Chapter 7: Functions

Notes

- Function prototypes can be local, but function definitions cannot.
- Do not return addresses of local variables as they will/may not be valid after the function is finished.

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()`'. It's like cramming all of your kitchen appliances (if you're not a bachelor and you *have* kitchen appliances) into one monolithic thing that does them all. It works, yes, but it's hardly optimal. Programs get broken up into separate useful actions, like appliances are applied to specific needs. This is known as modularization and part of how this is done in C++ is through *functions*.

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`'[1]. Whereas an operator is like a general verb in English like "run", "eat", "sleep", "barf"; a function is like a specific action "Run 10 Miles", "Eat Ice Cream", "Sleep Forever", "Barf Doggie Bits". The specifics of the action, you code yourself.

A function contains logic, i.e. C++ instructions. So far you have seen all of your C++ instructions in the function 'main'. A function is simply a *disjointed* statement block with a name. This meaning that a function cannot be declared locally, or *nested*, within another block. When the function name is called upon, the code in the associated block is executed. The '`main`' function is special because it is called automatically when the program is run. Executing a function's logic is known as calling or invoking.

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

Functions can have *parameters* which are values that are *passed* to the function as items to be used in the logic. For example a ‘BreadMaker’ function would take (be passed) a pile of dough and turn it into bread. Likewise functions may also have a *return value* which is a value that they pass back (or return) to the caller upon completion of the logic. If you pass dough to ‘BreadMaker’ you expect bread (or some semblance of) to be given back.

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?).

Defining a Function

You have already created one function in your programs all along: ‘main’. Here we’ll explore defining functions other than ‘main’. To define a function you write out its return type, then name, a parameter list and end with the function body:

```
return_type name(parameter_list)
{
    // function body
}
```

For now I will simplify things by using ‘`void`’ for the return type and parameter list:

```
void name(void)
{
    // function body
}
```

This means the function simply performs some logic; it is not passed any values nor does it pass any back. The code of a function is written as statements inside a block. This block is known as the function body.

To cause a function’s code to be executed, it must be *called*. To call a function that does not require parameters you write its name followed by empty parenthesis and ended with a semi-colon. Below is a program with a function called ‘HelloWorld’ that is called from ‘`main`’:

```
#include <iostream.h>

void HelloWorld(void)
{
    cout << “Hello World” << endl;
}

int main()
{
    HelloWorld();
    return 0;
12  }
```

Lines 3 to 6 are the *function definition* of ‘HelloWorld’. A function cannot be called until it has been defined. For a function to be defined it must have an associated block of C++ code known as the *function body*. Lines 4 to 6 are the function body of ‘HelloWorld’. The output of this program is the familiar:

```
Hello World
```

How does this work? Even though ‘HelloWorld’ is defined before ‘main’, it is not executed first. In fact, it will never be executed unless it is called. Think of a ‘BreadMaker’. It is there before you walk in the kitchen, but it does not start making bread until you start it explicitly. The only function that is automatically called is ‘main’.

```
10      HelloWorld();
```

This is the line where the function ‘HelloWorld’ is explicitly called. Execution jumps from the call *into* the function body and executes all instructions there. When the function ends, execution jumps back into ‘main’ just after the point where the function is called. The function that called the current function is known as the *caller*. Thus, when execution is inside ‘HelloWorld’, the caller is ‘main’. Inside ‘main’ there is no caller.

A function can be assumed to always return unless something catastrophic, and unexpected, happens. Execution will return to the point just after where the function was called. Thus, if you called ‘HelloWorld’ twice from ‘main’, both would be executed back to back and in order:

```
HelloWorld();    // called first
HelloWorld();    // called second
```

Blarg.

Prototyping

A function must be defined before it can be used. If we were to define ‘HelloWorld’, from the previous section, *after* ‘main’ then it would not work. The compiler would see ‘HelloWorld’ being used in ‘main’ and wonder what it was. The function ‘HelloWorld’ is not *known* to exist until later, even though it is defined. The solution here is *prototyping*[2].

Function prototyping is a fancy term for *declaring* a function. This declaration acknowledges that the function exists *somewhere* so that it can be used anywhere, regardless of where it is defined; just as long as it *is* defined. This prototype looks much

[2] The C++ Standard requires that all functions (of the sort we will be using in this chapter) be prototyped. Most compilers do not enforce this by default.

like the definition without the function body; and it must end with a semi-colon. The prototype for ‘HelloWorld’ would be:

```
void HelloWorld(void);
```

A function can be used at any point after it has been prototyped and as long as it is defined *somewhere*. If you prototype a function, but do not define it, you will get a lovely *linker* error.

```
01  #include <iostream.h>
02
03  void HelloWorld(void);
04
05  int main()
06  {
07      HelloWorld();
08      return 0;
09  }
10
11  void HelloWorld(void)
12  {
13      cout << “Hello World” << endl;
14  }
```

Here’s another example of a prototyped function that is called from ‘main’. 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 to remember here.

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

This is the function prototype. 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 (both of which are currently 'void'). The compiler will then make sure that the name is linked to an actual block of code.

```
        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 this function causes the execution to jump from the current function ('`main`') into the specified one ('`CountToTen`').

```
void CountToTen(void)
```

This is the beginning of the function definition. It is immediately followed by the function body, the associated statement block. A function definition at this point looks exactly like the prototype but without the semi-colon and with the body.

The body of a function looks like any other statement block. In fact you may notice 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 to the caller after the function has executed. This return value can be used in any expression where that value type is expected. If a function returns an 'int' then you can use the function call as you would any 'int' value. When calling a function that returns a value, the call will represent the function's return. For example, the following program calls a function which returns '5':

```
#include <iostream.h>

int GiveMe5(void);

int main()
{
    cout << "Function gives " << GiveMe5() << endl;
    return 0;
}

```

```
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!). The return type must be specified in both the prototype and the definition.

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:

```
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      int x = 3;
13      return ((x - 1) * 2) + 10 - 9;
14  }
```

This returns the same value ('5'), but it comes from an expression rather than a singular value.

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.[3] 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 returns no value. You cannot get bread from a clock just as you can't get an '`int`' from a '`void`' function.

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. 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`'.

Author's Preference: 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.

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 effect as using the '`break`' statement from within a loop. Execution is broken from that controlled block and returns to the caller:

```
void CountToTen(void)
{
    int i = 1;
    for ( ; ; i++)
    {
        cout << i << endl;
        if (i == 10)
            return;
    }
}
```

The above version of the '`CountToTen`' function does the same thing as the old one, but does it in a different way. When '`i`' gets to '`10`' ('`i == 10`') the '`return;`' statement is executed. This causes the entire function to end and execution to return to the caller.

Correct Main

As per the C++ Standard, a 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:

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

```
int main()
{
    return 0;
}
```

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).

You may also have seen 'main' defined as either of these:

```
void main()
main()
```

Neither of these implementations usually contains a return statement; it is possible, as explained, for the prior one to contain an "empty" return statement. Also the latter can be used with an integer return (such as '`return 0;`') and is sometimes seen as such.

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

A parameter is a named value *passed* to the function as *input*. This value is external to the function; it is copied to the function's parameter so that it can be used within the function body (locally). The value is used in the function liked you would use any other value. I use the term "value" because a parameter can be either a constant or variable. When a function can be passed values as input, it is known to "accept" input. The amount and types of values accepted are determined by the function definition and prototype in what is called the *parameter list*. The parameter list is just that: a list of parameters.

The parameter list is specified to the right of a function name in its definition or prototype. Up until this point I have omitted the parameter list by specifying 'void' (which means the function *accepts* no parameters). A parameter list is one or more variable/constant declarations separated by commas:

```
void ShowBoth(int x, int y)
{
    cout << x << " and " << y << endl;
}
```

The above function has two parameters in its list: 'x' and 'y'. Both are of type 'int'. Notice that these are simply variable declarations without semi-colons and separated by commas, nothing more. The parameters listed are created within the function scope and their life ends when the function exits. For this function to work you would have to *pass* it two 'int' values that could be copied to 'x' and 'y' and used in the function. For example, if you wanted to print '5 and 10' using this function, you'd have to assign 'x' to '5' and 'y' to '10'. Assigning values to parameters is done through *passing values* to the function:

```
ShowBoth(5, 10);
```

The above would pass the values '5' and '10' to the function. The values you *pass in* are assigned in the same order parameters are declared. The first value, '5', is assigned to 'x' and the second, '10', is assigned to 'y'. Commas are used to separate the values passed in, like how parameters are separated.

Passing values to a function is the same as *initializing* a variable or constant. The parameters themselves are just like any other variables or constants and have a scope limited to the function body and lifetime limited to the function execution. When the function is called, the parameters are created and the values you pass in are used to *initialize* them. When the function ends, they are destroyed.

Thus in the above, passing '5' and '10' basically causes the following to happen before the function begins:

```
int x = 5;
int y = 10;
```

Obviously the more parameters listed, the more declarations and initializations are done.

The parameter list inside a prototype must have the same order and types, but names are optional (and possibly ignored). Thus the following are all valid prototypes for '`ShowBoth`':

- `void ShowBoth(int x, int y);`
- `void ShowBoth(int, int y);`
- `void ShowBoth(int x, int);`
- `void ShowBoth(int, int);`

Names are not needed for a function to be declared. A declaration acknowledges existence. You need a certain amount of knowledge about a function for it to be properly prototyped (declared). This knowledge consists only of the name, return type, and parameter types. With this knowledge you can call a function without having defined it because the number and types of the parameters are known as well as what type of value it results in. The actual parameter names are used *within* the function body only and therefore do not need to be known to the rest of the program.

The following program fully illustrates the ‘ShowBoth’ function with prototype, definition, and usage:

```
#include <iostream.h>

void ShowBoth(int,int);

int main()
{
    ShowBoth(5,10);
    return 0;
}

void ShowBoth(int x, int y)
{
    cout << x << “ and ” << y;
}
```

Functional Example

Return values and parameters are used to better glue functions, separate logic pieces, together. Without them you’d have to declare variables globally to use functions. Consider the following program using a “void function” (techno-babble for function that has a ‘`void`’ return type):

```
#include <iostream.h>

void Square(void);

int square_val = 0;
int square_result = 0;

int main()
{
    square_val = 2;
    Square();
    cout << “The square of 2 is ” << square_result
         << endl;
    return 0;
}

void Square(void)
{
    square_result = square_val * square_val;
}
```

This can be done differently if the ‘Square’ function returned a value and accepted a parameter for the value to square:

```
#include <iostream.h>

int Square(int);

int main()
{
    cout << “The square of 2 is ‟ << Square(2)
         << endl;
    return 0;
}

int Square(int val)
{
    return val * val;
}
```

Passing By Value

Conceptually passing by value means to give a function input that it can use, but not effect externally. This is all I’ve shown you this far. The previous examples pass values to functions that use them, but those values are not altered. Let’s look at a function where parameters are passed by value:

```
void Apple(int x)
{
    x = 5;
}
```

Examining the definition above, the temptation might be to use this function as so:

```
int y = 0;
Apple(y);
```

But what would the value of ‘`y`’ be after the function returns? It would still be zero (0). This is because we *passed by value* or in other words we *initialized the parameter with a value*; passing anything really means to initialize some function parameter. In the case of ‘`Apple`’ we initialize parameter ‘`x`’ to the variable ‘`y`’:

```
int x = y;
```

Since ‘`x`’ is merely initialized to the *value* of ‘`y`’, it cannot affect it in any way, shape, or form. It simply now has the same value. The value of ‘y’ is *copied* into ‘x’ and anything done to ‘x’ affects this copied value:

<value being copied to ‘x’ from ‘y’>

Passing by value means that whatever is passed to that parameter (whatever the parameter is initialized to) will not (or cannot) be changed by the function. This function '`Apple`' is totally useless because it tries to assign a value to its parameter '`x`', a local variable, which is destroyed when the function ends.

But it isn't all bad. Passing by value means you can pass any expression as long as it results in the same type as the parameter being initialized. For example:

```
#include <iostream.h>

void Igloos(int);

int main()
{
    int x = 2;
    Igloos(x);
    Igloos(x + 2);
    int y = 4;
    Igloos(x * y – 1);
    Igloos(x * (y – 1));
    return 0;
}

void Igloos(int i)
{
    cout << i << " igloos!" << endl;
}
```

The output of this is:

```
2 igloos!
4 igloos!
7 igloos!
6 igloos!
```

Rather than just passing in some literal value, I've passed in several different expressions. Each time I call 'Igloos' the expression is evaluated and its result is used to initialize the parameter 'i'. For example:

```
        Igloos(x * (y – 1));
```

This is the most complex line in the program. It would basically break down to[4]:

```
int i = x * (y - 1);
int i = x * 3;
int i = 6;
```

[4] This example break-down is meant to be easy to understand. What usually happens is the expression is evaluated and the result is stored in an intermediate value. That value is then used to initialize the parameter when the function is executed.

Take note that parameters are the only bridge between local variables of two different functions. It is not possible to use the variables ‘x’ and ‘y’ inside ‘Igloos’ nor would it be possible to use ‘i’ within ‘main’; but through parameters these values can be communicated.

The names of parameters, ye any local variables, can be the same between two different functions. No name-clashing occurs because the variables are declared in disjointed statement blocks (i.e. separate islands) and have no interaction with one another.

Passing By Reference

An alternative to passing by value is passing by reference. This is done by declaring parameters are as references rather than normal variables. References are variable “nicknames” in that they have no value of their own; they *refer* to another variable and its value. A parameter is made a reference by preceding the parameter identifier (name) with an ampersand (&). This specifies that the parameter should be passed a *variable* of the same data type. Both the definition *and* prototype require the ampersand because it is part of the parameter’s type, not its name. Thus, to prototype a function with a single ‘int’ reference parameter:

```
void Oranges(int &o);
```

In the definition the parameter must be named, but it is optional in the prototype. Recall that references must *always* be initialized when they are created. The initializer for a reference must be a variable of the same data type as the reference. It works the same with reference parameters and it works out quite nicely seeing as parameters are created *and initialized* when the function is called. So, in calling the above function with an integer variable, the reference parameter is initialized properly:

```
int x = 5;
Oranges(x);
```

The reference parameter ‘o’ is then created and initialized as an *alias* to ‘x’. Anything done to ‘o’ will now affect ‘x’. In essence it is the same as this:

```
int &o = x;
```

What does this mean? Now the value of ‘x’ can be affected in the function body, simply by changing the value of ‘o’. Observe:

```
#include <iostream.h>

void Oranges(int&);

int main()
{
    int x = 2;
    Oranges(x);
    cout << “x = “ << x << endl;
    return 0;
}

void Oranges(int &o)
{
    o *= 2;
}
```

This program expands on our ‘Oranges’ by defining an actual function. This function takes a single reference parameter of ‘int’ data type. It multiplies the parameter by two, effectively multiplying the original variable, and then ends. The output for this program is:

```
x = 4
```

When the function is called, the reference parameter, ‘o’, is initialized to the variable ‘x’. Thus it now refers to the variable ‘x’. The parameter ‘o’ has become a reference, or alias, to the variable we passed in. Anything done to ‘o’ will effect ‘x’.

Passing By Psuedo-Reference

When using pointer variables are parameters, it is possible to pass by reference as well. This is sometimes considered a pseudo-reference because you are still passing a value; you are passing a memory address. A pointer parameter is declared in the same way as a pointer, by preceding the name with an asterisk:

```
void Grapes(int *p);
```

The above function accepts the address of an ‘int’ variable. To use this function a variable name cannot simply be used; it must be passed the *address* of a variable. To get the address of a variable you precede it with an ampersand, the address operator:

```
int x = 2;
Grapes(&x);
```

This would get the address of the variable ‘x’ and use that to initialize the function’s pointer parameter. Thus, it has the same effect as:

```
int *p = &x;
```

Now the pointer variable can be used inside the function to affect the variable whose address it holds. To affect the original variable a pointer must be dereferenced. Anything done to the dereferenced pointer will change the variable whose address is being pointed to. If the purpose of the ‘Grapes’ function was to multiply the original variable by ‘16’, it might look like this:

```
void Grapes(int *p)
{
    (*p) *= 16;
}
```

The pointer-indirection operator, or asterisk, is used to dereference the pointer. This has the effect of temporarily turning the pointer into the variable whose address it stores. Once the pointer has been dereferenced, anything done to it will modify the original variable:

```
int x = 2;
Grapes(&x);
```

Considering the 'Grapes' function above, 'x' will be '32' (2 * 16) after the above code is executed.

Not only can you pass the addresses of variables to pointer parameters, but you can use a certain literal as well. The pointer value of zero (0), or *NULL*, is used to signify a pointer that contains no valid address. That is, if a pointer is null it does not point to anything (it points to null or nothing). Public functions that accept pointers, usually check them for NULL before using them. Since NULL is zero (0), you can use a simple 'if' on a pointer:

```
if (!p)
{
    // 'p' is NULL
}
```

With this knowledge we might change 'Grapes' to look like so:

```
void Grapes(int *p)
{
    if (!p)
        return;
    (*p) *= 16;
}
```

Now the function 'Grapes' will simply return (end and exit the function) if the parameter 'p' is NULL (invalid). The C++ language does not define "NULL" as a keyword, but you may use an all capitalized version which is part of the standard library. Thus, we could call 'Grapes' with NULL by either of the following:

```
Grapes(0);
Grapes(NULL);
```

Author's Preference: Nowadays I use just zero (0) instead of NULL. I'm lazy and I know it!

Constant Parameters

Constants can be parameters as well. Declaring a constant is much like the declaration of a variable, but you precede the entire statement with the keyword ‘const’. This creates a named-value that can never be altered. It will act like a variable in that it can be used as part of expressions, but *its value will never ever change*:

```
void Pears(const int x)
{
    x = 5; // will not work
}
```

Constants, like reference variables, must be initialized when they are created. Again, this works out fine with functions because parameters are created when the function is called and initialized to the values passed in.

A constant parameter can be passed a variable or literal of the same type. That is, if ‘Pears’ above was valid and accepted a single ‘int’ parameter ‘x’, you could use it like so:

```
int y = 10;
Pears(y);
Pears(6);
Pears(y + 5);
```

The opposite is partially true as well. If you have a function that accepts a variable, you can pass it a constant. Take the following function for example:

```
void VariableFruits(int v)
{
    cout << “You have ‟ << v << ‟ fruits!” << endl;
}
```

We can use this function like so:

```
int x = 5;
const int c = 18;
VariableFruits(c);
VariableFruits(x);
VariableFruits(7);
VariableFruits(x + 10);
```

Those all work fine. However, you cannot pass a constant reference or pointer to a function accepting a variable without *explicit casting*. That is, you must force the compiler to allow you to pass a constant value to something accepting a variable. The reason is simple: theoretically a constant cannot be modified and getting a *variable pointer* or *reference* to a constant value makes this rule debunk. Once you have a variable pointer or reference, you can change the original value. Consider the following function which accepts a variable pointer:

```
void Nosey(int *p)
```

```
{
    *p = 5;
}
```

This function changes the value of the variable pointed to by ‘p’ to five (5). You would not be allowed to call this function with a constant ‘int’ or literal because it modifies the value passed in:

```
const int x = 10;
Nosey(&x);  // won’t work (tries to pass ‘const int *’ value)
```

You can, maybe unfortunately, bend the rules by cleverly casting a constant so that it becomes a variable:

```
const int x = 10;
Nosey(const_cast<int*>(&x));
```

Calling ‘const_cast’ causes the expression result of ‘&x’ to become the type ‘int *’ or an ‘int’ pointer. This then can be used in the call to ‘Nosey’ which then changes the value of the variable passed in. Now the constant ‘x’ is no longer ten (10), but five (5). Without the explicit cast (using ‘const_cast’) the call would be illegal because the result of ‘&’ on a constant is a constant memory address or constant pointer.

A function that accepts a reference, on the other hand, cannot be deceived to allow a constant by cleverly casting it:

```
void NotNosey(int &ref)
{
    ref = 5;
}
```

There is no way to directly use this above function with a constant ‘int’.

In terms of references and pointers, passing constants to variable parameters is illegal, but passing variables to constant parameters is not. A constant parameter cannot be modified and thus it does not expose unruly access; rather it prevents it. This becomes useful when you use larger types like structures.

Using Structures with Functions

Passing structures to functions causes some degree of confusion. A structure is a blob of memory with pieces partitioned off to separate member variables. When you pass a structure to a function, the structure parameter is created and its contents are initialized with the structure variable you passed in. The issue here is *over-head*, or the amount of memory and/or CPU power being wasted. Each time a structure variable is passed to a function, a new structure must be created and its contents initialized:

<example>

Rather than pass a structure by value, you can pass it by reference or pseudo-reference as well: just like any variable. This is faster because the only thing being passed to the function is the memory address of the structure variable. However it is more dangerous because if you change any of the members in the structure passed by reference, the original structure is modified as well.

<example>

If you want a function to accept a reference or pointer to a structure, but not let it be modified, you can mark the parameter as constant as well. It is possible to have a reference constant as it is a reference variable. The entire structure is now protected from modification.

<example>

To be done: returning structures, returning pointers to structures, etc.

Defaults

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 that value is used to initialize that parameter if the caller does not pass something in for it. The value the parameter defaults to can be a literal, constant, or 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 usually 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. If the caller passed in a second value, it would supercede the default.

To be done: elaborate, more examples, why you can't have selective defaults, etc.

Three I/O Flavors

There are three flavors of input and output with functions. The first and the least complicated is the return value which is always output. You cannot leverage it in anyway for function input. Then there are parameters. These can be used just for input as when passing by value or for both input and output as when passing by reference. Parameters that are known to be used for output are called *output parameters*. Whether or not a parameter is actually *used* for output is up to the function designer. When documenting a function, it is a good idea to mention if parameters are used for input, output, or both. In some cases parameters are only used for output as their original value is not used at all. The following function would be an example of such:

```
void Set(int &dest, int src)
{
    dest = src;
}
```

All the above function does is set the reference parameter '`dest`' to the value of normal parameter '`src`'. In this case '`dest`' is an output parameter because its original value is not used in anyway; it is always over-written.

To recap: Parameters can be used for both input and output. Passing by value is used to give the function input, while passing by reference can do that as well as provide a place for output. Passing a memory address (by value) can also provide a place for output. A function's return value can, obviously, only be used for output.

When a function needs to output more than one value, you can either return a structure or use multiple output parameters. Whether or not a parameter is used for input, output, or both is up to the function designer and should be documented appropriately.

Constant parameters can only be used for input even if they are references or pointers.[5]

Function Scope

Parameters and variables that are declared within a function have *function scope*. That is, they are known only within the function they were declared. Additionally their lifetime is limited to the function call itself. When the function returns, or ends, any variables created, including parameters, are destroyed.

You cannot use variables from one function in another. Nor will the variables become ambiguous if they are named the same thing, but are in different functions. If you declare 'x' in 'main', you cannot use it in another function. This is the purpose of parameters: to provide a bridge between the data in the caller (function calling) and the logic in the function being called.

[5] It *is* possible to circumvent this through explicit casting.

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 all other functions as islands. Each function parameter and return result is a bridge with a road that is either two-way, one-way from the island, or one-way to the island. Thus each bridge represents a bridge for data input, output, or both. Only what you pass to the island through the bridges can be used there. All variables are crates and values are bananas stored in crates.

If you want to use any of the bananas from the continent ('main()') you need to bring them over one of the bridges, but it has to be one leading to the island. Thus you could not take the return bridge because it only goes one way: away from the island. Likewise with functions; to use a variable from 'main()' it or its value must be passed to one of the function's parameters.

When you pass by value, you carry bananas from a crate on the continent to a new crate on the island. When you pass by reference you actually carry the crate from the continent to the island and then back again.

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. It is an island without any bridges and must be self-reliant.

Global variables can be accessed by any function in a program. Think of a crate hanging from a helicopter in our island analogy … it doesn't need to use the bridges to be used. It can be flown to any island to collect or drop off bananas. Just like a global variable that can be used anywhere in your program.

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!).

Static Locals

A static local is a constant or variable that has the scope of a local variable, but the lifetime of a global variable. A normal local is created when it is declared and destroyed when it goes out of scope (i.e. the statement block it was declared in ends). A static local is created when it is first run across and remains until the program ends; much like a global variable. This kind of value, however, has the same scope as a local variable: it can be used locally, but not globally.

Now that you're thoroughly confused, I will elaborate to clear the mists (excuse me). A static local in the island analogy would be a crate that always exists on an island. It exists until the island itself is destroyed; when your program ends.

To declare a local variable as static, precede its declaration with the keyword '`static`'. This keyword is known as a *storage-class modifier* because it modifies the storage class of the variable being declared. The storage is being altered because it will exist for the length of the entire program rather than just the current statement block. Consider the following program:

<counter function program>

Each time the function '`Counter`' is called, the local variable '`count`' is incremented and printed. Notice that, although it is initialized to zero, its count continues to increase. This is due to the '`static`' storage-class modifier its declaration statement. A local variable is one that is declared within a statement block; thus any variable declared within any statement block (even nested ones) can be static.

Parameters can *not* be static because they are always created when a function is called for the purpose of bridging data. A static parameter would not bridge data because a static is *only initialized once*.

Parameter Terminology Argument

There are several ways to phrase the usage of parameters and functions. I have used "pass", "invoke", and "parameter" quite frequently. But these are not the only words you will hear referring to the concepts explained in this chapter.

There are some programmers that insist the values passed to functions are parameters while the declaration of function I/O variables (that I have referred to thus far as parameters) are known as *arguments*. As far as I can tell, there is no correct answer: it is a matter of opinion (and *style*?). The usage of "argument" is practically synonymous with my usage of "parameter". However, as mentioned, "parameter" is sometimes used to describe the *value* being passed to a function. For example:

```
int x;
Oranges(x);
Oranges(10);
```

The parameters in these cases is 'x' on the second line and '10' on the third.

Author's Opinion: As long as you know what is happening in this code, the terminology is irrelevant.

Standard Functions

From now on I expect you'll be able to call functions you've created and that already exist for you as part of the standard library. For instance, if you add '#include <math.h>'

below ‘#include <iostream.h>’, you can use one or more of the many standard math functions. However, before showing the usage of a function, I will explain it briefly including the parameters it accepts and its return value (if any).

The ‘`sqrt`’ function has one ‘double’ parameter and returns a ‘double’ value. It calculates the square root of the parameter and returns the result. If you called ‘`sqrt(9)`’ it would result in three (3):

```
#include <iostream.h>
#include <math.h>

int main()
{
    double n;
    cout << “Input a number: “;
    cin >> n;
    cout << “The square root of “ << n << “ is “
         << sqrt(n) << endl;
    return 0;
}
```

Here is some sample output from the above program:

```
Input a number: 9
The square root of 9 is 3
```

There were a few new things in this program that you haven’t seen before. Obviously they were the ‘`#include <math.h>`’ and the line using ‘`sqrt()`’. When you use ‘`#include`’[6] with greater than/less than signs (< >), you are including collections of *declarations*, and sometimes definitions, of constants, variables, and yes: functions. I won’t go into depth on these things here, but including one of the following files gives you the ability to use various constants, variables, and functions:

<list of standard C++ header files>

These are standard files that all compilers should have. Some also have ‘`conio.h`’ which contains easy cheesy “Console I/O” functions. In particular, compilers with ‘`conio.h`’ give the programmer a simple way to get input without requiring the user to press enter, position the cursor in the console, and even usually to change the text color. These things you can do on any compiler, of course, but ‘`conio.h`’ has functions to make it simpler. If you do not have this file, do not despair. I will eventually cover how to do these things anyway, with or without that file.

The ‘`sqrt()`’ function[7] can be used if you ‘`#include`’ the file ‘`math.h`’. As you can see, the names of most files will summarize their contents. This function is

[6] Known in speaking terms as “pound include” because ‘#’ is the pound sign.
[7] When denoting functions, I usually follow the name with empty parenthesis to distinguish them from other identifiers (like constants and variables).

prototyped in '`math.h`', but where is the function body? The function definition of this and many more are inside pre-compiled *library* files. The logic inside these compiled libraries can be referenced using the declarations in the standard files I just listed. This means that you don't have to define '`sqrt()`' or prototype it; both have already been done for you. All you have to do is *use* it; and other standard functions like it.

If you haven't guessed, '`cout`' and '`cin`' are declared somewhere deep inside '`iostream.h`'. They are not functions or normal variables: they are object variables. These kinds of variables possess both data *and* logic and are an advanced topic you will find in Part 2 of this book.