**Day 5: Organizing into Functions**

**What Is a Function?**

**Function** - is, in effect, a **subprogram** that can act on **data** and **return a value**. Because these functions are not part of an object, they are called ***“global”***—that is, they can be accessed from **anywhere** in your program. Functions can **receive** values and can return a value.

***Calling*** **Function** - Each function has its own **name**, and when that name is **encountered**, the execution of the program **branches** to the **body** of that function.

*“****Well****-****designed*** *functions* ***perform*** *a* ***single****,* ***specific****, and* ***easily******understood******task****, identified by the function* ***name****.* ***Complicated******tasks*** *should be broken down into* ***multiple*** *functions, and then each can be* ***called******in******turn****.”*

***Functions come in two varieties:***

* ***Built-in*** **functions** - are part of your **compiler** **package**—they are supplied by the manufacturer for your use.
* ***User***-***defined*** **functions** - are the functions you **write** **yourself**.

**Return Values, Parameters, and Arguments**

***Return Value*** - When you **call** a function, it can do work and then **send** **back** a value as a **result** of that work and the **type** of that **return** value must be **declared**.

**int** ***myFunction*** (**int** ***someValue***, **float** ***someFloat***)**;**

This declaration indicates that myFunction will **still** return an integer, but it will also take two values.

**Parameter List -** When you **send** values into a function, these values **act** as variables that you can **manipulate** from within the function. Parameter describes the ***type*** of the value that will be passed into the function when the function is **called**. The parameter list is a list of all the parameters and their ***types***, separated by **commas**.

***Syntax for Parameter List:***

**Int** ***SampleFunction*** (**int** ***SampleValue*** , **int** ***AnotherSampleValue***)**;**

**Arguments -** The actual values you pass into the function.

***Syntax for Arguments:***

***Int CallingTheFunction = SampleFunction (5 , 10*** //ARGUMENTVALUES***);***

***“The type of the arguments must match the declared parameter types.”***

**Declaring and Defining Functions**

***“Using functions in your program requires that you first declare the function and that you then define the function. No function can be called from any other function if it hasn’t first been declared.”***

**Function Declaration (*Prototype*)** - The declaration tells the compiler the **name**, **return** ***type***, and **parameters** of the function.

**Function Definition** (***Defining***) - The definition tells the compiler **how** the function **works**.

**Function****Prototype** *(****also******Declaration****) -* is a statement, which means it **ends** with a **semicolon**. It consists of the function’s ***return*** **type** and function **signature** or **name** and **parameter** **list**.

***Two ways exist to declare a function:***

• Write your prototype into a file, and then use the #include directive to include it in your program.

• Write the prototype into the file in which your function is used.

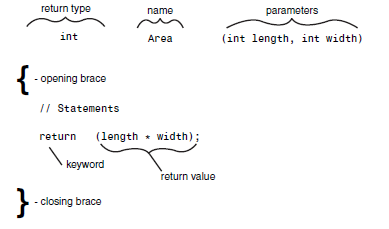
***“The function prototype and the function definition must agree exactly about the return type and signature. If they do not agree, you receive a compile-time error. Note, however, that the function prototype does not need to contain the names of the parameters, just their types.”***

* Adding parameter names makes your prototype clearer. The same function with named parameters might be clearer to understand.
* Note that all functions have a return type. If none is explicitly stated, the return type defaults to int. Your programs will be easier to understand, however, if you explicitly declare the return type of every function.

***Void*** - If your function does **not** actually return a value, you declare its return type to be ***void***. Because void is used as the return time, **nothing** is **returned**.

**Defining the Function**

***Syntax of the Definition of the Function:***

* The definition of a function consists of the function **header** and its **body**. The header is **like** the function prototype **except** that the parameters **must** be named, and **no** terminating semicolon is used.
* The body of the function is a **set** of statements enclosed in braces.

**Execution of Functions**

***“When you call a function, execution begins with the first statement after the opening brace ({). Branching can also be accomplished by using the if statement. When a function is done executing, control is returned to the calling function.***

***When the main() function finishes, control is returned to the operating system.”***

**Determining Variable Scope**

**Local Variables -** Variables you declare within the body of the function are called “local” because they exist only locally within the function itself. When the function returns, the local variables are no longer available; they are marked for destruction by the compiler. Local variables are defined the same as any other variables. The parameters passed in to the function are also considered local variables and can be used exactly as if they had been defined within the body of the function.

**Local Variables Within Blocks**

*“You can define variables anywhere within the function, not just at its top. The scope of the variable is the block in which it is defined. Thus, if you define a variable inside a set of braces within the function, that variable is available only within that block.”*

**Parameters Are Local Variables**

*“The arguments passed in to the function are local to the function. Changes made to the arguments do not affect the values in the calling function. This is known as passing by value, which means a local copy of each argument is made in the function. These local copies are treated the same as any other local variables.”*

**Global Variables**

**Global** **Variables** - Variables defined outside of any function have global scope, and thus are available from any function in the program, including main().

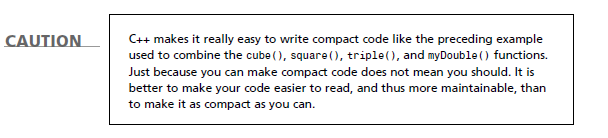
* Local variables with the same name as global variables do not change the global variables. A local variable with the same name as a global variable hides the global variable, however. If a function has a variable with the same name as a global variable, the name refers to the local variable—not the global—when used within the function.
* In C++, global variables are legal, but they are almost never used. C++ grew out of C, and in C global variables are a dangerous but necessary tool.

**Considerations for Creating Function**

**Statements**

* Virtually no limit exists to the number or types of statements that can be placed in the body of a function. Although you can’t define another function from within a function, you can call a function, and of course, main() does just that in nearly every C++ program.
* Many programmers advise keeping your functions short enough to fit on a single screen so that you can see the entire function at one time.
* Each function should carry out a single, easily understood task. If your functions start getting large, look for places where you can divide them into component tasks.

**More About Function Arguments**

* Any valid C++ expression can be a function argument, including constants, mathematical and logical expressions, and other functions that return a value. The important thing is that the result of the expression matches the argument type that is expected by the function.
* It is even valid for a function to be passed as an argument. After all, the function will evaluate to its return type. Using a function as an argument, however, can make for code that is hard to read and hard to debug.

**More About Return Values**

***Return* Values (again)** - Functions return a value or return void. Void is a signal to the compiler that no value will be returned.

* To return a value from a function, write the keyword *return* followed by the value you want to return. The value might itself be an expression that returns a value.
* When the return keyword is encountered, the expression following return is returned as the value of the function. Program execution returns immediately to the calling function, and any statements following the return are not executed.
* It is legal to have more than one return statement in a single function.

**Default Parameters**

**Default Parameter** - if the function prototype declares a default value for the parameter. A default value is a value to use if none is supplied. The preceding declaration could be rewritten as: long myFunction (int x = 50);

The function definition is not changed by declaring a default parameter. The function definition header for this function would be:  
 long myFunction (int x)

If the calling function did not include a parameter, the compiler would fill x with the default value of 50. The name of the default parameter in the prototype need not be the same as the name in the function header;

Restriction:

If any of the parameters does not have a default value, no previous parameter can have a default value.

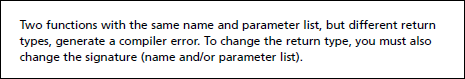
If the function prototype looks like

long myFunction (int Param1, int Param2, int Param3);

you can assign a default value to Param2 only if you have assigned a default value to Param3. You can assign a default value to Param1 only if you’ve assigned default values to both Param2 and Param3.

**Overloading Functions**

**Function** **Overloading** (***function*** ***polymorphism***) - enables you to create more than one function with the same name. The functions must differ in their parameter list with a different type of parameter, a different number of parameters, or both. The return types can be the same or different on overloaded functions. Example:

int myFunction (int, int);  
int myFunction (long, long);  
int myFunction (long);

*“Function overloading is also called function polymorphism. Poly means many, and morph means form: A polymorphic function is many-formed.”*

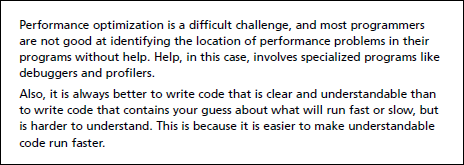
*Function polymorphism refers to the capability to “overload” a function with more than one meaning. By changing the number or type of the parameters, you can give two or more functions the same function name, and the right one will be called automatically by matching the parameters used. This enables you to create a function that can average integers, doubles, and other values without having to create individual names for each function, such as AverageInts(), AverageDoubles(), and so on. With function overloading, you make this declaration:*

*int Double(int);  
long Double(long);  
float Double(float);  
double Double(double);*

**Inline Functions**

* When you define a function, normally the compiler creates just one set of instructions in memory.
* If you call the function 10 times, your program jumps to the same set of instructions each time. This means only one copy of the function exists, not 10.

***“When programmers speak of efficiency, they usually mean speed; the program runs faster if the function call can be avoided.”***

***Inline*** - If a function is declared with the keyword inline, the compiler does not create a real function; it copies the code from the inline function directly into the calling function. No jump is made; it is just as if you had written the statements of the function right into the calling function.

**Recursion**

**Recursion** - A function can call itself. Recursive functions need a stop condition. Something must happen to cause the program to stop recursing, or it will never end. Like (n < 3), it is a stop condition (that is, when n is less than 3 the program can stop working on the problem).

**Direct** **Recursion** - when a function calls itself.

**Indirect** **Recursion** - when a function calls another function that then calls the first function.

*“It is important to note that when a function calls itself, a new copy of that function is run.”*

*“The local variables in the second version are independent of the local variables in the first, and they cannot affect one another directly, any more than the local variables in main() can affect the local variables in any function it calls”*

*“Recursion is not used often in C++ programming, but it can be a powerful and elegant tool for certain needs.”*

***“Recursion is a tricky part of advanced programming. It is presented because it can be useful to understand the fundamentals of how it works, but don’t worry too much if you don’t fully understand all the details.”***

**Levels of Abstraction**

One of the principal hurdles for new programmers is grappling with the many layers of intellectual abstraction. Computers, of course, are only electronic machines. They don’t know about windows and menus, they don’t know about programs or instructions, and they don’t even know about ones and zeros. All that is really going on is that voltage is being measured at various places on an integrated circuit. Even this is an abstraction:

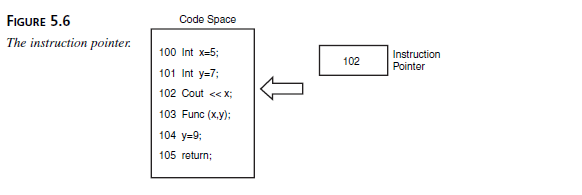
Electricity itself is just an intellectual concept representing the behavior of subatomic particles, which arguably are themselves intellectual abstractions(!). Few programmers bother with any level of detail below the idea of values in RAM. After all, you don’t need to understand particle physics to drive a car, make toast, or hit a baseball, and you don’t need to understand the electronics of a computer to program one.

You do need to understand how memory is organized, however. Without a reasonably strong mental picture of where your variables are when they are created and how values are passed among functions, it will all remain an unmanageable mystery.

Partitioning RAM

When you begin your program, your operating system (such as DOS, Linux/Unix, or Microsoft Windows) sets up various areas of memory based on the requirements of your compiler. As a C++ programmer, you’ll often be concerned with the global namespace, the free store, the registers, the code space, and the stack. Global variables are in global namespace.

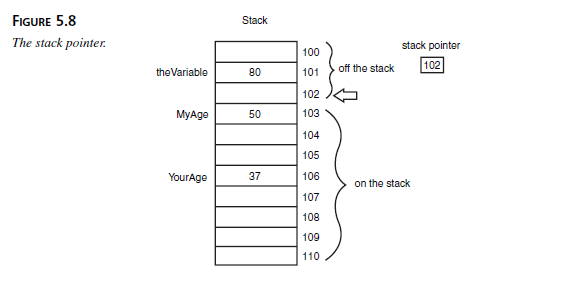
Registers are a special area of memory built right into the central processing unit (or CPU). They take care of internal housekeeping. A lot of what goes on in the registers is beyond the scope of this book, but what you should be concerned with is the set of registers responsible for pointing, at any given moment, to the next line of code. These registers, together, can be called the instruction pointer. It is the job of the instruction pointer to keep track of which line of code is to be executed next.

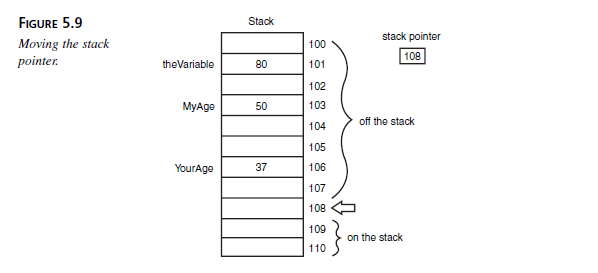
The code itself is in the code space, which is that part of memory set aside to hold the binary form of the instructions you created in your program. Each line of source code is translated into a series of instructions, and each of these instructions is at a particular address in memory. The instruction pointer has the address of the next instruction to execute.

The stack is a special area of memory allocated for your program to hold the data required by each of the functions in your program. It is called a stack because it is a last-in, first-out queue, much like a stack of dishes at a cafeteria. Last-in, first-out means that whatever is added to the stack last is the first thing taken off. This differs from most queues in which the first in is the first out (like a line at a theater: The first one in line is the first one off). A stack is more like a stack of coins: If you stack 10 pennies on a tabletop and then take some back, the last three you put on top are the first three you take off.

When data is pushed onto the stack, the stack grows; as data is popped off the stack, the stack shrinks. It isn’t possible to pop a dish off the stack without first popping off all the dishes placed on after that dish.

A stack of dishes is the common analogy. It is fine as far as it goes, but it is wrong in a fundamental way. A more accurate mental picture is of a series of cubbyholes aligned top to bottom. The top of the stack is whatever cubby the stack pointer (which is another register) happens to be pointing to.

Each of the cubbies has a sequential address, and one of those addresses is kept in the stack pointer register. Everything below that magic address, known as the top of the stack, is considered to be on the stack. Everything above the top of the stack is considered to be off the stack and invalid.

When data is put on the stack, it is placed into a cubby above the stack pointer, and then the stack pointer is moved to the new data. When data is popped off the stack, all that really happens is that the address of the stack pointer is changed by moving it down the stack. The data above the stack pointer (off the stack) might or might not be changed at any time. These values are referred to as “garbage” because their value is no longer reliable.

**The Stack and Functions**

The following is an approximation of what happens when your program branches to a function. (The details will differ depending on the operating system and compiler.)

1. The address in the instruction pointer is incremented to the next instruction past the function call. That address is then placed on the stack, and it will be the return address when the function returns.
2. Room is made on the stack for the return type you’ve declared. On a system with two-byte integers, if the return type is declared to be int, another two bytes are added to the stack, but no value is placed in these bytes (that means that whatever “garbage” was in those two bytes remains until the local variable is initialized).
3. The address of the called function, which is kept in a special area of memory set aside for that purpose, is loaded into the instruction pointer, so the next instruction executed will be in the called function.
4. The current top of the stack is now noted and is held in a special pointer called the stack frame. Everything added to the stack from now until the function returns will be considered “local” to the function.
5. All the arguments to the function are placed on the stack.
6. The instruction now in the instruction pointer is executed, thus executing the first instruction in the function.
7. Local variables are pushed onto the stack as they are defined.

When the function is ready to return, the return value is placed in the area of the stack reserved at step 2. The stack is then popped all the way up to the stack frame pointer, which effectively throws away all the local variables and the arguments to the function.

The return value is popped off the stack and assigned as the value of the function call itself, and the address stashed away in step 1 is retrieved and put into the instruction pointer. The program thus resumes immediately after the function call, with the value of the function retrieved.

Some of the details of this process change from compiler to compiler, or between computer operating system or processors, but the essential ideas are consistent across environments.

***“In general, when you call a function, the return address and the parameters are put on the stack. During the life of the function, local variables are added to the stack. When the function returns, these are all removed by popping the stack.”***