Functions

A function is a self-contained block of code with a specific purpose. A function has a name that both identifies it and is used to call it for execution in a program. The name of a function is governed by the same rules as those for a variable.

You pass information to a function by means of **arguments** specified when you invoke it. These arguments need to correspond with **parameters** that appear in the definition of the function. The arguments that you specify replace the parameters used in the definition of the function when the function executes. The code in the function then executes as though it was written using your argument values.

Figure 1 illustrates the relationship between arguments in the function call and the parameters specified in the definition of the function.

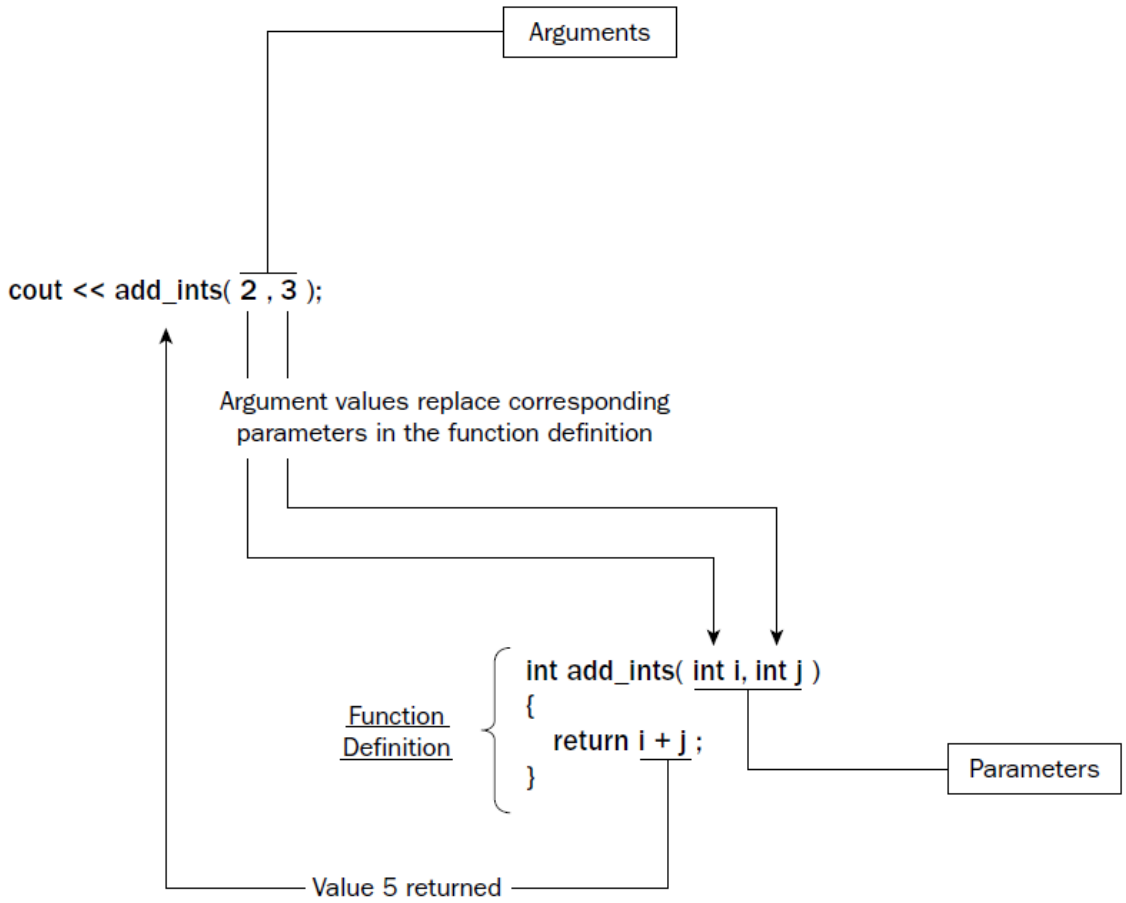

Figure 1

In this example, the function returns the sum of the two arguments passed to it. In general, a function returns either a single value to the point in the program where it was called, or nothing at all, depending on how the function is defined.

Why Do We Need Functions?

One major advantage that a function offer is that it can be executed as many times as necessary from different points in a program. Without the ability to package a block of code into a function, programs would end up being much larger because you would typically need to replicate the same code at various points in them. But the real reason that you need functions is to break up a program into easily manageable chunks for development and testing.

Imagine a really big program let's say a million lines of code. A program of this size would be virtually impossible to write without functions. Functions enable you to segment a program so that you can write the code piecemeal, and test each piece independently before bringing it together with the other pieces. It also allows the work to be divided among members of a programming team, with each team member taking responsibility for a tightly specified piece of the program.

Structure of a Function

As you have seen when writing the function main(), a function consists of a **function header** that identifies the function, followed by the **body** of the function between curly braces containing the executable code for the function. Let's look at an example. You could write a function to raise a value to a given power, that is, to compute the result of multiplying the value x by itself n times, which is x^n:

```
// Function to calculate x to the power n,
// with n greater than or equal to 0

double power(double x, int n)   // Function header

{                               // Function body starts here...

        double result = 1.0;    // Result stored here
        for (int i = 1; i <= n; i++)
                result *= x;
        return result;

}                               // ...and ends here
```

The Function Header

Let's first examine the function header in this example. The following is the first line of the function.

double power(double x, int n) // Function header

It consists of three parts:

- ❑ The type of the **return value** (***double*** in this case)
- ❑ The name of the function, (***power*** in this case)
- ❑ The parameters of the function enclosed between parentheses (***x*** and ***n*** in this case, of types double and int respectively)

The return value is returned to the calling function when the function is executed, so when the function is called, it results in a value of type double in the expression in which it appears. Our function has two parameters: x, the value to be raised to a given power which is of type double, and the value of the power, n, which is of type int. The computation that the function performs is written using these parameter variables together with another variable, result, declared in the body of the function. The parameter names and any variables defined in the body of the function are local to the function.

Note that no semicolon is required at the end of the function header or after the closing brace for the function body.

The General Form of a Function Header

The general form of a function header can be written as follows.

return_type function_name(parameter_list)

The return_type can be any legal type. If the function does not return a value, the return type is specified by the keyword void. The keyword void is also used to indicate the absence of parameters, so a function that has no parameters and doesn't return a value would have the following function header.

void my_function(void)

An empty parameter list also indicates that a function takes no arguments, so you could omit the keyword void between the parentheses like:

void my_function()

A function with a return type specified as void should not be used in an expression in the calling program. Because it doesn't return a value, it can't sensibly be part of an expression, so using it in this way causes the compiler to generate an error message.

The Function Body

The desired computation in a function is performed by the statements in the function body following the function header. The first of these in our example declares a variable result that is initialized with the value 1.0. The variable result is local to the function, as are all automatic variables declared within the function body. This means that the variable result ceases to exist after the function has completed execution.

The calculation is performed in the for loop. A loop control variable i is declared in the for loop which assumes successive values from 1 to n. The variable result is multiplied by x once for each loop iteration, so this occurs n times to generate the required value. If n is 0, the statement in the loop won't be executed at all because the loop continuation condition immediately fails, and so result is left as 1.0.

There is nothing to prevent you from using the same names for variables in other functions for quite different purposes. Indeed, it's just as well this is so because it would be extremely difficult to ensure variables names were always unique within a program containing a large number of functions, particularly if the functions were not all written by the same person.

The return Statement

The return statement returns the value of result to the point where the function was called. The general form of the return statement is:

return expression;

where expression must evaluate to a value of the type specified in the function header for the return value. The expression can be any expression you want, as long as you end up with a value of the required type. It can include function calls even a call of the same function in which it appears.

If the type of return value has been specified as void, there must be no expression appearing in the return statement. It must be written simply as:

return;

Using a Function

At the point at which you use a function in a program, the compiler must know something about it to compile the function call. It needs enough information to be able to identify the function, and to verify that you are using it correctly. Unless you the definition of the function that you intend to use appears earlier in the same source file, you must declare the function using a statement called a **function prototype**.

Function Prototypes

A prototype of a function contains the same information as appears in the function header, with the addition of a semicolon. Clearly, the number of parameters and their types must be the same in the function prototype as they are in the function header in the definition of the function.

The prototypes for the functions that you call from within another function must appear before the statements doing the calling and are usually placed at the beginning of the program source file.

For the power() function example, you could write the prototype as:

```
double power(double value, int index);
```

Note that we have specified names for the parameters in the function prototype that are different from those we used in the function header when we defined the function. This is just to indicate that it is possible. Most often, the same names are used in the prototype and in the function header in the definition of the function.

If you like, you can even omit the names altogether in the prototype, and just write:

```
double power(double, int);
```

This provides enough information for the compiler to do its job; however, it's better practice to use some meaningful name in a prototype because it aids readability and, in some cases, makes all the difference between clear code and confusing code. If you have a function with two parameters of the same type (suppose our index was also of type double in the function power(), for example), the use of suitable names indicates which parameter appears first and which second.

Example:

```
// Declaring, defining, and using a function
#include <iostream>
using namespace std;
double power(double x, int n); // Function prototype

int main(void)
{
        int index = 3;          // Raise to this power
```

```
        double x = 3.0;       // Different x from that in function power
        double y = 0.0;

        y = power(5.0, 3);   // Passing constants as arguments
        cout << endl << "5.0 cubed = " << y;
        cout << endl << "3.0 cubed = "<< power(3.0, index);
                                        // Outputting return value

        x = power(x, power(2.0, 2.0)); // Using a function as an
                                       //argument
         cout << endl<< "x = " << x;
        cout << endl;
        return 0;
}

// Function to compute positive integral powers of a double value
// First argument is value, second argument is power index

double power(double x, int n)
{                                      // Function body starts here...
        double result = 1.0;          // Result stored here
        for (int i = 1; i <= n; i++)
        result *= x;
        return result;
}                                      // ...and ends here
```

This program shows some of the ways in which you can use the function power(), specifying the arguments to the function in a variety of ways. If you run this example, you get the following output:

```
5.0 cubed = 125
3.0 cubed = 27
x = 81
```

The function **power()** is used next in this statement:

```
x = power(x, power(2.0, 2.0)); // Using a function as an argument
```

Here the power() function is called twice. The first call to the function is the rightmost in the expression, and the result supplies the value for the second argument to the leftmost call. Although the arguments in the sub-expression power(2.0, 2.0) are both specified as the double literal 2.0, the function is actually called with the first argument as 2.0 and the second argument as the integer literal, 2. The compiler converts the double value specified for the

second argument to type int because it knows from the function prototype (shown again below) that the type of the second parameter has been specified as int.

double power(double x, int n); // Function prototype

The double result 4.0 is returned by the first call to the power() function, and after conversion to type int, the value 4 is passed as the second argument in the next call of the function, with x as the first argument. Because x has the value 3.0, the value of 3.04 is computed and the result, 81.0, stored in x. This sequence of events is illustrated in Figure 2.

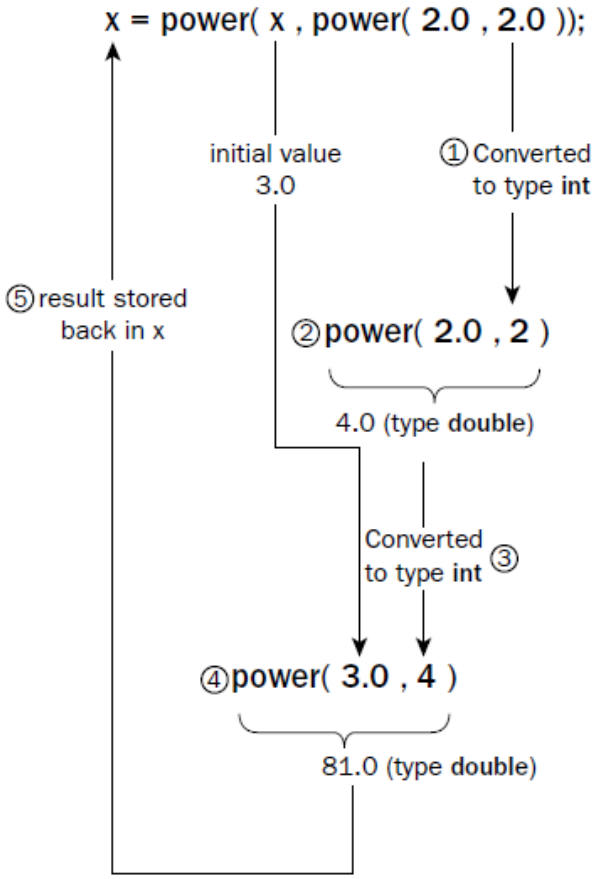

Figure 2