Chapter 7: Functions

Notes

- Function prototypes can be local, but function definitions cannot.

Modularization

At this point in your learning you could write any of a number of programs using your knowledge of user input, variables, flow control, and rudimentary output. But in order to do all of this you've had to cram all the instructions into the single function '`main()`'.

A function is a named block of code that performs some sort of action. A function may contain all of the logic to perform the action or it may need to call on other functions. If you were to drive to the store you'd first have to find your keys, locate your other shoe, get in the car, turn it on, etc. etc. Large actions like "drive to store" typically encompass many smaller actions. You've already seen this with expressions which can encompass many operations. The term '*encompass*', incidentally means to contain. You might also hear '*encapsulate*' which means the same thing. Programmers like to use fancy terms, it's a bad habit.

Naming a function is much like naming a variable and involves the same thing: an identifier. The name of the one function we've been using is '`main`'. It's a nice simple name, but I might have preferred '`start`' or '`entrance`' because the program doesn't necessarily have to exist there, but that's where it begins[1].

I use the term 'code block' quite intentionally in my initial synopsis of what functions are. A function is simply a statement block with a name. When that name is called upon, the code in the associated statement block is executed. With '`main`' the compiler sets it up so that is where execution begins.

The usage of functions in a program makes it more modular. Rather than all the code sitting in one place (all the eggs in one basket) it is placed into modules (an Easter egg hunt?).

Prototyping and Definition

To create your own function you must define it and usually declare it as well. When you declare something in programming, you are acknowledging its presence. With simple

[1] For writing programs on specific platforms or for certain API's you may be required to use other terms; like '`WinMain`' for Windows™.

variables, like the ones we've worked with, there isn't much more to them than their presence alone. We can tell from their declaration (type and name) exactly what they are about. This is not the case with functions. When you declare a function you will simply acknowledge that it exists. The technical term for declaring a function is known as function *prototyping*. You haven't done any of this yet, because '`main`' doesn't need to be prototyped. You don't need to declare '`main`' because the compiler already assumes that it exists. However, to be able to write your own functions you will need to know how to do this.

The second, required, part of creating a function is *defining* it. This we have done so far. We define a function by giving it a body. A function *body* is the function's associated statement block. You will probably never hear of a function's 'associated statement block' outside of this book, so get used to the term 'function body'.

A declaration is saying that something exists and a definition is the actual *thing* that has been declared.

First Example

I can't go further without actually *showing* you what I'm talking about. The following program declares the function '`CountToTen`' and then defines it after '`main`':

```
01  #include <iostream.h>
02
03  void CountToTen(void);
04
05  int main()
06  {
07      CountToTen();
08      return 0;
09  }
10
11  void CountToTen(void)
12  {
13      int num = 1;
14      while (num <= 10)
15      {
16          cout << num << endl;
17      }
18  }
```

There are three important lines here. I'll begin by describing line 3:

```
03  void CountToTen(void);
```

This is the function prototype or declaration. It tells the compiler that the function '`CountToTen`' exists in the program and describes the input to a function and the

output from a function. The compiler will then make sure that the name is linked to an actual block of code.

```
07      CountToTen();
```

In '`main`' we call the function we have created. The compiler takes care of finding the function definition since we declared it earlier so it (function definition) may exist *after* the code that we call it in. I take advantage of this above. I don't actually *define* the function until after the definition of '`main`'. Calling a function causes the execution to jump from the current function into the specified one.

```
11  void CountToTen(void)
```

This is the beginning of the function definition. It must be immediately followed by the function body (the associated statement block). A function definition sometimes looks exactly like the prototype; and certainly has all the same input and output descriptions.

The body of a function looks like any other statement block. In fact you'll see that I simply ripped the code from '`main`' in a program in the previous chapter and inserted it as the function body.

Return

In the last example the function I created simply existed and when called it performed an operation. There was no data sent to the function nor was there any returned. But there are methods to do both and in fact later on you will see how they can be combined. For now we will stick to the traditional methods and start with output *from* the function. This is known as a function *return value*.

A return value is passed back after the function has executed. When calling the function, the call will represent the function's return value. For example, the following program calls a function which returns '5':

```
01  #include <iostream.h>
02
03  int GiveMe5(void);
04
05  int main()
06  {
07      cout << "Function gives " << GiveMe5() << endl;
08      return 0;
09  }
10
11  int GiveMe5(void)
12  {
13      return 5;
14  }
```

For a function to return a value you must first declare and define the *type* of the return value. In the case of our program above I prototype and define the function to return an '`int`'. The return value's type should precede the name of the function. It is much like declaring the type of a variable, since it *does* come before (what a koinky dink!).

When a function returns a value it can be used as part of a larger expression as I have done. The call of the function will be represented by the value that the function returns. In the above program the function returns '5' so the '`GiveMe5()`' on line 7 can be seen as '5'. Since this is the case, the output of the program is simply:

```
5
```

Yes, an amazing revelation. A function's return type can be any of the valid data types, just like variables. In fact you could return a variable as well as a literal. The syntax of '`return`' is simply:

```
return expression;
```

This is much like everything you've seen so far with flow control statements. The expression can be anything so long as it represents some sort of value. Bearing that in mind, here is a slightly tastier version of the above program:

Void

You have seen this word thrown about minimally thus far; luckily I was able to shield it from you earlier by omitting it altogether. The keyword '`void`' in C++ means what it means: emptiness, nada, black hole, nothing, etc. This is a special type that functions can have and variables cannot.[2] If a function returns this then it simply does not return any value at all. A function with this return type cannot be used as part of an expression because it represents no value.

In C++ there is an alternative which works slightly the same: a non-existent return type. It is possible to declare a function beginning with its name and leaving out the type altogether. This is what we have done with '`main()`' all this time. A function of this type may give compiler warnings, but works and works in curious ways. I don't currently know what the correct *standard* implementation of this implies, but so far I have seen it yield one of two effects: (1) the function defaults to returning an '`int`' or (2) the function defaults to returning '`void`'.

Either way it's not a good idea to do this. A function should be defined with a clear idea of its purpose and that should include its return value.

Oh and on a last note, you can use '`return`' *without* an expression in a function returning '`void`' to cause it to break out (or *return*) at that point. This has the same

[2] The exception to this rule is void *pointer* variables, as seen in later chapters.

effect as using the ‘break’ statement from within a loop. Execution is broken from that controled block.

Correct Main

A more correct implementation of the ‘`main()`’ function returns an integer type ‘`int`’. Traditionally this will return a zero if no problems occur, which means an empty ‘`main()`’ should look like so:

```
01  int main()
02  {
03      return 0;
04  }
```

This return value is used as the *exit code* of the program. Whatever launched the program can check on this code when it ends to check for a success or failure. If this exit code is non-zero, the meaning of it is dependent upon the program. However returning ‘-1’ is seen as a general failure. Other return codes (synonymous with exit code) may be used by the program. To be effective exit codes have to be utilized by the launching system (albeit another program or the operating system itself).

Even though the rule of thumb is to return an integer zero from the program, I have yet to find a situation where not doing this causes catastrophic effects. Many people have been aggrieved at me for taunting them with this. ☺ They *insist* that all programs should return an integer and for success it should be zero. It is true that it is not a good idea to return an error if there was none; however the return type has never seemed very important (though the compiler will whine). If you do not provide a return type for ‘`main()`’ (either blank or ‘`void`’) any good compiler (and all that I have found) will automatically have the program return zero anyway.

Parameters

If you haven’t guessed, the right side of the function prototype is for declaring the function *parameters*. Parameters are values passed as *input* to the function. Most functions require input of some kind, but not necessarily through parameters. For example, a function might utilize a global variable that is processed or modified. However, for this section we will be focusing slowly on parameters which are explicitly *passed* to the function.

Passing parameters is the process of sending a value to the function from the caller. Parameters to a function are like operands to an operator. Except that there can be many, *many* parameters. The type and amount of values passed to a function is both prototyped and defined. First let us explore the prototype.

Currently I have been using '`void`' in place of a *parameter list* for my functions. A parameter list is a list of data and names that describe each of the parameters the function will input. This list is much like declaring multiple variables because each item is separated by commas. In a function prototype naming parameters in the list is optional, but types are required. So, if we were to make a function that *takes* two parameters (as in you *pass* it two parameters and it takes them), both of type '`int`', the prototype would look like this:

```
void function(int, int);
```

In the function definition each parameter must have a type *and* a name. For example, the definition for the above function might start like:

```
void function(int x, int y)
{
```

When a function is called, each of the parameters in the list is created just as if you had declared the variable yourself. Parameter variables have a scope limited to the function body. When the function ends, the variables disappear. Let's look at a functional (pun pun) example:

```
01  #include <iostream.h>
02
03  int Add(int, int);
04
05  int main()
06  {
07      cout << "2 + 2 = " << Add(2,2) << endl;
08      return 0;
09  }
10
11  int Add(int x, int y)
12  {
13      return x + y;
14  }
```

This program answers the age old mystery: "What is 2 + 2?" Yes, computers are truly a wonderful invention! It also demonstrates function return values and input parameters with a function called '`Add`'. The function has two '`int`' parameters and returns an '`int`' type. Its sole purpose is to add the two values passed in and return the result.

The flow of this program runs as follows. First the text '2 + 2 = ' is printed. Next the function call to '`Add`' is encountered. Notice that passing the values *to* the function is much like when you define them *for* the function. Each value is matched with a corresponding parameter, by the order in which both are written. So in the above the first '2' is assigned to '`x`' and the second is assigned to '`y`'. It happens exactly as if you had written the following:

```
int x = 2;
```

```
int y = 2;
```

After the parameters have been passed, the function body is entered. Here the '`x`' and '`y`' parameters (which are just normal variables) are added together. The result is returned. Now the execution resumes where we left off calling the function, except now we have a value: '4'. This is then printed, followed by printing an end of line, and then the program is finished. As a series of operations this program might look like so:

```
cout << "2 + 2 = "
int x = 2;
int y = 2;
intermediate_result = x + y;
cout << intermediate_result;
cout << endl;
```

However, remember that parameter variables have a scope limited to the body of the function they were defined for. You wouldn't be able to use '`x`' and '`y`' inside main, nor would the function be able to use any variables declared in '`main()`'. Let's look at this yet, another way. The following program does exactly the same thing without a function (it uses a nested statement block instead):

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int Add;
06      {
07          int x = 2, y = 2;
08          Add = x + y;
09      }
10      cout << "2 + 2 = " << Add << endl;
11      return 0;
12  }
```

The one difference is the variable '`Add`' in which I store the result of adding '`x`' and '`y`' together. The compiler will automatically generate intermediate result storage units for any complex operations, including calls to functions. Where did you think the results went? Not all results go into variables.

Passing by Value

All of the things I've done with parameters so far has involved passing them by value. Passing in this way causes new variables to get created and have their values set to the value that is passed in. With the function '`Add`' in the last section, the first parameter was passed in by value and therefore a new variable, '`x`', was created and its value was set to the value passed in. This is the most common way to give a function input and the advantage is that when you call the function you can use pass expressions as parameters because it is their results which will be assigned to the function's parameter variables:

```
01  #include <iostream.h>
02
03  int Add(int, int);
04
05  int main()
06  {
07      cout << "(2+8) + 2 = " << Add(2+8,2) << endl;
08      return 0;
09  }
10
11  int Add(int x, int y)
12  {
13      return x + y;
14  }
```

In the above I use an expression for the first parameter. But the expression isn't passed to the function, its *result* is. So, in the above '`x`' would be set to '10' and '`y`' would be set to '2' (as usual, that '`y`' is so damn boring). Since you can use any expression (that results in a value) as a parameter to a function you could, if you so desired, use a function's return value as the parameter to another function, use a variable, use a variable assignment expression, etc. etc. The following program is a bit of a crazy take on this infamous '`Add`' function program:

```
01  #include <iostream.h>
02
03  int Add(int, int);
04
05  int main()
06  {
07      int x = 2;
08      cout << "(2+8) + 2 = " << Add(Add(x+8),2) << endl;
09      return 0;
10  }
11
12  int Add(int x, int y)
13  {
14      return x + y;
15  }
```

Output from this program would be the same as before, but I've done several things. Not only is the first parameter (for '`x`') the return value from the same function, but I use a variable called '`x`' in the call to it.

I purposely used a variable called 'x' to demonstrate scope. The value stored in the '`x`' variable in the function '`main()`' is completely different than the value stored in the '`x`' variable in the function '`Add()`'. Modifying either's value would *not* affect the other '`x`' variables.

Passing by Reference

It is very possible to *link* a parameter with a source variable. This is known as passing by reference. You basically pass a reference to a variable as a parameter. Any changes made to the parameter in the function will affect the variable that was passed in. The two variables (the one you pass in and the function parameter) have different names but they are both linked to the same value and storage unit. The parameter which represents the variable passed in is known as the *reference parameter*.

Think of it like passing your whole wallet to the clerk instead of counting out the money to him. He has complete control over the contents of your wallet, just as a reference parameter does over the variable that is passed in. There are two downsides to this type of value passing: it can be dangerous because the function now has complete control over the data and you cannot pass in an expression, you must use a variable.

A parameter must be known as being a reference parameter before you can use it as such. Both the function prototype and definition must make note of this. To describe a reference parameter you simply precede its name (follow its type) by an ampersand (`&`).

The following program is a take on our last adding function. This one is called '`AddTo`' and it adds the second parameter to the first, returning nothing:

```
01  #include <iostream.h>
02
03  void AddTo(int&, int);
04
05  int main()
06  {
07      int num = 2;
08      AddTo(num, 2);
09      cout << "2 + 2 = " << num << endl;
10      return 0;
11  }
12
13  void AddTo(int &x, int y)
14  {
15      x += y;
16  }
```

I used 'num' on purpose because it is different than the parameter name 'x'. If you run this program, you will see the same boring output:

```
2 + 2 = 4
```

The interesting part is how we got that result. We pass '`num`' to '`AddTo`' along with '2'. But the value of '`num`' is not assigned to the parameter '`x`'. Instead, '`x`' is initialized simply as a new name; the rest of its information is the same as '`num`'. It isn't so much that a new thing has been created, but 'num' has been given an alternative identifier. We do this in the real world as well. We may take on nicknames or alternative titles at work.

These *references to us* are not the same as our name, but they still *refer* to us. Such is the case with references.

<analogy picture of nickname vs real name referring to same person>

For the time that 'num' (or any other variable passed as the first parameter to 'AddTo') is inside 'AddTo' it will be known as 'x'. Anytime 'x' is modified in anyway, it is actually the value of 'num' that is changed. And likewise, the value of 'x' is drawn from 'num' rather than its own storage unit.

<picture of 'num' and 'x' sharing same value>

When you pass by reference, the variable types *must* be the same and you cannot use literals or constants. The following lines would all fail bearing that the 'AddTo' function is the same as the one I showed you above:

```
int num = 2;
float flnum = 2.0;
AddTo(flnum, 2);
AddTo(2, 2);
AddTo(num + 2, 2);
```

The last line *looks* like it should work, but it doesn't. Why? When you perform a regular addition between two operands, the result is always a new *intermediate* variable. The addition is not changing either operand. If it did we wouldn't be able to simply call '2 + 2' because those are literals and cannot be changed. When the above calls 'num + 2' the result is a new constant with the value of '4'. We can't do anything with this value other than assign it to something.

However, we still can add '2' to 'num' *and* pass 'num' by reference at the same time. You might have already guessed it. Assignment operators of any kind result in the destination operand. So, if we were to call:

```
num = 2;
```

The result of this expression is 'num' again which now has a value of '2'. Likewise, we could pass this expression in as a reference to 'AddTo' and it would be valid:

```
AddTo(num = 2, 2);
```

The reference will still be to 'num' because the result of the expression is 'num' which is then referenced by 'x' (phew!). But there is more than just plain assignment if you remember your operators. We have arithmetic assignment like addition assignment, subtraction assignment, etc. The following is valid because it adds '2' to 'num' and then results in 'num':

```
AddTo(num += 2, 2);
```

References will be covered in more detail after I've shown you how to create complex data types (this is when references shine like the almighty sun).

Constant Parameters

When a program is executed, reference parameters are usually dealt with faster then value parameters because a new variable isn't being created. So you may be inclined to use them for that reason. But then you may not want them to be modified within the function either. Sure you can rely on the function behaving itself, especially if you wrote it, but what if someone else uses it? It is a good idea to make non-modifiable parameters *constant*. By doing this, you can pass variables to the function in the same way, but the function will not have access to modify them in anyway. To make a parameter constant, precede its type with the keyword '`const`':

```
int Add(const int &x, const int &y)
{
    return x + y;
}
```

The above function adds the two parameters together and returns the result. It does not have permission to modify either because both parameters are marked with '`const`'. Once a few chapters ago I glossed over constants, but this is where they shine. Later on you will learn of more complex variables that truly benefit from being passed by reference but constant. In this case it's a little silly, but it works. ☺ Beware of the '`const`' keyword as suddenly the compiler may become very picky.

You can mark normal parameters as constant as well, though it is rarely useful to do so. The following is the same function except it uses normal parameters that are constant:

```
int Add(const int x, const int y)
{
    return x + y;
}
```

Since modifying either of these normal parameters wouldn't effect the original variable (if any) passed in, this isn't very necessary or useful. But it's possible. There *are* times, though, when this comes in handy with "globs" of data. There'll be more on that later, chill.

Scope

Parameters and variables declared within a function have a scope limited to that function and any statement blocks within that function. The same variables cannot be used in functions that are called from the current function. This means that any variables you

declare in '`main()`' you cannot use in functions you call from there unless they are explicitly passed in as parameters.

For instance, in a program above I defined a function called '`AddTo`' which I called from '`main()`'. Any variable I created in '`main()`' could not be used in '`AddTo`' unless it was passed in. This is the purpose of passing parameters, because otherwise you don't have access to the *caller*'s variables.

Remember that each function is its own separate module of logic that can be passed parameters and return a result. Imagine '`main()`' as a continent and '`AddTo`' as an island where each parameter, as well as the return value, is a bridge. Only what you pass to the island through the bridges can be used there. Let's say that each variable is a crate and its value is an amount of bananas. If you want those crates from the mainland – they have to go across a bridge. If you want crates from the island – they too must go across a bridge. Also, the direction that you could go on the bridges might be one of three things: in for passing by value, in/out for passing by reference (since what you bring in you can change), and out for returning a value. '`AddTo`' would look like this:

<island analogy picture with AddTo>

Nothing can be used unless it is *sent* to the function. A function with '`void`' (no) return value and a '`void`' (no) parameter list cannot access any local variables except for its own. Don't forget about global variables. It is possible for any bit of C++ code to access a global variable. Think of a helicopter in our island analogy … it doesn't need to use the bridges to be used.

Author's Preference: Making global variables so they can be used in functions is dangerous *and* sloppy. If you don't use globals sparingly, you will soon come up with design flaws *and* flack from yours truly (heating up my whip – beware!).

Default Parameter Values

It is possible to define a function in C++ to have a default value for any number of parameters. If a parameter has a default value, then it is used to initialize that parameter if the caller does not pass something in for it. The value the parameter defaults to should be a constant, but can be a global variable. It is not possible to selectively avoid passing certain parameters except for the trailing ones. For that reason, parameters can only have defaults if they are the last in the list *or* the next parameter has a default as well.

You can set parameter defaults in either the prototype or the definition, but you must be consistent. Since the prototype is sometimes known before the definition, it is best to do it there. The following would prototype an 'Add' function whose second parameter has a default of zero (0):

```
int Add(int, int = 0);
```

Bearing this, if the caller were to only pass in one parameter, the second would automatically be initialized to zero.

Call Stack

Barg.

Pointer Parameters

Another way to pass by reference is to use pointer variables rather than reference variables. When you call a function, a new block is entered and all the function parameters are created. This happens just as if you had declared the variables yourself. The things that you pass in are used to initialize the parameters to a value.