**C++ FUNCTIONS – related concepts to review….**

**MODULAR PROGRAMMING**: A program may be broken up into a setoff manageable functions, or modules. This is often called modular programming.

**DEFINING AND CALLING FUNCTIONS**: A function call is a statement that causes a function to execute. A function definition contains the statements that make up the function. All function definitions have the following parts: *Name, Parameter list, body* and *return type*.

**FUNCTION PROTOTYPES**: A function prototype eliminates the need to place a function definition before all calls to the function. Function prototypes are also known as *function declarations*.

**SENDING DATA INTO A FUNCTION**: When a function is called, the program may send values into the function. The values that are passed into a function are called arguments, and the variables that receive those values are called parameters. Like all variables parameters have a scope. The scope of a parameter is limited to the body of the functions which it uses.

**PASSING DATA BY VALUE**: When an argument is passed into a parameter by value, only a copy of the argument’s value is passed. Changes to the parameter do not affect the original argument.

**THE RETURN STATEMENT**: The return statement causes a function to end immediately.

**RETURNING A VALUE FROM A FUNCTION**: A function may send a value back to the part of the program that called the function. In order to return multiple values from a function, they must be “packaged” in such a way that they are treated as a single value.

**RETURNING A BOOLEAN VALUE**: Functions may return *true* or *false* values.

**LOCAL AND GLOBAL VARIABLES**: A local variable is defined inside a function and is not accessible outside the function. A global variable is defined outside all functions and is accessible to all functions in its scope.

**LOCAL VARIABLES**: Variables defined inside a function are local to that function. They are hidden from the statements in other functions, which normally cannot access them. Furthermore a local variable exists only while the function it is defined in is executing. This is known as the lifetime of a local variable. When the function begins, its parameter variables and any local variables it defines are created in memory, and when the function ends, they are destroyed. This means that any values stored in a function’s parameters or local variables are lost between calls to the function. Remember that local variables are not automatically initialized as global variables are. The programmer must handle this.

**GLOBAL VARIABLES**: A global variable is any variable defined outside all the functions in a program, including main. The scope of global variables is the portion of the program from the variable definition to the end of the entire program. This means that a global variable can be accessed by all functions that are defined after the global variable is defined. In C++, unless you explicitly initialize global variables, they are automatically initialized to zero. Global character variables are initialized to NULL. Although global variables can be useful, you should restrict your use of them.

**GLOBAL CONSTANTS**: If you wish to share global information among modules, you should declare a global constant. A *global constant*, defined with the key word const, is a read only variable that is visible to all modules, but that cannot be changed by any of them. Therefore modules do not have to worry about who else may be using them.

**LOCAL AND GLOBAL VARIABLES WITH THE SAME NAME**: You cannot have two local variables with the same name in the same function. This applies to parameter variables as well. A parameter variable is, in essence, a local variable. So, you cannot give a parameter variable and a local variable in the same function the same name. However, you can have a parameter or local variable with the same name as a global variable or constant. When you do this, the name of the parameter or local variable *shadows* (aka. name hiding) the name of the global variable or constant. This means that the global variable or constant’s name is hidden by the name of the parameter or local variable. So, the global variable or constant can’t be seen or used in this part of the program.

**STATIC LOCAL VARIABLES**: If a function is called more than once in a program, the values stored in the function’s local variables do not persist between function calls. This is because local variables are destroyed when a function terminates and are then recreated when the function starts again. Sometimes, however, it’s desirable for a program to “remember” what value is stored in a local variable between function calls. This can be accomplished by making the variable static. Static local variables are not destroyed when a function returns. They exist for the entire lifetime of the program, even though their scope is only the function in which they are defined.

**DEFAULT ARGUMENTS**: Default arguments are passed to parameters automatically if no argument is provided in the function call. A function’s default arguments should be assigned in the earliest occurrence of the function name. This will usually be the function prototype. However, if a function does not have a prototype, default arguments may be specified in the function header. Although C++’s default arguments are very convenient, they are not totally flexible in their use. When an argument is left out of a function call, all arguments that come after it must be left out as well. It is possible, however, for a function to have some parameters with default arguments and some without.

**USING REFERENCE VARIABLES AS PARAMETERS**: A reference variable is a variable that references the memory location of another variable. Any change made to the reference variable is actually made to the one it references. Reference variables are sometimes used as function parameters. A reference variable is an alias for another variable ie., instead of having its own memory location for storing data, it accesses the memory location of another variable. When data is passed to a parameter in this manner, the argument is said to be *passed by reference.* Reference variables are defined like regular variables, except there is an ampersand (&) between the data type and the name. The ampersand must appear in both the prototype and the header of any function that uses a reference variable as a parameter. It does not appear in the function call. Only variables may be passed by reference. If you attempt to pass a non-variable argument, such as a literal, a constant, or an expression, into a reference parameter, an error will result.

**WHEN TO PASS ARGUMENTS BY REFERENCE AND WHEN TO PASS BY ARGUMENTS BY VALUE**: Here are some general guidelines:

* When an argument is a constant, it must be passed by value. Only variables can be passed by reference
* When a variable passed as an argument should not have its value changed, it should be passed by value. This protects it from being altered.
* When exactly one value needs to be “sent back” from a function to the calling routine, it should generally be returned with a return statement rather than through a reference parameter.
* When two or more variables passed as arguments to a function need to have their value changed by that function, they should be passed by reference.
* When a copy of the argument cannot reasonable or correctly be made, such as when the argument is a file stream object, it must be passed by reference.

Here are three common instances when reference parameters are used.

* When data values being input in a function need to be known by the calling function.
* When a function must change existing values in the calling function.
* When a file stream object is passed to a function.

**OVERLOADING FUNCTIONS**: Two or more functions may have the same name, as long as their parameter lists are different.

**THE exit () FUNCTION**: The exit() function causes a program to terminate, regardless of which function or control mechanism is executing. A C++ program stops executing when a return statement in function main is encountered. When other functions end, however, the program does not stop. Control of the program goes back to the place immediately following the function call. Sometimes, however, the programmer wishes, under certain conditions, to terminate a program in a function other than main. To accomplish this, the exit function is used. To use the exit function be sure to include the cstdlib header file. The function takes an integer argument such as exit(0) indicating the program ended successfully.  
If you are unsure which code to use with the exit function, there are two named constants, EXIT\_FAILURE and EXIT\_SUCCESS, defined in the cstdlib for you to use. The exit () function is a nonstructured programming technique. Use with caution.

**STUBS AND DRIVERS**: Stubs and drivers are very helpful tools for testing and debugging programs that use functions. They allow you to test individual functions in a program, in isolation from the parts of the program that call the functions.

A *stub* is a dummy function that is called instead of the actual function it represents. It usually displays a test message acknowledging that it was called, and nothing more. Primarily, the stub allows you to determine whether your program is calling a function when you expect it to and confirm that valid values are being passed to the function. If the stub represents a function that returns a value, then the stub should return a test value. This confirms that the returned value is being handled properly. When parts of the program that call a function are debugged to your satisfaction, you can move on to testing and debugging the actual function themselves. This is where drivers become useful.

A *driver* is a program that tests a function by simply calling it. If the function accepts any arguments, the driver passes test data. If the function returns a value, the driver displays the return value to the screen. This allows you to see how the function performs in isolation from the rest of the program it will eventually be a part of.