C++ from the ground up

LEARN C++ FROM THE MASTER

herbert SCHILDT

OSBORNE

Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations or book distributors outside of the U.S.A., please write to Osborne **McGraw-Hill** at the above address.

C++ from the Ground Up

567890 DOC 998765

ISBN 0-07-881969-5

Publisher
Lawrence Levitsky

Acquisitions Editor
Jeff Pepper

Project Editor
Nancy McLaughlin

Technical Editor
James Turley

Proofreader
Audrey Baer Johnson

Computer Designer
Marcela V. Hancik

Quality Control Specialist
Joe Scuderi

Illustrator
Rhys Elliott

Interior Designer
Marla Shelasky

Indexer
Sheryl Schildt

Cover Design
Ted Mader Associates

```
    cout << " Inside myfunc() ";
}
```

The program works like this. First, **main()** begins, and it executes the first **cout** statement. Next, **main()** calls **myfunc()**. Notice how this is achieved: the function's name, **myfunc**, appears, followed by parentheses, and finally by a semicolon. A function call is a C++ statement and, therefore, must end with a semicolon. Next, **myfunc()** executes its **cout** statement, and then returns to **main()** at the line of code immediately following the call. Finally, **main()** executes its second **cout** statement and then terminates. Hence, the output on the screen is this:

In main( ) Inside myfunc( ) Back in main( )

There is one other important statement in the preceding program:

```
void myfunc(); // myfunc's prototype
```

A prototype declares a function prior to its first use.

As the comment states, this is the *prototype* for **myfunc()**. Although we will discuss prototypes in detail later in this book, a few words are necessary now. A function prototype declares the function prior to its definition. The prototype allows the compiler to know the function's return type, as well as the number and type of any parameters that the function may have. The compiler needs to know this information prior to the first time the function is called. This is why the prototype occurs before **main()**.

As you can see, **myfunc()** does not contain a **return** statement. The keyword **void**, which precedes both the prototype for **myfunc()** and its definition, formally states that **myfunc()** does not return a value. In C++, functions that don't return values are declared as **void**.

Function Arguments

An argument is a value passed to a function when it is called.

It is possible to pass one or more values to a function. A value passed to a function is called an *argument*. In the programs that you have studied so far, none of the functions take any arguments. Specifically, neither **main()** nor **myfunc()** in the preceding examples have an argument. However, functions in C++ can have anywhere from no arguments at all to many arguments. The upper limit is determined by the compiler you are using, but the proposed C++ standard specifies that a function must be able to take at least 256 arguments.

Here is a short program that uses one of C++'s standard library (i.e., built-in) functions, called **abs()**, to display the absolute value of number. The **abs()**

function takes one argument, converts it into its absolute value, and returns the result.

```
// Use the abs() function.
#include <iostream.h>
#include <stdlib.h> // required by abs()

main()
{
  cout << abs(-10);

  return 0;
}
```

Here, the value –10 is passed as an argument to **abs()**. The **abs()** function receives the argument that it is called with and returns its absolute value, which is 10 in this case. Although **abs()** takes only one argument, other functions can have several. The key point here is that when a function requires an argument, it is passed by specifying it between the parentheses that follow the function's name.

The return value of **abs()** is used by the **cout** statement to display the absolute value of –10 on the screen. The reason this works is that whenever a function is part of a larger expression, it is automatically called so that its return value can be obtained. In this case, the return value of **abs()** becomes the value of the right side of the << operator and is, therefore, displayed on the screen.

Notice one other thing about the preceding program: it also includes the header file **stdlib.h**. This is the header file required by **abs()**. In general, whenever you use a library function, you must include its header file. The header file provides the prototype for the library function, among other things.

A parameter is a variable defined by a function that receives an argument.

When you create a function that takes one or more arguments, the variables that will receive those arguments must also be declared. These variables are called the *parameters* of the function. For example, the function shown next prints the product of the two integer arguments used in the call to the function.

```
void mul(int x, int y)
{
  cout << x * y << " ";
}
```

Each time **mul()** is called, it will multiply the value passed to **x** by the value passed to **y**. Remember, however, that **x** and **y** are simply the operational variables that receive the values you use when calling the function.

Consider the following short program, which illustrates how to call **mul()**:

```
// A simple program that demonstrates mul().

#include <iostream.h>

void mul(int x, int y); // mul()'s prototype

main()
{
  mul(10, 20);
  mul(5, 6);
  mul(8, 9);

  return 0;
}

void mul(int x, int y)
{
  cout << x * y << " ";
}
```

This program will print 200, 30, and 72 on the screen. When **mul()** is called, the C++ compiler copies the value of each argument into the matching parameter. That is, in the first call to **mul()**, 10 is copied into **x** and 20 is copied into **y**. In the second call, 5 is copied into **x** and 6 into **y**. In the third call, 8 is copied into **x** and 9 into **y**.

If you have never worked with a language that allows parameterized functions, then the preceding process may seem a bit strange. Don't worry; as you see more examples of C++ programs, the concept of arguments, parameters, and functions will become clear.

Remember: The term *argument* refers to the value that is used to call a function. The variable that receives the value of an argument is called a parameter. In fact, functions that take arguments are called parameterized functions.

In C++ functions, when there are two or more arguments, they are separated by commas. In this book, the term *argument list* will refer to comma-separated arguments. The argument list for **mul()** is x,y.

Functions Returning Values

Many of the C++ library functions that you will use return values. For example, the **abs()** function used earlier returned the absolute value of its argument. Also, functions you write may return values to the calling routine.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.