in braces. Form 2

```
if (expression)
    statement1;
else
    statement2;
next statement;
```

If the expression evaluates `TRUE`, `statement1` is executed; otherwise, `statement2` is executed. Afterwards, the program continues with the next statement.

```
if (SomeValue < 10)
  cout << "SomeValue is less than 10");
else
  cout << "SomeValue is not less than 10!");
cout << "Done." << endl;
```

Advanced if Statements

It is worth noting that any statement can be used in an `if` or `else` clause, even another `if` or `else` statement. Thus, you might see complex `if` statements in the following form:

```
if (expression1)
{
    if (expression2)
        statement1;
    else
    {
        if (expression3)
            statement2;
        else
            statement3;
    }
}
else
    statement4;
```

This cumbersome `if` statement says, "If expression1 is true and expression2 is true, execute statement1. If expression1 is true but expression2 is not true, then if expression3 is true execute statement2. If expression1 is true but expression2 and expression3 are false, execute statement3.

Finally, if expression1 is not true, execute statement4." As you can see, complex `if` statements can be confusing!

Listing below gives an example of such a complex `if` statement.

A complex, nested if statement.

Source Code: ***NestedIf.cpp***

```
 1  // A complex nested if statement
 2  #include <iostream>
 3  using namespace std;
 4  int main()
 5  {
 6    // Ask for two numbers
 7    // Assign the numbers to bigNumber and littleNumber
 8    // If bigNumber is bigger than littleNumber,
 9    // see if they are evenly divisible
10    // If they are, see if they are the same number
11
12    int firstNumber, secondNumber;
13    cout << "Enter two numbers.\nFirst: ";
14    cin >> firstNumber;
15    cout << "\nSecond: ";
16    cin >> secondNumber;
17    cout << "\n\n";
18
19    if (firstNumber >= secondNumber)
20    {
21          if ( (firstNumber % secondNumber) == 0) // evenly divisible?
22          {
23                  if (firstNumber == secondNumber)
24                          cout << "They are the same!\n";
25                  else
26                          cout << "They are evenly divisible!\n";
27          }
28          else
29                  cout << "They are not evenly divisible!\n";
30    }
31    else
32          cout << "Hey! The second one is larger!\n";
33    return 0;
34  }
```

Output

```
Enter two numbers.
First: 10

Second: 2

They are evenly divisible!
```

Analysis: Two numbers are prompted for one at a time, and then compared. The first `if` statement, on line 19, checks to ensure that the first number is greater than or equal to the second. If not, the `else` clause on line 31 is executed.

If the first `if` is true, the block of code beginning on line 20 is executed, and the second `if` statement is tested, on line 21. This checks to see whether the first number modulo the second number yields no remainder. If so, the numbers are either evenly divisible or equal. The `if` statement on line 23 checks for equality and displays the appropriate message either way.

If the `if` statement on line 21 fails, the `else` statement on line 28 is executed.

Using Braces in Nested if Statements

Although it is legal to leave out the braces on `if` statements that are only a single statement, and it is legal to nest `if` statements, such as

```
if (x > y)                // if x is bigger than y
    if (x < z)            // and if x is smaller than z
        x = y;           // then set x to the value in z
```

When writing large nested statements, this can cause enormous confusion. Remember, whitespace and indentation are a convenience for the programmer; they make no difference to the compiler. It is easy to confuse the logic and inadvertently assign an `else` statement to the wrong `if` statement. Listing below illustrates this problem.

A demonstration of why braces help clarify which else statement goes with which if statement.

Source Code:*Braces.cpp*

```
1  //Listing demonstrates why braces are important in nested if statements
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6    int x;
7    cout << "Enter a number less than 10 or greater than 100: ";
8    cin >> x;
9    cout << "\n";
10
11   if (x > 10)
12         if (x > 100)
13                cout << "More than 100, Thanks!\n";
14         else                              // not the else intended!
15                cout << "Less than 10, Thanks!\n";
16
17   return 0;
18 }
```

Output

```
Enter a number less than 10 or greater than 100: 20

Less than 10, Thanks!
```

Analysis: The programmer intended to ask for a number between 10 and 100, check for the correct value, and then print a thank-you note.

If the `if` statement on line 11 evaluates `TRUE`, the following statement (line 12) is executed. In this case, line 12 executes when the number entered is greater than 10. Line 12 contains an `if` statement also. This `if` statement evaluates `TRUE` if the number entered is greater than 100. If the number is not greater than 100, the statement on line 13 is executed.

If the number entered is less than or equal to 10, the `if` statement on line 10 evaluates to `FALSE`. Program control goes to the next line following the `if` statement, in this case line 16. If you enter a number less than 10, the output is as follows:

```
Enter a number less than 10 or greater than 100: 9
```

The `else` clause on line 14 was clearly intended to be attached to the `if` statement on line 11, and thus is indented accordingly. Unfortunately, the `else` statement is really attached to the `if` statement on line 12, and thus this program has a subtle bug.

It is a subtle bug because the compiler will not complain. This is a legal C++ program, but it just doesn't do what was intended. Further, most of the times the programmer tests this program, it will appear to work. As long as a number that is greater than 100 is entered, the program will seem to work just fine.

Listing below fixes the problem by putting in the necessary braces.

A demonstration of the proper use of braces with an if statement

Source Code: ***Braces2.cpp***

```
 1  // Listing demonstrates proper use of braces in nested if statements
 2  #include <iostream>
 3  using namespace std;
 4  int main()
 5  {
 6    int x;
 7    cout << "Enter a number less than 10 or greater than 100: ";
 8    cin >> x;
 9    cout << "\n";
10
11    if (x > 10)
12    {
13          if (x > 100)
14                cout << "More than 100, Thanks!\n";
15          }
16          else                              // not the else intended!
17                cout << "Less than 10, Thanks!\n";
18    return 0;
19  }
```

Output

```
Enter a number less than 10 or greater than 100: 20
```

Analysis: The braces on lines 12 and 15 make everything between them into one statement, and now the `else` on line 16 applies to the `if` on line 11 as intended.

The user typed `20`, so the `if` statement on line 11 is true; however, the `if` statement on line 13 is false, so nothing is printed. It would be better if the programmer put another `else` clause after line 14 so that errors would be caught and a message printed.

NOTE: The programs shown in this course are written to demonstrate the particular issues being discussed. They are kept intentionally simple; there is no attempt to "bulletproof" the code to protect against user error. In professional-quality code, every possible user error is anticipated and handled gracefully.

Logical Operators

Often you want to ask more than one relational question at a time. "Is it true that x is greater than y, and also true that y is greater than z?" A program might need to determine that both of these conditions are true, or that some other condition is true, in order to take an action.

Imagine a sophisticated alarm system that has this logic: "If the door alarm sounds AND it is after six p.m. AND it is NOT a holiday, OR if it is a weekend, then call the police." C++'s three logical operators are used to make this kind of evaluation. These operators are listed in Table below.

Table: *The Logical Operators.*

Operator	*Symbol*	*Example*
AND	&&	expression1 && expression2
OR	||	expression1 || expression2
XOR	^	expression1 ^ expression2
NOT	!	!expression

Logical AND

A logical `AND` statement evaluates two expressions, and if both expressions are true, the logical `AND` statement is true as well. If it is true that you are hungry, AND it is true that you have money, THEN it is true that you can buy lunch. Thus, the following would evaluates `TRUE` if both `x` and `y` are equal to 5, and it would evaluate `FALSE` if either one is not equal to 5. Note that both sides must be true for the entire expression to be true. Note that the logical `AND` is two `&&` symbols. A single `&` symbol is a different operator.

```
if ( (x == 5) && (y == 5) )
```

Logical OR

A logical OR statement evaluates two expressions. If either one is true, the expression is true. If you have money OR you have a credit card, you can pay the bill. You don't need both money and a credit card; you need only one, although having both would be fine as well. Thus following evaluates TRUE if either x or y is equal to 5, or if both are. Note that the logical OR is two || symbols. A single | symbol is a different operator.

```
if ( (x == 5) || (y == 5) )
```

Logical NOT

A logical NOT statement evaluates true if the expression being tested is false. Again, if the expression being tested is false, the value of the test is TRUE! Thus

```
if ( !(x == 5) )
```

is true only if x is not equal to 5. This is exactly the same as writing

```
if (x != 5)
```

Relational Precedence

Relational operators and logical operators, being C++ expressions, each return a value: 1 (TRUE) or 0 (FALSE). Like all expressions, they have a precedence order (see Appendix A) that determines which relations are evaluated first. This fact is important when determining the value of the statement

```
if ( x > 5 &&  y > 5  || z > 5)
```

It might be that the programmer wanted this expression to evaluate TRUE if both x and y are greater than 5 or if z is greater than 5. On the other hand, the programmer might have wanted this expression to evaluate TRUE only if x is greater than 5 and if it is also true that either y is greater than 5 or z is greater than 5.

If x is 3, and y and z are both 10, the first interpretation will be true (z is greater than 5, so ignore x and y), but the second will be false (it isn't true that both x and y are greater than 5 nor is it true that z is greater than 5).

Although precedence will determine which relation is evaluated first, parentheses can both change the order and make the statement clearer:

```
if (  (x > 5)  && (y > 5 ||  z > 5) )
```

Using the values from earlier, this statement is false. Because it is not true that x is greater than 5, the left side of the AND statement fails, and thus the entire statement is false. Remember that an AND

statement requires that both sides be true--something isn't both "good tasting" AND "good for you" if it isn't good tasting.

NOTE: It is often a good idea to use extra parentheses to clarify what you want to group. Remember, the goal is to write programs that work and that are easy to read and understand.

More About Truth and Falsehood

In C++, zero is false, and any other value is true. Because an expression always has a value, many C++ programmers take advantage of this feature in their `if` statements. A statement such as

```
if (x)            // if x is true (nonzero)
    x = 0;
```

can be read as "*If `x` has a nonzero value, set it to 0.*" This is a bit of a cheat; it would be clearer if written

```
if (x != 0)       // if x is nonzero
    x = 0;
```

Both statements are legal, but the latter is clearer. It is good programming practice to reserve the former method for true tests of logic, rather than for testing for nonzero values.

These two statements also are equivalent:

```
if (!x)           // if x is false (zero)
if (x == 0)       // if x is zero
```

The second statement, however, is somewhat easier to understand and is more explicit.

DO put parentheses around your logical tests to make them clearer and to make the precedence explicit. **DO** use braces in nested `if` statements to make the `else` statements clearer and to avoid bugs. **DON'T** use `if(x)` as a synonym for `if(x != 0)`; the latter is clearer. **DON'T** use `if(!x)` as a synonym for `if(x == 0)`; the latter is clearer.

NOTE: It is common to define your own enumerated Boolean (logical) type with `enum Bool {FALSE, TRUE};`. This serves to set `FALSE` to `0` and `TRUE` to `1`.

Conditional (Ternary) Operator

The conditional operator (`?:`) is C++'s only ternary operator; that is, it is the only operator to take three terms.

The conditional operator takes three expressions and returns a value:

```
(expression1) ? (expression2) : (expression3)
```

This line is read as "If expression1 is true, return the value of expression2; otherwise, return the value of expression3." Typically, this value would be assigned to a variable.

Listing below shows an `if` statement rewritten using the conditional operator.

A demonstration of the conditional operator.

Source Code: ***ConditionalOperators.cpp***

```
 1  // Listing demonstrates the conditional operator
 2  #include <iostream>
 3  using namespace std;
 4  int main()
 5  {
 6    int x, y, z;
 7    cout << "Enter two numbers.\n";
 8    cout << "First: ";
 9    cin >> x;
10    cout << "\nSecond: ";
11    cin >> y;
12    cout << "\n";
13
14    if (x > y)
15          z = x;
16    else
17          z = y;
18
19    cout << "z: " << z;
20    cout << "\n";
21
22    z =  (x > y) ? x : y;
23
24    cout << "z: " << z;
25    cout << "\n";
26    return 0;
27  }
```

```
Output
Enter two numbers.
First: 5

Second: 8

z: 8
z: 8
```

Analysis: Three integer variables are created: `x`, `y`, and `z`. The first two are given values by the user. The `if` statement on line 14 tests to see which is larger and assigns the larger value to `z`. This value is printed on line 19.

The conditional operator on line 22 makes the same test and assigns `z` the larger value. It is read like this: "If `x` is greater than `y`, return the value of `x`; otherwise, return the value of `y`." The value returned is assigned to `z`. That value is printed on line 24. As you can see, the conditional statement is a shorter equivalent to the `if...else` statement.

Functions

Although object-oriented programming has shifted attention from functions and toward objects, functions nonetheless remain a central component of any program. Today you will learn

- What a function is and what its parts are.
- How to declare and define functions.
- How to pass parameters into functions.
- How to return a value from a function.

What Is a Function?

A function is, in effect, a subprogram that can act on data and return a value. Every C++ program has at least one function, `main()`. When your program starts, `main()` is called automatically. `main()` might call other functions, some of which might call still others.

Each function has its own name, and when that name is encountered, the execution of the program branches to the body of that function. When the function returns, execution resumes on the next line of the calling function. This flow is illustrated in Figure below.

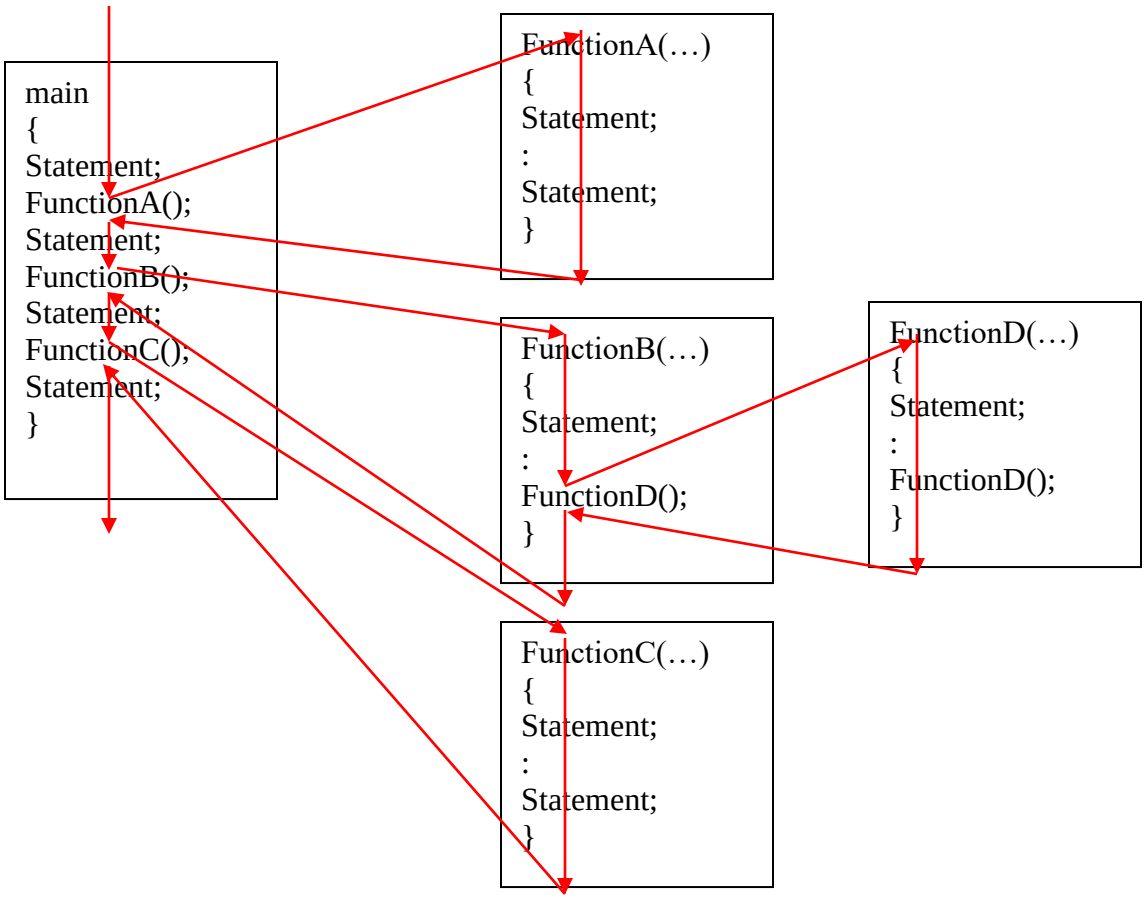


Figure: *Illustration of flow*

When a program calls a function, execution switches to the function and then resumes at the line after the function call. Well-designed functions perform a specific and easily understood task. Complicated tasks should be broken down into multiple functions, and then each can be called in turn.

Functions come in two varieties: user-defined and built-in. Built-in functions are part of your compiler package--they are supplied by the manufacturer for your use.

Declaring and Defining Functions

Using functions in your program requires that you first declare the function and that you then define the function. The declaration tells the compiler the name, return type, and parameters of the function. The definition tells the compiler how the function works. No function can be called from any other function that hasn't first been declared. The declaration of a function is called its prototype.

Declaring the Function

There are three ways to declare a function:

- Write your prototype into a file, and then use the `#include` directive to include it in your program.
- Write the prototype into the file in which your function is used.
- Define the function before it is called by any other function. When you do this, the definition acts as its own declaration.

Although you can define the function before using it, and thus avoid the necessity of creating a function prototype, this is not good programming practice for three reasons.

First, it is a bad idea to require that functions appear in a file in a particular order. Doing so makes it hard to maintain the program as requirements change.

Second, it is possible that function `A()` needs to be able to call function `B()`, but function `B()` also needs to be able to call function `A()` under some circumstances. It is not possible to define function `A()` before you define function `B()` and also to define function `B()` before you define function `A()`, so at least one of them must be declared in any case.

Third, function prototypes are a good and powerful debugging technique. If your prototype declares that your function takes a particular set of parameters, or that it returns a particular type of value, and then your function does not match the prototype, the compiler can flag your error instead of waiting for it to show itself when you run the program.

Function Prototypes

Many of the built-in functions you use will have their function prototypes already written in the files you include in your program by using `#include`. For functions you write yourself, you must include the prototype.

The function prototype is a statement, which means it ends with a semicolon. It consists of the function's return type, name, and parameter list.

The parameter list is a list of all the parameters and their types, separated by commas. Figure below illustrates the parts of the function prototype.

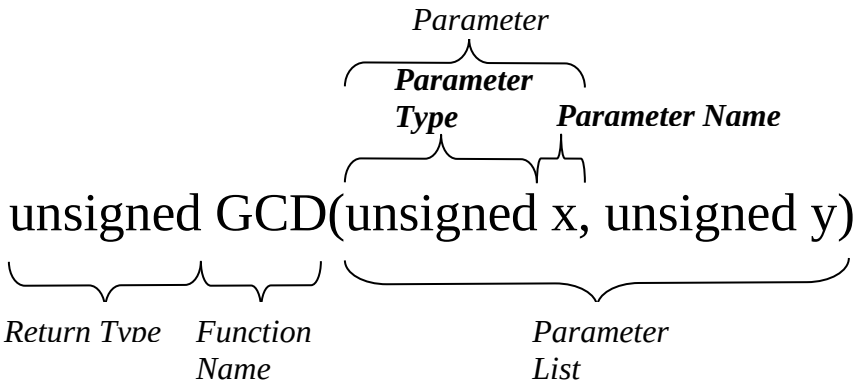


Figure: *Parts of a function prototype.*

The function prototype and the function definition must agree exactly about the return type, the name, and the parameter list. If they do not agree, you will get a compile-time error. Note, however, that the function prototype does not need to contain the names of the parameters, just their types. A prototype that looks like this is perfectly legal:

```
long Area(int, int);
```

This prototype declares a function named `Area()` that returns a `long` and that has two parameters, both integers. Although this is legal, it is not a good idea. Adding parameter names makes your prototype clearer. The same function with named parameters might be

```
long Area(int length, int width);
```

It is now obvious what this function does and what the parameters are.

Note that all functions have a return type. If none is explicitly stated, the return type defaults to `int`. Your programs will be easier to understand, however, if you explicitly declare the return type of every function, including `main()`. Listing below demonstrates a program that includes a function prototype for the `Area()` function.

A function declaration and the definition and use of that function.

Source Code: *FunctionPrototype.cpp*

```
 1  // Listing demonstrates the use of function prototypes #include
 2  typedef unsigned short USHORT;
 3  <iostream>
 4  using namespace std;
 5  USHORT FindArea(USHORT length, USHORT width); //function prototype
 6
 7  int main()
 8  {
 9    USHORT lengthOfYard;
10    USHORT widthOfYard;
11    USHORT areaOfYard;
12
13    cout << "\nHow wide is your yard? ";
14    cin >> widthOfYard;
15    cout << "\nHow long is your yard? ";
16    cin >> lengthOfYard;
17
18    areaOfYard= FindArea(lengthOfYard,widthOfYard);
19
20    cout << "\nYour yard is ";
21    cout << areaOfYard;
22    cout << " square feet\n";
23    return 0;
24  }
25
26  USHORT FindArea(USHORT l, USHORT w)
27  {
28    return l * w;
29  }
```

Output

```
How wide is your yard? 100

How long is your yard? 200

Your yard is 20000 square feet
```

Analysis: The prototype for the `FindArea()` function is on line 5. Compare the prototype with the definition of the function on line 26. Note that the name, the return type, and the parameter types are the same. If they were different, a compiler error would have been generated. In fact, the only required difference is that the function prototype ends with a semicolon and has no body. Also note that the parameter names in the prototype are `length` and `width`, but the parameter names in the definition are `l` and `w`. As discussed, the names in the prototype are not used; they are there as information to the programmer. When they are included, they should match the implementation when possible. This is a matter of good programming style and reduces confusion, but it is not required, as you see here.

The arguments are passed in to the function in the order in which they are declared and defined, but there is no matching of the names. Had you passed in `widthOfYard`, followed by `lengthOfYard`, the `FindArea()` function would have used the value in `widthOfYard` for `length` and `lengthOfYard` for

`width`. The body of the function is always enclosed in braces, even when it consists of only one statement, as in this case.

Defining the Function

The definition of a function consists of the function header and its body. The header is exactly like the function prototype, except that the parameters must be named, and there is no terminating semicolon.

The body of the function is a set of statements enclosed in braces. Figure 5.3 shows the header and body of a function.

```
unsigned GCD(unsigned x, unsigned y)
{unsigned tmp;

        while(y){
                tmp = x;
                x = y;
                y = tmp % y;
        }
        return x;
}
```

Figure: *The header and body of a function.*

Functions

Function Prototype Syntax

```
return_type function_name ( [type [parameterName]]...);
```

Function Definition Syntax

```
return_type function_name ( [type parameterName]...)
{
    statements;
}
```

A function prototype tells the compiler the return type, name, and parameter list. Functions are not required to have parameters, and if they do, the prototype is not required to list their names, only their types. A prototype always ends with a semicolon (;). A function definition must agree in return type and parameter list with its prototype. It must provide names for all the parameters, and the body of the function definition must be surrounded by braces. All statements within the body of the function must be terminated with semicolons, but the function itself is not ended with a semicolon; it ends with a closing brace. If the function returns a value, it should end with a `return` statement, although `return` statements can legally appear anywhere in the body of the function. Every function has a return type. If one is not explicitly designated, the return type will be `int`. Be sure to give every function an explicit return type. If a function does not return a value, its return type will be `void`.

Function Prototype Examples

```
long FindArea(long length, long width); // returns long, has two parameters
void PrintMessage(int messageNumber); // returns void, has one parameter
int GetChoice();                      // returns int, has no parameters

BadFunction();                        // returns int, has no parameters
```

Function Definition Examples

```
long Area(long l, long w)
{
     return l * w;
}

void PrintMessage(int whichMsg)
{
     if (whichMsg == 0)
          cout << "Hello.\n";
     if (whichMsg == 1)
          cout << "Goodbye.\n";
     if (whichMsg > 1)
          cout << "I'm confused.\n";
}
```

Execution of Functions

When you call a function, execution begins with the first statement after the opening brace (`{`). Branching can be accomplished by using the `if` statement. Functions can also call other functions and can even call themselves (see the section "Recursion," later in this chapter).

Local Variables

Not only can you pass in variables to the function, but you also can declare variables within the body of the function. This is done using local variables, so named because they exist only locally within the function itself. When the function returns, the local variables are no longer available.

Local variables are defined like any other variables. The parameters passed in to the function are also considered local variables and can be used exactly as if they had been defined within the body of the function. Listing below is an example of using parameters and locally defined variables within a function.

The use of local variables and parameters.

Source Code: ***LocalVariables.cpp***

```
 1  <iostream>
 2  using namespace std;
 3  float Convert(float);
 4  int main()
 5  {
 6    float TempFer;
 7    float TempCel;
 8
 9    cout << "Please enter the temperature in Fahrenheit: ";
10    cin >> TempFer;
11    TempCel = Convert(TempFer);
12    cout << "\nHere's the temperature in Celsius: ";
13    cout << TempCel << endl;
14    return 0;
15  }
16
17  float Convert(float TempFer)
18  {
19    float TempCel;
20    TempCel = ((TempFer - 32) * 5) / 9;
21    return TempCel;
22  }
```

Output

```
Please enter the temperature in Fahrenheit: 212

Here's the temperature in Celsius: 100

Please enter the temperature in Fahrenheit: 32

Here's the temperature in Celsius: 0

Please enter the temperature in Fahrenheit: 85

Here's the temperature in Celsius: 29.4444
```

Analysis: On lines 6 and 7, two `float` variables are declared, one to hold the temperature in Fahrenheit and one to hold the temperature in degrees Celsius. The user is prompted to enter a Fahrenheit temperature on line 9, and that value is passed to the function `Convert()`. Execution jumps to the first line of the function `Convert()` on line 19, where a local variable, also named `TempCel`, is declared. Note that this local variable is not the same as the variable `TempCel` on line 7. This variable exists only within the function `Convert()`. The value passed as a parameter, `TempFer`, is also just a local copy of the variable passed in by `main()`.

This function could have named the parameter `FerTemp` and the local variable `CelTemp`, and the program would work equally well. You can enter these names again and recompile the program to see this work.

The local function variable `TempCel` is assigned the value that results from subtracting 32 from the parameter `TempFer`, multiplying by 5, and then dividing by 9. This value is then returned as the return value of the function, and on line 11 it is assigned to the variable `TempCel` in the `main()` function. The value is printed on line 13.

The program is run three times. The first time, the value `212` is passed in to ensure that the boiling point of water in degrees Fahrenheit (212) generates the correct answer in degrees Celsius (100). The second test is the freezing point of water. The third test is a random number chosen to generate a fractional result.

As an exercise, try entering the program again with other variable names as illustrated here:

```
 1  <iostream>
 2  using namespace std;
 3  float Convert(float);
 4  int main()
 5  {
 6    float TempFer;
 7    float TempCel;
 8
 9    cout << "Please enter the temperature in Fahrenheit: ";
10    cin >> TempFer;
11    TempCel = Convert(TempFer);
12    cout << "\nHere's the temperature in Celsius: ";
13    cout << TempCel << endl;
14  }
15
16  float Convert(float Fer)
17  {
18    float Cel;
19    Cel = ((Fer - 32) * 5) / 9;
20    return Cel;
21  }
```

You should get the same results.

New Term: A variable has scope, which determines how long it is available to your program and where it can be accessed. Variables declared within a block are scoped to that block; they can be accessed only within that block and "go out of existence" when that block ends. Global variables have global scope and are available anywhere within your program.

Normally scope is obvious, but there are some tricky exceptions. Currently, variables declared within the header of a `for` loop `(for int i = 0; i<SomeValue; i++)` are scoped to the block in which the `for` loop is created, but there is talk of changing this in the official C++ standard.

None of this matters very much if you are careful not to reuse your variable names within any given function.

Global Variables

Variables defined outside of any function have global scope and thus are available from any function in the program, including `main()`.

Local variables with the same name as global variables do not change the global variables. A local variable with the same name as a global variable hides the global variable, however. If a function has a variable with the same name as a global variable, the name refers to the local variable--not the global--when used within the function. Listing below illustrates these points.

Demonstrating global and local variables.

Source Code: ***GlobalVariables.cpp***

```
 1  <iostream>
 2  using namespace std;
 3  void myFunction();           // prototype
 4  int x = 5, y = 7;            // global variables
 5  int main()
 6  {
 7
 8    cout << "x from main: " << x << "\n";
 9    cout << "y from main: " << y << "\n\n";
10    myFunction();
11    cout << "Back from myFunction!\n\n";
12    cout << "x from main: " << x << "\n";
13    cout << "y from main: " << y << "\n";
14    return 0;
15  }
16
17  void myFunction()
18  {
19    int y = 10;
20
21    cout << "x from myFunction: " << x << "\n";
22    cout << "y from myFunction: " << y << "\n\n";
23  }
```

Output

```
x from main: 5
y from main: 7

x from myFunction: 5
y from myFunction: 10

Back from myFunction!

x from main: 5
y from main: 7
```

Analysis: This simple program illustrates a few key, and potentially confusing, points about local and global variables. On line 4, two global variables, `x` and `y`, are declared. The global variable `x` is initialized with the value `5`, and the global variable `y` is initialized with the value `7`.

On lines 8 and 9 in the function `main()`, these values are printed to the screen. Note that the function `main()` defines neither variable; because they are global, they are already available to `main()`.

When `myFunction()` is called on line 10, program execution passes to line 18, and a local variable, `y`, is defined and initialized with the value `10`. On line 21, `myFunction()` prints the value of the variable `x`, and the global variable `x` is used, just as it was in `main()`. On line 22, however, when the variable name `y` is used, the local variable `y` is used, hiding the global variable with the same name.

The function call ends, and control returns to `main()`, which again prints the values in the global variables. Note that the global variable `y` was totally unaffected by the value assigned to `myFunction()`'s local `y` variable.

Global Variables: A Word of Caution

In C++, global variables are legal, but they are almost never used. C++ grew out of C, and in C global variables are a dangerous but necessary tool. They are necessary because there are times when the programmer needs to make data available to many functions and he does not want to pass that data as a parameter from function to function.

Globals are dangerous because they are shared data, and one function can change a global variable in a way that is invisible to another function. This can and does create bugs that are very difficult to find.

More on Local Variables

Variables declared within the function are said to have "local scope." That means, as discussed, that they are visible and usable only within the function in which they are defined. In fact, in C++ you can define variables anywhere within the function, not just at its top. The scope of the variable is the block in which it is defined. Thus, if you define a variable inside a set of braces within the function, that variable is available only within that block. Listing below illustrates this idea.

Variables scoped within a block.

Source Code: *Scoping.cpp*

```
 1  // Listing demonstrates variables scoped within a block
 2  #include <iostream>
 3
 4  using namespace std;
 5
 6  void myFunc();
 7
 8  int main()
 9  {
10    int x = 5;
11    cout << "\nIn main x is: " << x;
12
13    myFunc();
14
15    cout << "\nBack in main, x is: " << x;
16    return 0;
17  }
```

```
18
19  void myFunc()
20  {
21
22    int x = 8;
23    cout << "\nIn myFunc, local x: " << x << endl;
24
25    {
26          cout << "\nIn block in myFunc, x is: " << x;
27
28          int x = 9;
29
30          cout << "\nVery local x: " << x;
31    }
32
33    cout << "\nOut of block, in myFunc, x: " << x << endl;
34  }
```

Output

```
In main x is: 5
In myFunc, local x: 8

In block in myFunc, x is: 8
Very local x: 9
Out of block, in myFunc, x: 8

Back in main, x is: 5
```

Analysis: This program begins with the initialization of a local variable, `x`, on line 10, in `main()`. The printout on line 11 verifies that `x` was initialized with the value `5`. `MyFunc()` is called, and a local variable, also named `x`, is initialized with the value `8` on line 22. Its value is printed on line 23.

A block is started on line 25, and the variable `x` from the function is printed again on line 26. A new variable also named `x`, but local to the block, is created on line 28 and initialized with the value `9`.

The value of the newest variable `x` is printed on line 30. The local block ends on line 31, and the variable created on line 28 goes "out of scope" and is no longer visible.

When `x` is printed on line 33, it is the `x` that was declared on line 22. This `x` was unaffected by the `x` that was defined on line 28; its value is still `8`.

On line 34, `MyFunc()` goes out of scope, and its local variable `x` becomes unavailable. Execution returns to line 15, and the value of the local variable `x`, which was created on line 10, is printed. It was unaffected by either of the variables defined in `MyFunc()`.

Needless to say, this program would be far less confusing if these three variables were given unique names!

Function Statements

There is virtually no limit to the number or types of statements that can be in a function body. Although you can't define another function from within a function, you can call a function, and of course `main()` does just that in nearly every C++ program. Functions can even call themselves, which is discussed soon, in the section on recursion.

Although there is no limit to the size of a function in C++, well-designed functions tend to be small. Many programmers advise keeping your functions short enough to fit on a single screen so that you can see the entire function at one time. This is a rule of thumb, often broken by very good programmers, but a smaller function is easier to understand and maintain.

Each function should carry out a single, easily understood task. If your functions start getting large, look for places where you can divide them into component tasks.

Function Arguments

Function arguments do not have to all be of the same type. It is perfectly reasonable to write a function that takes an integer, two `long`s, and a character as its arguments. Any valid C++ expression can be a function argument, including constants, mathematical and logical expressions, and other functions that return a value.

Using Functions as Parameters to Functions

Although it is legal for one function to take as a parameter a second function that returns a value, it can make for code that is hard to read and hard to debug.

As an example, say you have the functions `double()`, `triple()`, `square()`, and `cube()`, each of which returns a value. You could write

```
Answer = (double(triple(square(cube(myValue)))));
```

This statement takes a variable, `myValue`, and passes it as an argument to the function `cube()`, whose return value is passed as an argument to the function `square()`, whose return value is in turn passed to `triple()`, and that return value is passed to `double()`. The return value of this doubled, tripled, squared, and cubed number is now passed to `Answer`.

It is difficult to be certain what this code does (was the value tripled before or after it was squared?), and if the answer is wrong it will be hard to figure out which function failed.

An alternative is to assign each step to its own intermediate variable:

```
unsigned long myValue = 2;
unsigned long cubed  =   cube(myValue);         // cubed = 8
unsigned long squared = square(cubed);          // squared = 64
unsigned long tripled = triple(squared);        // tripled = 196
unsigned long Answer =  double(tripled);        // Answer = 392
```

Now each intermediate result can be examined, and the order of execution is explicit.

Parameters Are Local Variables

The arguments passed in to the function are local to the function. Changes made to the arguments do not affect the values in the calling function. This is known as passing by value, which means a local copy of each argument is made in the function. These local copies are treated just like any other local variables. Listing below illustrates this point.

A demonstration of passing by value.

Source Code:*ByValue.cpp*

```
 1  // Listing demonstrates parameter passing by value
 2  #include <iostream>
 3  using namespace std;
 4
 5  void swap(int x, int y);
 6
 7  int main()
 8  {
 9    int x = 5, y = 10;
10
11    cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
12    swap(x,y);
13    cout << "Main. After swap, x: " << x << " y: " << y << "\n";
14    return 0;
15  }
16
17  void swap (int x, int y)
18  {
19    int temp;
20
21    cout << "Swap. Before swap, x: " << x << " y: " << y << "\n";
22
23    temp = x;
24    x = y;
25    y = temp;
26
27    cout << "Swap. After swap, x: " << x << " y: " << y << "\n";
28
29  }
```

Output

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

Analysis: This program initializes two variables in `main()` and then passes them to the `swap()` function, which appears to swap them. When they are examined again in `main()`, however, they are unchanged!

The variables are initialized on line 9, and their values are displayed on line 11. `swap()` is called, and the variables are passed in.

Execution of the program switches to the `swap()` function, where on line 21 the values are printed again. They are in the same order as they were in `main()`, as expected. On lines 23 to 25 the values are swapped, and this action is confirmed by the printout on line 27. Indeed, while in the `swap()` function, the values are swapped.

Execution then returns to line 13, back in `main()`, where the values are no longer swapped.

As you've figured out, the values passed in to the `swap()` function are passed by value, meaning that copies of the values are made that are local to `swap()`. These local variables are swapped in lines 23 to 25, but the variables back in `main()` are unaffected.

Return Values

Functions return a value or return `void`. `Void` is a signal to the compiler that no value will be returned.

To return a value from a function, write the keyword `return` followed by the value you want to return. The value might itself be an expression that returns a value. For example:

```
return 5;
return (x > 5);
return (MyFunction());
```

These are all legal `return` statements, assuming that the function `MyFunction()` itself returns a value. The value in the second statement, `return (x > 5)`, will be zero if `x` is not greater than 5, or it will be `1`. What is returned is the value of the expression, `0` (`false`) or `1` (`true`), not the value of `x`.

When the `return` keyword is encountered, the expression following `return` is returned as the value of the function. Program execution returns immediately to the calling function, and any statements following the return are not executed.

It is legal to have more than one `return` statement in a single function. Listing below illustrates this idea.

A demonstration of multiple return statements.

Source Code:*MultipleReturen.cpp*

```
 1  // Listing demonstrates multiple return
 2  #include <iostream>
 3
 4  using namespace std;
 5
 6  int Doubler(int AmountToDouble);
 7
 8  int main()
 9  {
10
11    int result = 0;
```

```
12    int input;
13
14    cout << "Enter a number between 0 and 10,000 to double: ";
15    cin >> input;
16
17    cout << "\nBefore doubler is called... ";
18    cout << "\ninput: " << input << " doubled: " << result << "\n";
19
20    result = Doubler(input);
21
22    cout << "\nBack from Doubler...\n";
23    cout << "\ninput: " << input << "   doubled: " << result << "\n";
24
25
26    return 0;
27  }
28
29  int Doubler(int original)
30  {
31    if (original <= 10000)
32          return original * 2;
33    else
34          return -1;
35    cout << "You can't get here!\n";
36  }
```

Output

```
Enter a number between 0 and 10,000 to double: 9000

Before doubler is called...
input: 9000 doubled: 0

Back from doubler...

input: 9000   doubled: 18000

Enter a number between 0 and 10,000 to double: 11000

Before doubler is called...
input: 11000 doubled: 0

Back from doubler...
input: 11000  doubled: -1
```

Analysis: A number is requested on lines 14 and 15, and printed on line 18, along with the local variable result. The function `Doubler()` is called on line 20, and the input value is passed as a parameter. The result will be assigned to the local variable `result`, and the values will be reprinted on lines 22 and 23.

On line 31, in the function `Doubler()`, the parameter is tested to see whether it is greater than 10,000. If it is not, the function returns twice the original number. If it is greater than 10,000, the function returns `-1` as an error value.

The statement on line 35 is never reached, because whether or not the value is greater than 10,000, the function returns before it gets to line 35, on either line 32 or line 34. A good compiler will warn that this statement cannot be executed, and a good programmer will take it out!

Default Parameters

For every parameter you declare in a function prototype and definition, the calling function must pass in a value. The value passed in must be of the declared type. Thus, if you have a function declared as

```
long myFunction(int);
```

the function must in fact take an integer variable. If the function definition differs, or if you fail to pass in an integer, you will get a compiler error.

The one exception to this rule is if the function prototype declares a default value for the parameter. A default value is a value to use if none is supplied. The preceding declaration could be rewritten as

```
long myFunction (int x = 50);
```

This prototype says, "`myFunction()` returns a `long` and takes an integer parameter. If an argument is not supplied, use the default value of `50`." Because parameter names are not required in function prototypes, this declaration could have been written as

```
long myFunction (int = 50);
```

The function definition is not changed by declaring a default parameter. The function definition header for this function would be

```
long myFunction (int x)
```

If the calling function did not include a parameter, the compiler would fill `x` with the default value of `50`. The name of the default parameter in the prototype need not be the same as the name in the function header; the default value is assigned by position, not name.

Any or all of the function's parameters can be assigned default values. The one restriction is this: If any of the parameters does not have a default value, no previous parameter may have a default value.

If the function prototype looks like

```
long myFunction (int Param1, int Param2, int Param3);
```

you can assign a default value to `Param2` only if you have assigned a default value to `Param3`. You can assign a default value to `Param1` only if you've assigned default values to both `Param2` and `Param3`. Listing below demonstrates the use of default values.

A demonstration of default parameter values.

Source Code: *DefaultParameter.cpp*

```
 1  // Listing demonstrates use of default parameter values
 2  #include <iostream>
 3
 4  using namespace std;
 5
 6  int AreaCube(int length, int width = 25, int height = 1);
 7
 8  int main()
 9  {
10    int length = 100;
11    int width = 50;
12    int height = 2;
13    int area;
14
15    area = AreaCube(length, width, height);
16    cout << "First area equals: " << area << "\n";
17
18    area = AreaCube(length, width);
19    cout << "Second time area equals: " << area << "\n";
20
21    area = AreaCube(length);
22    cout << "Third time area equals: " << area << "\n";
23    return 0;
24  }
25
26  AreaCube(int length, int width, int height)
27  {
28    return (length * width * height);
29  }
```

Output

```
First area equals: 10000
Second time area equals: 5000
Third time area equals: 2500
```

Analysis: On line 6, the `AreaCube()` prototype specifies that the `AreaCube()` function takes three integer parameters. The last two have default values.

This function computes the area of the cube whose dimensions are passed in. If no `width` is passed in, a `width` of 25 is used and a `height` of 1 is used. If the `width` but not the `height` is passed in, a `height` of 1 is used. It is not possible to pass in the `height` without passing in a `width`.

On lines 10-12, the dimensions `length`, `height`, and `width` are initialized, and they are passed to the `AreaCube()` function on line 15. The values are computed, and the result is printed on line 16.

Execution returns to line 18, where `AreaCube()` is called again, but with no value for `height`. The default value is used, and again the dimensions are computed and printed.

Execution returns to line 21, and this time neither the `width` nor the `height` is passed in. Execution branches for a third time to line 27. The default values are used. The area is computed and then printed.

DO remember that function parameters act as local variables within the function. **DON'T** try to create a default value for a first parameter if there is no default value for the second. **DON'T** forget that arguments passed by value can not affect the variables in the calling function. **DON'T** forget that changes to a global variable in one function change that variable for all functions.

Overloading Functions

C++ enables you to create more than one function with the same name. This is called function overloading. The functions must differ in their parameter list, with a different type of parameter, a different number of parameters, or both. Here's an example:

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

`myFunction()` is overloaded with three different parameter lists. The first and second versions differ in the types of the parameters, and the third differs in the number of parameters.

The return types can be the same or different on overloaded functions. You should note that two functions with the same name and parameter list, but different return types, generate a compiler error.

New Term: Function *overloading* i s also called function *polymorphism*. Poly means many, and morph means form: a polymorphic function is many-formed.

Function polymorphism refers to the ability to "overload" a function with more than one meaning. By changing the number or type of the parameters, you can give two or more functions the same function name, and the right one will be called by matching the parameters used. This allows you to create a function that can average integers, doubles, and other values without having to create individual names for each function, such as `AverageInts()`, `AverageDoubles()`, and so on.

Suppose you write a function that doubles whatever input you give it. You would like to be able to pass in an `int`, a `long`, a `float`, or a `double`. Without function overloading, you would have to create four function names:

```
int DoubleInt(int);
long DoubleLong(long);
float DoubleFloat(float);
double DoubleDouble(double);
```

With function overloading, you make this declaration:

```
int Double(int);
long Double(long);
float Double(float);
double Double(double);
```

This is easier to read and easier to use. You don't have to worry about which one to call; you just pass in a variable, and the right function is called automatically. Listing below illustrates the use of function overloading.

A demonstration of function polymorphism.

Source Code:*FunctionOverloading.cpp*

```
 1  // Listing demonstrates function polymorphism
 2  #include <iostream>
 3
 4  using namespace std;
 5
 6  int Double(int);
 7  long Double(long);
 8  float Double(float);
 9  double Double(double);
10
11  int main()
12  {
13    int       myInt = 6500;
14    long      myLong = 65000;
15    float     myFloat = 6.5F;
16    double    myDouble = 6.5e20;
17
18    int       doubledInt;
19    long      doubledLong;
20    float     doubledFloat;
21    double    doubledDouble;
22
23    cout << "myInt: " << myInt << "\n";
24    cout << "myLong: " << myLong << "\n";
25    cout << "myFloat: " << myFloat << "\n";
26    cout << "myDouble: " << myDouble << "\n";
27
28    doubledInt = Double(myInt);
29    doubledLong = Double(myLong);
30    doubledFloat = Double(myFloat);
31    doubledDouble = Double(myDouble);
32
33    cout << "doubledInt: " << doubledInt << "\n";
34    cout << "doubledLong: " << doubledLong << "\n";
35    cout << "doubledFloat: " << doubledFloat << "\n";
36    cout << "doubledDouble: " << doubledDouble << "\n";
37
38    return 0;
39  }
40
41  int Double(int original)
42  {
```

```
43    cout << "In Double(int)\n";
44    return 2 * original;
45  }
46
47  long Double(long original)
48  {
49    cout << "In Double(long)\n";
50    return 2 * original;
51  }
52
53  float Double(float original)
54  {
55    cout << "In Double(float)\n";
56    return 2 * original;
57  }
58
59  double Double(double original)
60  {
61    cout << "In Double(double)\n";
62    return 2 * original;
63  }
```

Output

```
myInt: 6500
myLong: 65000
myFloat: 6.5
myDouble: 6.5e+20
In Double(int)
In Double(long)
In Double(float)
In Double(double)
DoubledInt: 13000
DoubledLong: 130000
DoubledFloat: 13
DoubledDouble: 1.3e+21
```

Analysis: The `Double()` function is overloaded with `int`, `long`, `float`, and `double`. The prototypes are on lines 6-9, and the definitions are on lines 41-63.

In the body of the main program, eight local variables are declared. On lines 13-16, four of the values are initialized, and on lines 28-31, the other four are assigned the results of passing the first four to the `Double()` function. Note that when `Double()` is called, the calling function does not distinguish which one to call; it just passes in an argument, and the correct one is invoked.

The compiler examines the arguments and chooses which of the four `Double()` functions to call. The output reveals that each of the four was called in turn, as you would expect.

Special Topics About Functions

Because functions are so central to programming, a few special topics arise which might be of interest when you confront unusual problems. Used wisely, inline functions can help you squeak out that last bit of performance. Function recursion is one of those wonderful, esoteric bits of programming which, every once in a while, can cut through a thorny problem otherwise not easily solved.

Inline Functions

When you define a function, normally the compiler creates just one set of instructions in memory. When you call the function, execution of the program jumps to those instructions, and when the function returns, execution jumps back to the next line in the calling function. If you call the function 10 times, your program jumps to the same set of instructions each time. This means there is only one copy of the function, not 10.

There is some performance overhead in jumping in and out of functions. It turns out that some functions are very small, just a line or two of code, and some efficiency can be gained if the program can avoid making these jumps just to execute one or two instructions. When programmers speak of efficiency, they usually mean speed: the program runs faster if the function call can be avoided.

If a function is declared with the keyword `inline`, the compiler does not create a real function: it copies the code from the inline function directly into the calling function. No jump is made; it is just as if you had written the statements of the function right into the calling function.

Note that inline functions can bring a heavy cost. If the function is called 10 times, the inline code is copied into the calling functions each of those 10 times. The tiny improvement in speed you might achieve is more than swamped by the increase in size of the executable program. Even the speed increase might be illusory. First, today's optimizing compilers do a terrific job on their own, and there is almost never a big gain from declaring a function `inline`. More important, the increased size brings its own performance cost.

What's the rule of thumb? If you have a small function, one or two statements, it is a candidate for `inline`. When in doubt, though, leave it out. Listing below demonstrates an `inline` function.

Demonstrates an inline function.

Source Code:*inline.cpp*

```
 1  // Listing demonstrates inline functions
 2  #include <iostream>
 3  using namespace std;
 4
 5  inline int Double(int);
 6
 7  int main()
 8  {
 9    int target;
10
11    cout << "Enter a number to work with: ";
12    cin >> target;
13    cout << "\n";
```

```
14
15    target = Double(target);
16    cout << "Target: " << target << endl;
17
18    target = Double(target);
19    cout << "Target: " << target << endl;
20
21
22    target = Double(target);
23    cout << "Target: " << target << endl;
24    return 0;
25  }
26
27  int Double(int target)
28  {
29    return 2*target;
30  }
```

```
Output
Enter a number to work with: 20

Target: 40
Target: 80
Target: 160
```

Analysis: On line 5, `Double()` is declared to be an inline function taking an `int` parameter and returning an `int`. The declaration is just like any other prototype except that the keyword `inline` is prepended just before the return value.

This compiles into code that is the same as if you had written the following:

```
target = 2 * target;
```

everywhere you entered

```
target = Double(target);
```

By the time your program executes, the instructions are already in place, compiled into the OBJ file. This saves a jump in the execution of the code, at the cost of a larger program.

NOTE: Inline is a hint to the compiler that you would like the function to be inlined. The compiler is free to ignore the hint and make a real function call.

Recursion

A function can call itself. This is called recursion, and recursion can be direct or indirect. It is direct when a function calls itself; it is indirect recursion when a function calls another function that then calls the first function.

Some problems are most easily solved by recursion, usually those in which you act on data and then act in the same way on the result. Both types of recursion, direct and indirect, come in two varieties: those that eventually end and produce an answer, and those that never end and produce a runtime failure. Programmers think that the latter is quite funny (when it happens to someone else).

It is important to note that when a function calls itself, a new copy of that function is run. The local variables in the second version are independent of the local variables in the first, and they cannot affect one another directly, any more than the local variables in `main()` can affect the local variables in any function it calls, as was illustrated in next listing.

To illustrate solving a problem using recursion, consider the Fibonacci series:

```
1,1,2,3,5,8,13,21,34...
```

Each number, after the second, is the sum of the two numbers before it. A Fibonacci problem might be to determine what the 12th number in the series is.

One way to solve this problem is to examine the series carefully. The first two numbers are 1. Each subsequent number is the sum of the previous two numbers. Thus, the seventh number is the sum of the sixth and fifth numbers. More generally, the nth number is the sum of n - 2 and n - 1, as long as n > 2.

Recursive functions need a stop condition. Something must happen to cause the program to stop recursing, or it will never end. In the Fibonacci series, n < 3 is a stop condition.

The algorithm to use is this:

1. Ask the user for a position in the series.

2. Call the `fib()` function with that position, passing in the value the user entered.

3. The `fib()` function examines the argument (`n`). If `n < 3` it returns 1; otherwise, `fib()` calls itself (recursively) passing in `n-2`, calls itself again passing in `n-1`, and returns the sum.

If you call `fib(1)`, it returns `1`. If you call `fib(2)`, it returns `1`. If you call `fib(3)`, it returns the sum of calling `fib(2)` and `fib(1)`. Because `fib(2)` returns `1` and `fib(1)` returns `1`, `fib(3)` will return `2`.

If you call `fib(4)`, it returns the sum of calling `fib(3)` and `fib(2)`. We've established that `fib(3)` returns `2` (by calling `fib(2)` and `fib(1)`) and that `fib(2)` returns `1`, so `fib(4)` will sum these numbers and return `3`, which is the fourth number in the series.

Taking this one more step, if you call `fib(5)`, it will return the sum of `fib(4)` and `fib(3)`. We've established that `fib(4)` returns `3` and `fib(3)` returns `2`, so the sum returned will be `5`.

This method is not the most efficient way to solve this problem (in `fib(20)` the `fib()` function is called 13,529 times!), but it does work. Be careful: if you feed in too large a number, you'll run out of memory. Every time `fib()` is called, memory is set aside. When it returns, memory is freed. With recursion, memory continues to be set aside before it is freed, and this system can eat memory very quickly. Listing below implements the `fib()` function.

WARNING: When you run Listing below, use a small number (less than 15). Because this uses recursion, it can consume a lot of memory.

Demonstrates recursion using the Fibonacci series.

Source Code:*Fibonacci.cpp*

```
 1  // Listing demonstrates recursion
 2  //      Fibonacci find.
 3  //      Finds the nth Fibonacci number
 4  //      Uses this algorithm: Fib(n) = fib(n-1) + fib(n-2)
 5  //      Stop conditions: n = 2 || n = 1
 6  #include <iostream>
 7  using namespace std;
 8
 9  int fib(int n);
10
11  int main()
12  {
13
14    int n, answer;
15    cout << "Enter number to find: ";
16    cin >> n;
17
18    cout << "\n\n";
19
20    answer = fib(n);
21
22    cout << answer << " is the " << n << "th Fibonacci number\n";
23    return 0;
24  }
25
26  int fib (int n)
27  {
28    cout << "Processing fib(" << n << ")... ";
29
30    if (n < 3 )
31    {
32          cout << "Return 1!\n";
33          return (1);
34    }
35    else
36    {
37          cout << "Call fib(" << n-2 << ") and fib(" << n-1 << ").\n";
38          return( fib(n-2) + fib(n-1));
39    }
40  }
```

```
Output
Enter number to find:  5

Processing fib(5)... Call fib(3) and fib(4).
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(4)... Call fib(2) and fib(3).
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
5 is the 5th Fibonacci number
```

Analysis: The program asks for a number to find on line 15 and assigns that number to `target`. It then calls `fib()` with the `target`. Execution branches to the `fib()` function, where, on line 28, it prints its argument.
The argument `n` is tested to see whether it equals `1` or `2` on line 30; if so, `fib()` returns. Otherwise, it returns the sums of the values returned by calling `fib()` on `n-2` and `n-1`.

In the example, `n` is 5 so `fib(5)` is called from `main()`. Execution jumps to the `fib()` function, and `n` is tested for a value less than 3 on line 30. The test fails, so `fib(5)` returns the sum of the values returned by `fib(3)` and `fib(4)`. That is, `fib()` is called on `n-2` (5 - 2 = 3) and `n-1` (5 - 1 = 4). `fib(4)` will return `3` and `fib(3)` will return `2`, so the final answer will be 5.

Because `fib(4)` passes in an argument that is not less than 3, `fib()` will be called again, this time with 3 and 2. `fib(3)` will in turn call `fib(2)` and `fib(1)`. Finally, the calls to `fib(2)` and `fib(1)` will both return `1`, because these are the stop conditions.

The output traces these calls and the return values. Compile, link, and run this program, entering first 1, then 2, then 3, building up to 6, and watch the output carefully. Then, just for fun, try the number 20. If you don't run out of memory, it makes quite a show!

Recursion is not used often in C++ programming, but it can be a powerful and elegant tool for certain needs.

NOTE: Recursion is a very tricky part of advanced programming. It is presented here because it can be very useful to understand the fundamentals of how it works, but don't worry too much if you don't fully understand all the details.

Overview of Templates

A template declaration can be a function declaration, function definition, class declaration, or class definition. The template declaration takes one or more parameters, which can be values, types, or class templates.

In most cases, you can use a template simply by naming the template and providing arguments for the template parameters: constant expressions, types, or template references. This is known as instantiating the template. You can instantiate a template at its point of use, or declare a separate instantiation as a class or function declaration. An instance of a function template creates a function; an instance of a class template creates a class and all of its members.

A template lets you define a class or function once for a wide range of template arguments, but sometimes you need to customize a template for particular arguments. This is known as specializing the template. A template specialization, as its name implies, is a special case of the template pattern. When you instantiate a template, the compiler uses the template arguments to pick a specialization, or if no specialization matches the arguments, the original template declaration. A specialization can be total or partial. A total specialization specifies values for all of the template arguments, and a partial specialization specifies only some of the template arguments.

The terminology used in this book reflects the terminology that many C++ programmers have adopted, even though that terminology differs slightly from that used in the C++ standard. In the standard, "specialization" means an instance of a template. "Instantiation" also refers to an instance of a template. When you declare a special case of a template, that is known as explicit specialization.

Many C++ programmers prefer to keep specialization and instantiation as separate concepts and separate terms.

Note that a template declaration defines only the pattern. A specialization defines a pattern that applies to a specific set of template arguments. Only by instantiating a template do you declare or define a function or class. When you instantiate a template, the compiler uses the template arguments to pick which pattern to instantiate: a specialization or the main template.

Writing a template is more difficult than writing a non-template class or function. The template can be instantiated in almost any context, and the context can affect how the template definition is interpreted. Name lookup rules are more complicated for templates than for non-templates.

Following example shows several different kinds of templates and their uses.

Declaring and using templates

Source Code:*Templates.cpp*

```
1  #include <cmath>
2  #include <complex>
3  #include <iostream>
4  #include <ostream>
5
6  // Template declaration of point
7
```

```
 8 template<typename T>
 9 class point {
10 public:
11   typedef T value_type;
12   point(const T& x, const T& y) : x_(x), y_(y) {}
13   point() : x_(T(  )), y_(T(  )) {}
14   T x(  )             const { return x_; }
15   T& x(  )                  { return x_; }
16   void x(const T& new_x)    { x_ = new_x; }
17   T y(  )             const { return y_; }
18   T& y(  )                  { return y_; }
19   void y(const T& new_y)    { y_ = new_y; }
20 private:
21   T x_, y_;
22 };
23
24 // Instantiate point<>.
25 typedef point<std::complex<double> > strange;
26 strange s(std::complex<double>(1, 2), std::complex<double>(3, 4));
27
28 // Specialize point<int> to use call-by-value instead of const
29 references.
30 template<>
31 class point<int> {
32 public:
33   typedef int value_type;
34   point(int x, int y) : x_(x), y_(y) {}
35   point(  ) : x_(0), y_(0) {}
36   int x(  )      const { return x_; }
37   int& x(  )            { return x_; }
38   void x(int new_x)     { x_ = new_x; }
39   int y(  )      const { return y_; }
40   int& y(  )            { return y_; }
41   void y(int new_y)     { y_ = new_y; }
42 private:
43   int x_, y_;
44 };
45
46 // Instance of the specialized point<int>
47 point<int> p(42, 0);
48
49 // Instance of the general point<>, using long as the template
50 argument
51 point<long> p(42, 0);
52
53 // Function template
54 template<typename T>
55 T abs(T x){
56   return x < 0 ? -x : x;
57 }
58
59 namespace {
60   // Explicit instantiation
61   const int abs_char_min1 = abs<int>(CHAR_MIN);
62
63   // Implicit instantiation
64   const int abs_char_min2 = abs(CHAR_MIN);
65 }
```

```
66
67  // Overload abs(  ) with another function template.
68  template<typename floatT, typename T>
69  floatT abs(const point<T>& p){
70    return std::sqrt(static_cast<floatT>(p.x()*p.x()+p.y()*p.y()));
71  }
72
73  int main(  ){
74    point<double> p;
75
76    // Call instance of function template. Compiler deduces second
77  template
78    // argument (double) from the type of p.
79    double x = abs<long double>(p);
80    std::cout << x << '\n'; // prints 0
81    std::cout << abs_char_min1 << '\n';
82  }
```

Template Declarations

A template declaration begins with a template header followed by a function declaration or definition, or a class declaration or definition. Template declarations can appear only at namespace or class scope. The template name must be unique in its scope (except for overloaded functions).

The template header starts with the template keyword followed by the template parameters enclosed in angle brackets (<>). Multiple parameters are separated by commas. The syntax is:

```
template < parameter-list > declaration
```

There are three kinds of template parameters: values, types, and class templates. Similar to a function parameter, a template parameter has an optional name and an optional default argument.

Function templates cannot have default template arguments. If a class template has member definitions that are outside the class definition, only the class template takes default arguments; the individual member definitions do not. If a default argument is present, it is preceded by an equal sign. Only the rightmost parameters can have default arguments. If any parameter has a default argument, all parameters to its right must also have default arguments. Previous example shows valid and invalid member definitions.

Defining members of a class template

// OK: default argument for template parameter A
```
template<typename T, typename A = std::allocator<T> >

class hashset {
  bool empty(  ) const;
  size_t size(  ) const;
  ...
};
```

// Error: do not use default argument here
```
template<typename T, typename A = std::allocator<T> >
bool hashset<T,A>::empty(  ) { return size(  ) == 0; }
```

// OK
```
template<typename T, typename A>
size_t hashset<T,A>::size(  ) { ... }
```

Each template header defines the template name and template parameters. The scope of a parameter name extends from its declaration to the end of the declaration or definition of the class or function. A parameter name can be used in subsequent template parameters in the same template. The template parameter name must be unique in the template declaration and cannot be redeclared in its scope. If a class template has separate definitions for its members, each member definition is free to use different names for the template parameters. (See Section 7.4.1 later in this chapter for more information.)

There are three kinds of template parameters:

1. Value template parameter
2. Declared in the same manner as a function parameter:
3. type-specifiers declarator
4. type-specifiers declarator = expr

The type must be an integral, enumeration, pointer, reference, or pointer-to-member type. When the template is instantiated, the argument must be a constant integral or enumeration expression, the address of a named object or function with external linkage, or the address of a member:

```
template<unsigned Size>
struct name {
// ...
  unsigned size(  ) const { return Size; }
private:
  char name_[Size+1];
};
name<100> n;
```

Note that a string literal is an unnamed object with internal linkage, so you cannot use it as a template argument:

```
template<const char* Str> void print(const char* s = Str);
print<"default">(  ); // Error
const char def[] = "default";
print<def>(  );       // OK
```

The type-specifiers can be elaborated type specifiers that start with typename, that is, typename followed by a qualified type name. (If typename is followed by a plain identifier, the template parameter is a type parameter, as described later.) For more information about this use of typename.

```
template<typename list<int>::value_type value>
int silly(  ) { return value; }
```

Type template parameter

Introduced with the keyword typename followed by the optional parameter name:

```
typename identifier
typename identifier = type
```

The class keyword can be used in place of typename and has the same meaning in this context. (A useful convention is to use class when the argument must be a class and typename when the argument can be any type.) When the template is instantiated, the argument must be a type (that is, a list of type specifiers with optional pointer, reference, array, and function operators). In the following example, the point template is instantiated with unsigned long int as the template argument:

```
template<typename T> struct point {
  T x, y;
};

point<unsigned long int> pt;
```

If typename is followed by a qualified type name instead of a plain identifier, it declares a value parameter of that type, as described earlier.

Template template parameter

Must be a class template. It has the form of a template declaration:

```
template < parameter-list > class identifier
template < parameter-list > class identifier = template-id
```

When the template is instantiated, the argument must be a class template:

```
// Erase all occurrences of item from a sequence container.
template<template<typename T, typename A> class C, typename T, typename A>
void erase(C<T,A>& c, const T& item){
  c.erase(std::remove(c.begin(), c.end(  ), item), c.end(  ));
}

...

list<int> l;

...

erase(l, 42);
```

To use a template declaration, you must create an instance of the template, either explicitly (by naming the template and enclosing a list of template arguments in angle brackets) or implicitly (by letting the compiler deduce the template arguments from context) for a function template. In either case, the compiler must know about a template declaration before the template is used. Typically, a template is declared in an #include file or header. The header declares the function or class template and possibly provides the definition of the function template or the definitions of all the members of the class template.

A template instance must provide an argument for each template parameter. If a class template has fewer arguments than parameters, the remaining parameters must have default arguments, which are used for the template instance.

If a function template has fewer arguments than parameters, the remaining arguments are deduced from the context of the function call (as explained in the next section). If the arguments cannot be deduced, the compiler reports an error.

Function Templates

A function template defines a pattern for any number of functions whose definitions depend on the template parameters. You can overload a function template with a non-template function or with other function templates. You can even have a function template and a non-template function with the same name and parameters.

Function templates are used throughout the standard library. The best-known function templates are the standard algorithms (such as copy, sort, and for_each).

Declare a function template using a template declaration header followed by a function declaration or definition. Default template arguments are not allowed in a function template declaration or definition. In the following example, round is a template declaration for a function that rounds a number of type T to N decimal places (see following example for the definition of round), and min is a template definition for a function that returns the minimum of two objects:

```
// Round off a floating-point value of type T to N digits.
template<unsigned N, typename T>  T round(T x);

// Return the minimum of a and b.
template<typename T> T min(T a, T b)
{ return a < b ? a : b; }
```

To use a function template, you must specify a template instance by providing arguments for each template parameter. You can do this explicitly, by listing arguments inside angle brackets:

```
long x = min<long>(10, 20);
```

However, it is more common to let the compiler deduce the template argument types from the types of the function arguments:

```
int x = min(10, 20); // Calls min<int>(  )
```

Function Signatures

The signature of a function template includes the template parameters even if those parameters are never used in the function's return type or parameter types. Thus, instances of different template specializations produce distinct functions. The following example has two function templates, both named func. Because their template parameters differ, the two lines declare two separate function templates:

```
template<typename T> void func(T x = T(  ));
template<typename T, typename U> void func(T x = U(  ));
```

Typically, such function templates are instantiated in separate files; otherwise, you must specify the template arguments explicitly to avoid overload conflicts.

The function signature also includes expressions that use any of the template parameters. In the following example, two function templates overload func. The function parameter type is an instance of the demo class template. The template parameter for demo is an expression that depends on func's template parameter (T or i):

```
template<int x> struct demo {};
template<typename T> void func(demo<sizeof(T)>);
template<int i> void func(demo<i+42>);
```

Just as you can have multiple declarations (but not definitions) of a single function, you can declare the same function template multiple times using expressions that differ only by the template parameter names. Even if the declarations are in separate files, the compiler and linker ensure that the program ends up with a single copy of each template instance.

If the expressions differ by more than just the parameter names, they result in distinct functions, so you can have distinct function template definitions. If the argument expressions result in the same value, the program is invalid, although the compiler is not required to issue an error message:

```
template<int i> void func(demo<i+42>);
template<int x> void func(demo<x+42>) {}    // OK
template<int i> void func(demo<i+41>) {}    // OK
template<int i> void func(demo<i+40+2>) {} // Error
```

Follow the standard procedure of declaring templates in header files and using those headers in the source files where they are needed, thereby ensuring that every source file sees the same template declarations. Then you will not have to be concerned about different source files seeing declarations that differ only by the form of the template argument expressions. The compiler tries to ensure that the program ends up with a single copy of functions that are meant to be the same, but distinct copies of those that are meant to be distinct.

Deducing Argument Types

Most uses of function templates do not explicitly specify all of the template arguments, letting the compiler deduce the template arguments from the function call. You can provide explicit arguments for the leftmost template parameters, leaving the rightmost parameters for the compiler to deduce. If you omit all the template arguments, you can also choose to omit the angle brackets.

In the following discussion, make sure you distinguish between the parameters and arguments of a template and those of a function. Argument type deduction is the process of determining the template arguments, given a function call or other use of a function (such as taking the address of a function).

The basic principle is that a function call has a list of function arguments, in which each function argument has a type. The function argument types must be matched to function parameter types; in the

case of a function template, the function parameter types can depend on the template parameters. The ultimate goal, therefore, is to determine the appropriate template arguments that let the compiler match the function parameter types with the given function argument types.

Usually, the template parameters are type or template template parameters, but in a few cases, value parameters can also be used (as the argument for another template or as the size of an array).

The function parameter type can depend on the template parameter in the following ways, in which the template parameter is v for a value parameter, T for a type parameter, or TT for a template template parameter:

- T, T*, T&, const T, volatile T, const volatile T
- Array of T
- Array of size v
- Template TT, or any class template whose template argument is T or v
- Function pointer in which T is the return type or the type of any function parameter
- Pointer to a data member of T, or to a data member whose type is T
- Pointer to a member function that returns T, to a member function of T, or to a member function in which T is the type of any function parameter
- Types can be composed from any of these forms.

Thus, type deduction can occur for a struct in which the types of its members are also deduced, or a function's return type and parameter types can be deduced.

A single function argument can result in the deduction of multiple template parameters, or multiple function arguments might depend on a single template parameter.

If the compiler cannot deduce all the template arguments of a function template, you must provide explicit arguments. Following example shows several different ways the compiler can deduce template arguments.

Source Code: *DeducingTemplateArguments.cpp*

```
 1 #include <algorithm>
 2 #include <cstddef>
 3 #include <iostream>
 4 #include <list>
 5 #include <ostream>
 6 #include <set>
 7
 8 // Simple function template. Easy to deduce T.
 9 template<typename T>
10 T min(T a, T b){  return a < b ? a : b;}
11
12 template<typename T, std::size_t Size>
13 struct array{
14   T value[Size];
15 };
16
17 // Deduce the row size of a 2D array.
18 template<typename T, std::size_t Size>
19 std::size_t dim2(T [][Size]){
```

```
20    return Size;
21  }
22
23  // Overload the function size(  ) to return the number of elements
24  // in an array<> or in a standard container.
25  template<typename T, std::size_t Size>
26  std::size_t size(const array<T,Size>&){
27    return Size;
28  }
29
30  template<template<typename T, typename A> class container,
31           typename T, typename A>
32
33  typename container<T,A>::size_type size(container<T,A>& c){
34    return c.size(  );
35  }
36
37  template<template<typename T, typename C, typename A>
38             class container,
39           typename T, typename C, typename A>
40
41  typename container<T,C,A>::size_type
42  size(container<T,C,A>& c){
43    return c.size(  );
44  }
45
46  // More complicated function template. Easy to deduce Src,
47  // but impossible to deduce Dst.
48  template<typename Dst, typename Src>
49  Dst cast(Src s){
50    return s;
51  }
52
53  int main(  ){
54    min(10, 100);       // min<int>
55    min(10, 1000L);     // Error: T cannot be int and long
56    array<int,100> data1;
57
58    // Deduce T=int and SIZE=100 from type of data1.
59    std::cout << size(data1) << '\n';
60
61    int y[10][20];
62    std::cout << dim2(y) << '\n';  // Prints 20
63    std::list<int> lst;
64    lst.push_back(10);
65    lst.push_back(20);
66    lst.push_back(10);
67    std::cout << size(lst) << '\n';  // Prints 3
68    std::set<char> s;
69    const char text[] = "hello";
70    std::copy(text, text+5, std::inserter(s, s.begin(  )));
71    std::cout << size(s) << '\n';  // Prints 4
72    // Can deduce Src=int, but not return type, so explicitly set
73    // Dst=char
74
75    char c = cast<char>(42);
76  }
```

Overloading

Function templates can be overloaded with other function templates and with non-template functions. Templates introduce some complexities, however, beyond those of ordinary overloading.

As with any function call, the compiler collects a list of functions that are candidates for overload resolution. If the function name matches that of a function template, the compiler tries to determine a set of template arguments, deduced or explicit. It the compiler cannot find any suitable template arguments, the template is not considered for overload resolution. If the compiler can find a set of template arguments, they are used to instantiate the function template, and the template instance is added to the list of candidate functions. Thus, each template contributes at most one function to the list of overload candidates. The best match is then chosen from the list of template and non-template functions according to the normal rules of overload resolution: non-template functions are preferred over template functions, and more specialized template functions are preferred over less specialized template functions.

Following example shows how templates affect overload resolution. Sometimes, template instantiation issues can be subtle. The last call to print results in an error, not because the compiler cannot instantiate print, but because the instantiation contains an error, namely, that operator << is not defined for std::vector<int>.

Source Code: *OverloadingFunctionTemplates.cpp*

```
 1  #include <cctype>
 2  #include <iomanip>
 3  #include <iostream>
 4  #include <ostream>
 5  #include <string>
 6  #include <vector>
 7
 8  template<typename T>
 9  void print(const T& x){
10    std::cout << x;
11  }
12
13  void print(char c){
14    if (std::isprint(c))
15      std::cout << c;
16    else
17      std::cout << std::hex << std::setfill('0')
18                << "'\\x" << std::setw(2) << int(c) << '\''
19                << std::dec;
20  }
21
22  template<>
23  void print(const long& x){
24    std::cout << x << 'L';
25  }
26
27  int main(  ){
28    print(42L);     // Calls print<long> specialization
29    print('x');     // Calls print(char) overload
30    print('\0');     // Calls print(char) overload
31    print(42);      // Calls print<int>
```

```
32    print(std::string("y"));      // Calls print<string>
33    print(  );                    // Error: argument required
34    print(std::vector<int>(  )); // Error
35  }
```

Operators

You can create operator templates in the same manner as function templates. The usual rules for deducing argument types apply as described earlier in this section. If you want to specify the arguments explicitly, you can use the operator keyword:

```
template<typename T>
bigint operator+(const bigint& x, const T& y);
bigint a;
a = a + 42;                     // Argument type deduction
a = operator+<long>(a, 42);     // Explicit argument type
```

When using operator<, be sure to leave a space between the operator symbol and the angle brackets that surround the template argument, or else the compiler will think you are using the left-shift operator:

```
operator<<long>(a, 42);  // Error: looks like left-shift
operator< <long>(a, 42); // OK
```

Type conversion operators can use ordinary type cast syntax, but if you insist on using an explicit operator keyword, you cannot use the usual syntax for specifying the template argument. Instead, the target type specifies the template argument type:

```
struct demo {
  template<typename T>
  operator T*(  ) const { return 0; }
};

demo d;
char* p = d.operator int*(  );                // Error: illegal cast
char* c = d; // d.operator char*(  );         // OK
```

Class Templates

A class template defines a pattern for any number of classes whose definitions depend on the template parameters. The compiler treats every member function of the class as a function template with the same parameters as the class template.

Class templates are used throughout the standard library for containers (list<>, map<>, etc.), complex numbers (complex<>), and even strings (basic_string<>) and I/O (basic_istream<>, etc.).

The basic form of a class template is a template declaration header followed by a class declaration or definition:

```
template<typename T>
struct point {
  T x, y;
};
```

To use the class template, supply an argument for each template parameter (or let the compiler substitute a default argument). Use the template name and arguments the way you would a class name.

```
point<int> pt = { 42, 10 };
typedef point<double> dpoint;
dpoint dp = { 3.14159, 2.71828 };
```

In member definitions that are separate from the class definition, you must declare the template using the same template parameter types in the same order, but without any default arguments. (You can change the names of the template parameters, but be sure to avoid name collisions with base classes and their members. See Section 7.8 later in this chapter for more information.) Declare the member definitions the way you would any other definition, except that the class name is a template name (with arguments), and the definition is preceded by a template declaration header.

In the class scope (in the class definition, or in the definition of a class member), the bare class name is shorthand for the full template name with arguments. Thus, you can use the bare identifier as the constructor name, after the tilde in a destructor name, in parameter lists, and so on. Outside the class scope, you must supply template arguments when using the template name. You can also use the template name with different arguments to specify a different template instance when declaring a member template. The following example shows how the template name can be used in different ways:

```
template<typename T> struct point {
  point(T x, T y);    // Or point<T>(T x, T y);
  ~point<T>( );       // Or ~point( );
  template<typename U>
  point(const point<U>& that);
  ...
};
```

Following example shows a class template with several different ways to define its member function templates.

Source Code: *DefiningClassTemplate.cpp*

```
 1  template<typename T, typename U = int>
 2  class demo {
 3  public:
 4    demo(T t, U u);
 5    ~demo( );
 6    static T data;
 7    class inner; // inner is a plain class, not a template.
 8  };
 9
10  template<typename T, typename U>
11  demo<T,U>::demo(T, U)      // Or demo<T,U>::demo<T,U>(T, U)
12  {}
13
14  template<typename T, typename U>
15  demo<T,U>::~demo<T,U>( ) // Or demo<T,U>::~demo( )
```

```
16 {}
17
18 template<typename U, typename T> // Allowed, but confusing
19 U demo<U,T>::data;
20
21 template<typename T, typename U>
22 class demo<T,U>::inner {
23 public:
24   inner(T, demo<T,U>& outer);
25 private:
26   demo<T,U>& outer;
27 };
28
29 // The class name is "demo<T,U>::inner", and the constructor name
30 // is "inner".
31 template<typename T, typename U>
32 demo<T,U>::inner::inner(T, demo<T,U>& outer): outer(outer)
33 {}
```

The template defines only the pattern; the template must be instantiated to declare or define a class. See the sections Section 7.5 and Section 7.7 later in this chapter for more information.

Member Templates

A member template is a template inside a class template or non-template class that has its own template declaration header, with its own template parameters. It can be a member function or a nested type. Member templates have the following restrictions:

- Local classes cannot have member templates.
- A member function template cannot be a destructor.
- A member function template cannot be virtual.

If a base class has a virtual function, a member function template of the same name and parameters in a derived class does not override that function.

Type conversion functions have additional rules because they cannot be instantiated using the normal template instantiation or specialization syntax:

- A using declaration cannot refer to a type conversion template in a base class.
- Use ordinary type conversion syntax to instantiate a type conversion template:

```
template<typename T>
struct silly {
  template<typename U> operator U*(  ) { return 0; }
};

...

silly<int> s, t(42);

if (static_cast<void*>(s))
```

```
        std::cout << "not null\n";
```

A class can have a non-template member function with the same name as a member function template. The usual overloading rules apply (Chapter 5), which means the compiler usually prefers the non-template function to the function template. If you want to make sure the function template is called, you can explicitly instantiate the template:

```
template<typename T>
struct demo {
  template<typename U> void func(U);
  void func(int);
};

demo<int> d;
d.func(42);      // Calls func(int)
d.func<int>(42); // Calls func<int>
```

Friends

A friend (of a class template or an ordinary class) can be a template, a specialization of a template, or a non-template function or class. A friend declaration cannot be a partial specialization. If the friend is a template, all instances of the template are friends. If the class granting friendship is a template, all instances of the template grant friendship. A specialization of a template can be a friend, in which case instances of only that specialization are granted friendship:

```
template<typename T> class buddy {};
template<typename T> class special {};
class demo {
  // All instances of buddy<> are friends.
  template<typename T> friend class buddy;

  // special<int> is a friend, but not special<char>, etc.
  friend class special<int>;
};
```

When you use the containing class template as a function parameter, you probably want friend functions to be templates also:

```
template<typename T>
class outer {
  friend void func1(outer& o);    // Wrong

  template<typename U>
  friend void func2(outer<U>& o); // Right
};
```

A friend function can be defined in a class template, in which case every instance of the class template defines the function. The function definition is compiled even if it is not used.

You cannot declare a friend template in a local class.

If a friend declaration is a specialization of a function template, you cannot specify any default function arguments, and you cannot use the inline specifier. These items can be used in the original function template declaration, however.

All the rules for function templates apply to friend function templates. Following example shows examples of friend declarations and templates.

Source Code: *FriendTemplatesAndFriendsOfTemplates.cpp*

```
1  #include <algorithm>
2  #include <cstddef>
3  #include <iostream>
4  #include <iterator>
5  #include <ostream>
6
```

```
 7  template<typename T, std::size_t Size>
 8  class array {
 9  public:
10    typedef T value_type;
11    class iterator_base :
12      std::iterator<std::random_access_iterator_tag, T> {
13    public:
14      typedef T value_type;
15      typedef std::size_t size_type;
16      typedef std::ptrdiff_t distance_type;
17
18      friend inline bool operator==(const iterator_base& x,
19                                    const iterator_base& y){
20        return x.ptr_ == y.ptr_;
21      }
22
23      friend inline bool operator!=(const iterator_base& x,
24                                    const iterator_base& y){
25        return ! (x == y);
26      }
27
28      friend inline ptrdiff_t operator-(const iterator_base& x,
29                                        const iterator_base& y){
30        return x.ptr_ - y.ptr_;
31      }
32    protected:
33      iterator_base(const iterator_base& that): ptr_(that.ptr_) {}
34      iterator_base(T* ptr) : ptr_(ptr) {}
35      iterator_base(const array& a, std::size_t i)
36        : ptr_(a.data_ + i) {}
37      T* ptr_;
38    };
39
40    friend class iterator_base;
41
42    class iterator : public iterator_base {
43    public:
44      iterator(const iterator& that) : iterator_base(that) {}
45      T& operator*(  )          const { return *this->ptr_; }
46      iterator& operator++(  ) { ++this->ptr_; return *this; }
47      T* operator->(  )         const { return this->ptr_; }
48    private:
49      friend class array;
50      iterator(const array& a, std::size_t i = 0):iterator_base(a, i){}
51      iterator(T* ptr) : iterator_base(ptr) {}
52      friend inline iterator operator+(iterator iter, int off){
53        return iterator(iter.ptr_ + off);
54      }
55    };
56    array(  ) : data_(new T[Size]) {}
57    array(const array& that);
58    ~array(  )                      { delete[] data_; }
59    iterator begin(  )              { return iterator(*this); }
60    const_iterator begin(  ) const { return iterator(*this); }
61    iterator end(  );
62    const_iterator end(  )   const;
63
64    T& operator[](std::size_t i);
```

```
65    T operator[](std::size_t i);
66    template<typename U, std::size_t USize>
67    friend void swap(array<U,USize>& a, array<U,USize>& b);
68  private:
69    T* data_;
70  };
71
72  template<typename T, std::size_t Size>
73
74  void swap(array<T,Size>& a, array<T,Size>& b){
75    T* tmp = a.data_;
76    a.data_ = b.data_;
77    b.data_ = tmp;
78  }
```

Namespaces

A namespace is a named scope. By grouping related declarations in a namespace, you can avoid name collisions with declarations in other namespaces. For example, suppose you are writing a word processor, and you use packages that others have written, including a screen layout package, an equation typesetting package, and an exact-arithmetic package for computing printed positions to high accuracy with fixed-point numbers.

The equation package has a class called fraction, which represents built-up fractions in an equation; the arithmetic package has a class called fraction, for computing with exact rational numbers; and the layout package has a class called fraction for laying out fractional regions of a page. Without namespaces, all three names would collide, and you would not be able to use more than one of the three packages in a single source file.

With namespaces, each class can reside in a separate namespacefor example,

```
layout::fraction, eqn::fraction, and math::fraction.
```

C++ namespaces are similar to Java packages, with a key difference: in Java, classes in the same package have additional access rights to each other; in C++, namespaces confer no special access privileges.

Namespace Definitions

Define a namespace with the namespace keyword followed by an optional identifier (the namespace name) and zero or more declarations in curly braces. Namespace declarations can be discontiguous, even in separate source files or headers. The namespace scope is the accumulation of all definitions of the same namespace that the compiler has seen at the time it looks up a given name in the namespace. Namespaces can be nested. Following example shows a sample namespace definition.

Source Code: *Namespace.cpp*

```
 :  // The initial declaration
 :
12  namespace numeric {
13    class rational { ... }
14    template<typename charT, typename traits>
15    basic_ostream<charT,traits>& operator<<(
16      basic_ostream<charT,traits>& out, const rational& r);
17  }
18
 :   . . .
 :
27  //This is a second definition. It adds an operator to the namespace.
28  namespace numeric {
29    rational operator+(const rational&, const rational&);
30  }
31
32  //The definition of operator+ can appear inside or outside the
33  // namespace definition. If it is outside, the name must be
```

```
34 //qualified with the scope operator.
35 numeric::rational numeric::operator+(const rational& r1,
36                                       const rational& r2){
37    . . .
38 }
39
40 int main(  ){
41   using numeric::rational;
42   rational a, b;
43   std::cout << a + b << '\n';
44 }
```

You can define a namespace without a name, in which case the compiler uses a unique, internal name. Thus, each source file's unnamed namespace is separate from the unnamed namespace in every other source file.

You can define an unnamed namespace nested within a named namespace (and vice versa). The compiler generates a unique, private name for the unnamed namespace in each unique scope. As with a named namespace, you can use multiple namespace definitions to compose the unnamed namespace, as shown in the following example.

Source Code: *UnnamedNamespace.cpp*

```
 1 #include <iostream>
 2 #include <ostream>
 3
 4 namespace {
 5   int i = 10;
 6 }
 7
 8 namespace {
 9   int j;          // Same unnamed namespace
10   namespace X {
11     int i = 20;   // Hides i in outer, unnamed namespace
12   }
13   namespace Y = X;
14   int f(  ) { return i; }
15 }
16
17 namespace X {
18   int i = 30;
19   // X::unnamed is different namespace than ::unnamed.
20   namespace {
21     int i=40;   //Hides ::X::i, but is inaccessible outside the
22                 //unnamed namespace
23     int f(  ) { return i; }
24   }
25 }
26 int main(  ){
27   int i = X::i;  // ambiguous: unnamed::X or ::X?
28   std::cout << ::X::f(  ) << '\n'; // Prints 40
29   std::cout << Y::i << '\n';       // Prints 20
30   std::cout << f(  ) << '\n';      // Prints 10
31 }
32
```

The advantage of using an unnamed namespace is that you are guaranteed that all names declared in it can never clash with names in other source files. The disadvantage is that you cannot use the scope operator (::) to qualify identifiers in an unnamed namespace, so you must avoid name collisions within the same source file.

C programmers are accustomed to using global static declarations for names that are private to a source file. You can do the same in C++, but it is better to use an unnamed namespace because a namespace can contain any kind of declaration (including classes, enumerations, and templates), whereas static declarations are limited to functions and objects.

Declarations of static objects and functions at namespace scope are deprecated in C++.

Declarations outside of all namespaces, functions, and classes are implicitly declared in the global namespace. A program has a single global namespace, which is shared by all source files that are compiled and linked into the program. Declarations in the global namespace are typically referred to as global declarations. Global names can be accessed directly using the global scope operator (the unary ::).

Namespace Aliases

A namespace alias is a synonym for an existing namespace. You can use an alias name to qualify names (with the :: operator) in using declarations and directives, but not in namespace definitions. Following example shows some alias examples.

Source Code: *NamespaceAlias.cpp*

```
 1  namespace original {
 2    int f( );
 3  }
 4
 5  namespace = original;     // Alias
 6
 7  int ns::f( ) { return 42; }   // OK
 8
 9  using ns::f;               // OK
10
11  int g( ) { return f( ); }
12
13  namespace ns { // Error: cannot use alias here
14    int h( );
15  }
```

A namespace alias can provide an abbreviation for an otherwise unwieldy namespace name. The long name might incorporate a full organization name, deeply nested namespaces, or version numbers:

```
namespace tempest_software_inc {
  namespace version_1 { ... }
  namespace version_2 { ... }
}
namespace tempest_1 = tempest_software_inc::version_1;
namespace tempest_2 = tempest_software_inc::version_2;
```

using Declarations

A using declaration imports a name from one namespace and adds it to the namespace that contains the using declaration. The imported name is a synonym for the original name. Only the declared name is added to the target namespace, which means using an enumerated type does not bring with it all the enumerated literals. If you want to use all the literals, each one requires its own using declaration.

Because a name that you reference in a using declaration is added to the current namespace, it might hide names in outer scopes. A using declaration can also interfere with local declarations of the same name.

Following example shows some examples of using declarations.

Source Code: *DefiningANamespace.cpp*

```
 1  namespace numeric {
 2    class fraction {  . . .  };
 3    fraction operator+(int, const fraction&);
 4    fraction operator+(const fraction&, int);
 5    fraction operator+(const fraction&, const fraction&);
 6  }
 7
 8  namespace eqn {
 9    class fraction {  . . .  };
10    fraction operator+(int, const fraction&);
11    fraction operator+(const fraction&, int);
12    fraction operator+(const fraction&, const fraction&);
13  }
14
15  int main(  ){
16    numeric::fraction nf;
17    eqn::fraction qf;
18    nf = nf + 1;           // OK: calls numeric::operator+
19    qf = 1 + qf;           // OK: calls eqn::operator+
20    nf = nf + qf;          // Error: no operator+
21    using numeric::fraction;
22    fraction f;            // f is numeric::fraction
23    f = nf + 2;            // OK
24    f = qf;                // Error: type mismatch
25    using eqn::fraction;   // Error: like trying to declare
26                           // fraction twice in the same scope
27    if (f > 0) {
28      using eqn::fraction; // OK: hides outer fraction
29      fraction f;          // OK: hides outer f
30      f = qf;              // OK: same types
31      f = nf;              // Error: type mismatch
32    }
33    int fraction;          // Error: name fraction in use
34  }
```

You can copy names from one namespace to another with a using declaration. Suppose you refactor a program and realize that the numeric::fraction class has all the functionality you need in the equation package. You decide to use numeric::fraction instead of eqn::fraction, but you want to keep the eqn interface the same. So you insert using numeric::fraction; in the eqn namespace.

Incorporating a name into a namespace with a using declaration is not quite the same as declaring the name normally. The new name is just a synonym for the original name in its original namespace. When the compiler searches namespaces under argument-dependent name lookup, it searches the original namespace. Next example shows how the results can be surprising if you are not aware of the using declaration. The eqn namespace declares operator<< to print a fraction, but fraction is declared in the numeric namespace. Although eqn uses numeric::fraction, when the compiler sees the use of operator<<, it looks in only the numeric namespace, and never finds operator<<.

Source Code: *CreatingSynonymDeclarationsWithUsing.cpp*

```
 1  namespace eqn {
 2    using numeric::fraction;
 3    // Big, ugly declaration for ostream << fraction
 4    template<typename charT, typename traits>
 5    basic_ostream<charT,traits>& operator<<(
 6      basic_ostream<charT,traits>& out, const fraction& f){
 7      out << f.numerator(  ) << '/' << f.denominator(  );
 8      return out;
 9    }
10  }
11
12  int main(  ){
13    eqn::fraction qf;
14    numeric::fraction nf;
15    nf + qf;          // OK because the types are the same
16
17    std::cout << qf; // Error: numeric namespace is searched for
18                        // operator<<, but not eqn namespace
19  }
```

The using declaration can also be used within a class. You can add names to a derived class from a base class, possibly changing their accessibility. For example, a derived class can promote a protected member to public visibility. Another use of using declarations is for private inheritance, promoting specific members to protected or public visibility. For example, the standard container classes are not designed for public inheritance. Nonetheless, in a few cases, it is possible to derive from them successfully. Next example shows a crude way to implement a container type to represent a fixed-size array. The array class template derives from std::vector using private inheritance. A series of using declarations make selected members of std::vector public and keep those members that are meaningless for a fixed-size container, such as insert, private.

Source Code: *Importing members with using declarations.cpp*

```
 1  template<typename T>
 2
 3  class array: private std::vector<T>{
 4  public:
 5    typedef T value_type;
 6    using std::vector<T>::size_type;
 7    using std::vector<T>::difference_type;
 8    using std::vector<T>::iterator;
 9    using std::vector<T>::const_iterator;
10    using std::vector<T>::reverse_iterator;
11    using std::vector<T>::const_reverse_iterator;
```

```
12    array(std::size_t n, const T& x = T(  )) : std::vector<T>(n, x) {}
13    using std::vector<T>::at;
14    using std::vector<T>::back;
15    using std::vector<T>::begin;
16    using std::vector<T>::empty;
17    using std::vector<T>::end;
18    using std::vector<T>::front;
19    using std::vector<T>::operator[];
20    using std::vector<T>::rbegin;
21    using std::vector<T>::rend;
22    using std::vector<T>::size;
23  };
```

using Directives

A using directive adds a namespace to the list of scopes that is used when the compiler searches for a name's declaration. Unlike a using declaration, no names are added to the current namespace. Instead, the used namespace is added to the list of namespaces to search right after the innermost namespace that contains both the current and used namespaces. (Usually, the containing namespace is the global namespace.) The using directive is transitive, so if namespace A uses namespace B, and namespace B uses namespace C, a name search in A also searches C. Next example illustrates the using directive.

Source Code: *The using directive.cpp*

```
1   #include <iostream>
2   #include <ostream>
3
4   namespace A {
5     int x = 10;
6   }
7   namespace B {
8     int y = 20;
9   }
10  namespace C {
11    int z = 30;
12    using namespace B;
13  }
14  namespace D {
15    int z = 40;
16    using namespace B; //Harmless but pointless because D::y hides B::y
17    int y = 50;
18  }
19  int main(  ){
20    int x = 60;
21    using namespace A; //Does not introduce names, so there is no
22                       //conflict with x
23    using namespace C;
24    using namespace std;     // To save typing std::cout repeatedly
25    cout << x << '\n';       // Prints 60 (local x)
26    cout << y << '\n';       // Prints 20
27    cout << C::y << '\n';    // Prints 20
28    cout << D::y << '\n';    // Prints 50
29    using namespace D;
30    cout << y << '\n'; // Error: y is ambiguous. It can be found in
31                       // D::y and C's use of B::y.
32  }
```

How to Use Namespaces

Namespaces have no runtime cost. Don't be afraid to use them, especially in large projects in which many people contribute code and might accidentally devise conflicting names. The following are some additional tips and suggestions for using namespaces:

- When you define a class in a namespace, be sure to declare all associated operators and functions in the same namespace.
- To make namespaces easier to use, keep namespace names short, or use aliases to craft short synonyms for longer names.
- Never place a using directive in a header. It can create name collisions for any user of the header.
- Keep using directives local to functions to save typing and enhance clarity.
- Use using namespace std outside functions only for tiny programs or for backward compatibility in legacy projects.