HOUR 2

The Parts of a C++ Program

What You'll Learn in This Hour:

- Why you should choose C++
- The specific parts of any C++ program
- How the parts work together
- What a function is and what it does

Why C++ Is the Right Choice

C++ is the development language of choice for the majority of professional programmers because it offers fast, small programs developed in a robust and portable environment. Today's C++ tools make creating complex and powerful commercial applications a pretty straightforward exercise, but to get the most out of C++ you need to learn quite a bit about this powerful language.

Compared to some languages, C++ is relatively new. Of course, programming itself is only about 60 years old. During that time, computer languages have undergone a dramatic evolution. C++ is considered to be an evolutional improvement of C. The C language itself is almost 30 years old.

Early on, programmers worked with the most primitive computer instructions: **machine language**. These instructions were represented by long strings of ones and zeros. Soon, **assemblers** were invented that could 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**, such as BASIC, translates a program as it reads it, turning the programmer's program instructions or code directly into actions.

Compilers translate the code into what is called object code. The first step in this transformation is called **compiling**. A **compiler** produces an object file. The second step is called **linking**. A **linker** transforms the object file into an executable program. An executable program is one that "runs" on your operating system.

Note: the term *object code* is not related to the concept of objects as used in object-oriented programming (described later in this book).

Because interpreters read the code as it is written and execute the code on the spot, they are easy for the programmer to work with. Compilers introduce the inconvenient extra steps of compiling and linking the code. On the other hand, compilers produce a program that executes faster than the same program executed by an interpreter.

For many years, the principal 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.

Procedural, Structured, and Object-Oriented Programming

In **procedural programming**, programs are thought of as a series of actions performed on a set of data. **Structured programming** was invented to provide a systematic approach to organizing these procedures, and to managing large amounts of data.

The principle idea behind structured programming is as simple as the idea of divide and conquer. Any task that is too complex to be described is broken down into a set of smaller component tasks, until the tasks are small and self-contained enough that they are 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.

Yet, there are problems. The separation of data from the tasks that manipulate the data becomes harder and harder to comprehend and maintain as the amount of data grows. The more things you want to do with that data, the more confusing it becomes.

Procedural programmers find 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, she doesn't usually invent one; she goes to the big bin of transistors and finds one that works for what she needs, or perhaps modifies it. Until object-oriented programming, there was no similar option for a software engineer.

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 three pillars of object-oriented development: encapsulation, inheritance, and polymorphism.

Encapsulation

When an engineer is creating a new device, he wires together component pieces. He can wire in a resistor, a capacitor, and a transistor. The transistor has certain properties and can accomplish certain behaviors. He can use the transistor without understanding the details of how it *works*, as long as he knows what it *does*.

To achieve this, the transistor must be self-contained. It must do one well-defined thing, and it must do it completely. Doing one thing completely is called **encapsulation**.

All the properties of the transistor are encapsulated in the transistor object; they are not spread out through the circuitry. It is not necessary to understand how the transistor works in order to use it effectively.

C++ supports the properties of encapsulation through the creation of user-defined types, called classes. Once created, a well-defined class acts 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. You'll see how to create classes in Hour 7, "Basic Classes."

Inheritance and Reuse

In the late 1980s, I worked for Citibank building a device for home banking. We didn't want to start from scratch; we wanted to get something out into the market quickly. Therefore, we started with the telephone and "enhanced" it. Our new enhanced telephone was a kind of telephone; it just had added features. Thus, I was able to reuse all the calling features of a plain old telephone, but add new capabilities to extend its utility.

C++ supports the idea of reuse through **inheritance**. A new type can be declared that is an extension of an existing type. This new subclass is said to derive from the existing type, and is sometimes called a **derived type**. The enhanced telephone is derived from a plain old telephone and thus inherits all its qualities, but additional features can be added to it as needed. Inheritance and its application in C++ are discussed in Hour 16, "Inheritance."

Polymorphism

The enhanced telephone (ET) behaves differently when receiving a call. Rather than ringing a bell, the screen lights up and a voice says, "You have a call." The phone company doesn't know this, however. They don't send special signals to each kind of phone. The company just pulses the wire and the regular telephone rings, an electronic telephone trills, and ET speaks. Each phone does "the right thing" based on its understanding of the message from the phone company.

C++ supports this 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, and is discussed during Hour 17, "Polymorphism and Derived Classes," and Hour 18, "Advanced Polymorphism."

The Parts of a Simple Program

The simple program from the first hour, `hello.cpp`, had many interesting parts. This section will review this program in more detail. Listing 2.1 reproduces the original version of `hello.cpp` for your convenience.

LISTING 2.1 `hello.cpp` Demonstrates the Parts of a C++ Program

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

```
Hello World!
```

Output

Analysis

On line 0 the file `iostream` is included in the file. As far as the compiler is concerned, it is as if you typed the entire contents of the file `iostream` right into the top of `hello.cpp`.

Examining the `#include`, Character by Character

When you run your compiler, it first calls another program—the **preprocessor**. You don't have to invoke the preprocessor directly; it is called automatically each time you run the compiler.

The first character is the # symbol, which is a signal to the **preprocessor**. The job of the preprocessor is to read through your source code looking for lines that begin with the pound symbol (#). Each time it finds a line beginning with the pound symbol, the preprocessor modifies the code. It is the modified code that is given to the compiler.

Essentially, the preprocessor is an editor of your code—at compile time. The preprocessor directives are the commands for that editor.

The term `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` in the directory that holds all the header files for your compiler. These files are called "h files" or "include files" because they are *included* in source code files and traditionally ended with the extension `.h`.

By the Way

Include File Extensions

The new ANSI-standard include files often do not have any extension at all. In fact, your compiler probably comes with two versions for each of these files. For example, if you check your installation you might find that you have both the older traditional iostream and the new, ANSIstandard iostream. We'll use the new files in this book, as the old files are now obsolete.

If you are using an older compiler, you may need to include the old format of `#include <iostream.h>` .

The file `iostream` (Input-Output-STREAM) is used by `cout`, which assists with writing to the screen.

Notice that `cout` has the designation `std::` in front of it; this tells the compiler to use the standard input/output library. The details of how this works and what is going on here will be explained in more detail in coming hours. For now, you can treat `std::cout` as the name of the object that handles output, and `std::cin` as the name of the object that handles input.

The effect of line 0 is to include the file `iostream` into this program as if you had typed it in yourself. By the time the compiler sees this file, the included file is right there, and the compiler is none the wiser.

Line-by-Line Analysis

Line 2 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. Functions are invoked, (some programmers say they are *called*) by other functions, but `main()` is special. When your program starts, `main()` is called automatically.

The function `main()`, like all functions, must state what kind of value it will return. Once again, `main()` is special; it will always return `int`. The keyword *int* refers to an integer (a whole number) as explained in Hour 3, "Variables and Constants." Returning a value from a function will be discussed in detail in Hour 4, "Expressions and Statements."

Did you Know?

Returning Values?

The value returned from `main()` is actually passed to the operating system so it can be used to detect errors. This is a common technique in production and batch environments so that another tool or script that runs your program can report any error (possibly paging you in the middle of the night if it is a really important process).

All functions begin with an opening brace ({) and end with a closing brace (}). The braces for the `main()` function are on lines 3 and 6. Everything between the opening and closing braces is considered a part of the function.

The braces contain blocks. The outer block of a function in this case. You'll see other types of blocks used a little later.

The meat and potatoes of this program is on line 4. The object `cout` is used to print a message to the screen. We'll cover objects in general beginning in Hour 8, "More About Classes." The object `cout` and its related object `cin` are provided by your compiler vendor and enable the system to write to the screen (`cout`) and read in from the keyboard (`cin`).

Here's how `cout` is used: Write the word `cout` followed by the output redirection operator (<<). You create the output redirection operator by pressing Shift+comma twice. Whatever follows the output redirection operator is written to the screen. If you want a string of characters to be written, be sure to enclose them in double quotes (`"`) as shown on line 4. Note that 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!`

On line 5 we "return" the value `0` to the operating system. On some systems, this is used to signal success or failure to the operating system; the convention is to return `0` for success and any other number for failure. On modern windowing machines, this value is almost never used, and so all the programs in this book will return the value `0`.

The `main()` function ends on line 6 with the closing brace.

Comments

When you are writing a program, it often seems clear and self-evident what you are trying to do in the program. 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's always there.

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

Types of Comments

A **comment** is text that does not affect the operation of the program, but is added to instruct or inform the programmer. There are two types of comments in C++. The double-slash (`//`) comment, which will be referred to as a C++-style comment, tells the compiler to ignore everything that follows the slashes 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 because C++ inherited them from C. Remember, every `/*` must be matched with a closing `*/`.

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.

Watch Out!

Commenting Out Code

Be careful when using C-style comments for commenting out code that contains C-style comments. You can get into a situation where less code is commented out than you expect because C-style comments do not nest. Just because you have two slash-star comments does not mean that you need two star-slash comment marks. The first star-slash comment mark the compiler finds "closes" every slash-star comment that is currently open. A good IDE will apply a different color to commented out code to make it visually obvious. And the compiler will report an error when it finds the extra star-slash comment.

Using Comments in a Simple Sample Program

Comments are free; they don't cost anything in performance, and the compiler ignores them. Listing 2.2 illustrates this.

LISTING 2.2 `comments.cpp` Demonstrates Comments

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

Output

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

Analysis

The comments on lines 4 through 6 are completely ignored by the compiler, as are the comments on lines 8, 11, and 12. The comment on line 8 ends with the end of the line; however, the comments on 4 and 12 require a closing comment mark.

Writing comments well is a skill few programmers master.

It is best to assume your audience can read C++, but can't read your mind. Let the source code tell *what* you are doing, and use comments to explain *why* you are doing it.

Functions

Although `main()` is a function, it is an unusual one. Your operating system invokes `main()` to start your program. Other functions are called, or invoked, from `main()` or from one another during the course of your program.

The function `main()` always returns an `int`. As you'll see in the coming hours, other functions might return other types of values or might return nothing at all.

A program is executed line by line in the order it appears in your source code, until a function is called. Then the program branches off to execute the function.

When the function finishes, it returns control to the line following where the program called the function.

Imagine that you are drawing a picture of yourself. You draw the head, the eyes, the nose, and suddenly your pencil breaks. You "branch off" to the "sharpen my pencil" function. That is, you stop drawing, get up, walk to the sharpener, sharpen the pencil, and then return to what you were doing, picking up where you left off. (I think you were adding a cleft to your chin.)

Calling Functions

When a program needs a service performed, it calls a function to perform the service. When the function returns, the program resumes where it was just before the function was called. Listing 2.3 demonstrates this idea.

Listing 2.3—`callfunc.cpp` Demonstrating a Call to a Function

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

Output

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

Analysis

The function `DemonstrationFunction()` is defined on lines 4–7. As a definition, it shows what to do, but does not do anything until called. When it is called in line 15, it prints a message to the screen and then returns.

Line 12 is the beginning of the actual program. On line 14, `main()` prints out a message saying it is in `main()`. After printing the message, line 15

calls `DemonstrationFunction()`. This call causes the commands in `DemonstrationFunction()` to execute. In this case, the entire function consists of the code on line 6, which prints another message. When `DemonstrationFunction()` completes (line 7), it returns to where it was called from. In this case, the program returns to line 16, where `main()` prints its final line.

Using Functions

Functions either return a value, or they return data type `void`—nothing. A function that adds two integers might return the sum, and thus would be defined as returning 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 (line 4) and a body (lines 5–7). The header consists of the return type, the function name, and the list of parameters (if any) 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 book will use the terms interchangeably.

The name of the function and its parameters (that is the header without the return value) is called the function's **signature**. 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 might 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.

Using Parameters with Functions

Listing 2.4 demonstrates a function that takes two integer parameters and returns an integer value.

LISTING 2.4 `func.cpp` **Demonstrates a Simple Function**

```
0:  #include <iostream>
1:
2:  int Add (int x, int y)
3:  {
4:      std::cout << "In Add(), received " << x << " and " << y << "\n";
5:      return (x+y);
6:  }
7:
8:  int main()
9:  {
10:      std::cout << "I'm in main()!\n";
11:      std::cout << "\nCalling Add()\n";
12:      std::cout << "The value returned is: " << Add(3,4);
13:      std::cout << "\nBack in main().\n";
14:      std::cout << "\nExiting...\n\n";
15:      return 0;
16:  }
```

Output

```
I'm in main()!

Calling Add()
In Add(), received 3 and 4
The value returned is: 7
Back in main().

Exiting...
```

Analysis

The function `Add()` is defined on line 2. It takes two integer parameters and returns an integer value. The program itself begins on line 10 where it prints a message.

On line 12 `main()` prints a message and then prints the value returned from calling `Add(3,4)`. This invokes the function and processing branches to line 2. The two values passed in to `Add()` are represented as the parameters x and y. These values are added together and the result is returned on line 5.

The return value is printed on line 12, and is shown in the output (7). `Main()` then prints a message on lines 13 and 14 and the program exits when `main()` returns (returning control to the operating system).

Q&A

Q ***What does*** `#include` ***do?***

A This is a directive to the preprocessor, which runs when you call your compiler. This specific directive causes the file named after the word `include` to be read in as if it were typed in at that location in your source code.

Q ***What is the difference between `//` comments and `/*` style comments?***

A The double-slash comments (`//`) "expire" at the end of the line. Slash-star (`/*`) comments are in effect until a closing comment (`*/`). Remember, not even the end of the function terminates a slash-star comment; you must put in the closing comment mark or you will get a compile-time error.

Q ***What differentiates a good comment from a bad comment?***

A A good comment tells the reader why this particular code is doing whatever it is doing, or explains what a section of code is about to do. A bad comment restates what a particular line of code is doing. Lines of code should be written so that they speak for themselves: Reading the line of code should tell you what it is doing without needing a comment. Comments are an art form in and of themselves.

Workshop

Now that you've had the chance to learn about some of the pieces of a C++ program, there are a few questions to be answered and a couple of exercises to be performed, which will firm up your knowledge of the compiler.

Quiz

1. What data type does `main` return?
2. What do the braces do?
3. What is the difference between a compiler and an interpreter?
4. Why is reuse so important?

Exercises

1. Take the program in Listing 2.1 and split line 4 into multiple lines (Hello on one line and World on another). Has the way the result is printed changed any? How can you make them print on different lines?
2. With your compiler, try compiling Listing 2.1 using `#include <iostream>` and `#include <iostream.h>`. It may accept one or both of the filenames; that fact could be useful to you in the future.
3. Try compiling and linking some of the same programs using the IDE and command-line compiler. Are the final results different with any of your programs?

Quiz Answers

1. `main` returns type `int`—that is, an integer value.
2. Braces show the beginning and end of a block. They hold the contents of the function or program we wrote.
3. A compiler converts the entire program to machine language before it is executed. An interpreter converts each line as it is being executed.
4. Why do we reuse ideas outside of programming? Because it is easier, faster, and less expensive than building everything from scratch! But many programmers like to retain tight control over their code and work. So the sharing and thinking processes have to be taught.