C Programming Fundamentals

This document aims at explaining the basic programming principles of writing C code. It is broken down into a number of sections: files, variables, pointers, conditionals, loops, and functions. At the end of the document, there is an example showing a simple C program implemented for a microcontroller.

If you would like more help with C programming, please refer to this link. It contains wonderful examples and well written explanations of how to program in C. (Please note: the tutorial is for writing C code for your computer as opposed to a microcontroller.)

File Types:

Header files

Header files (typically denoted by the .h extension) can be used to contain function prototypes, structure definitions, and other C programming elements. The purpose of these files is typically to separate the source code files from information that is supplied to the compiler. Header files rarely contain source code. A header file can be included by multiple source files with the use of guard code. Please see the example at the end of this document to see guard code implemented. This code is used to ensure that the header file is not compiled multiple times, as this would cause errors relating to the redefinition of functions. Header files are not strictly necessary for writing C, but they make projects more understandable by keeping all important information in one place.

Source files

Source files (typically denoted by the .c extension) contain all of the source code for the functions specified by function prototypes. Source files may contain as many function definitions as users wish and they may also contain any of the information that a header file contains. The distinction of two file types (headers and source files) is done for understandability, but is not required to write C code. Source files may include a header file with the following line at the top of the file:

```
#include “header_file_name.h”
```

This line associates a header file with a source file and acts as a form of connector.

Variables:

Variables are a way of storing information in a C program. Variables should be declared at the start of a function. A variable declaration includes a type specification and an optional initial value. Variables are used for doing mathematical operations and conditional branching within C. Below is an example of some variable declarations. Each type will be explained shortly. Note that every line of C code is terminated by a semicolon “;”.

```
double myDouble = 34.1;
int myInteger = 5;
float myFloat = 3.542;
```

Data Types

int

(Integer, typically 16-32 bits) An int is used to specify an integral value (e.g. 1, 2, -5, 10) and may not have a decimal point. Mathematical operations on two integers will result in truncation of the result (e.g. 5/2 = 2.5 => 2)

float

(floating point – typically 16-32 bits) A float is a floating point type that may contain decimal points.

double

(floating point – typically 32-64 bits) A double is called double because it is typically twice the size of a floating point number and can contain larger values than those of a float.

char

(character – 8 bits) Character variables are used for 8-bit characters, such as 'a', 'b', or 'd'. The maximum integral value of a character is 255 (2 to the 8th power).

void

(Typeless) A void type cannot be instantiated as a variable type. This is typically used as a "generic" type when used as a pointer.

Specifiers

The unsigned and long specifies can be used in front of a data type to change the functionality of the data type

Unsigned

Unsigned data types cannot be negative, granting the type an extra bit to be used for storing larger values

Long

Long types provide an extra number of bytes for the data type so that it can store even larger values. This is useful if your variable exceeds the restrictions of an integer or a double.

Scope

All variables have a "scope" associated with them. This means that a variable declared and used in one scope may not be used in another scope. You can determine the scope of a variable by looking at what subset of curly braces it is within "{}". Example:

```
{
        int myVariable = 5;
        {
                int myNewVariable = myVariable + 1;
        }
        //The following can not be done because myNewVariable has left scope.
        //myVariable = myNewVariable + 1;
}
```

Arrays
Arrays of variables can be created by providing square brackets (“[]”) after the variable name with an amount of variables in the curly braces. Example:

```
int myArray[100];
```

This is an array declaration of 100 integers. To access an element of the array, just type the square braces after the variable name with the element index inside of it. Note, indexing starts at 0 and ends at size-1. Do not index into the element at index of [size], as this element does not exist and your code will break. Arrays are useful for holding a large number of variables within one variable name.

```
myArray[1] = 3; //Accesses the second element of the array, index 1
myArray[0] = 0; //Accesses the first element of the array, index 0
myArray[99] = 99 //Accesses the last element of the array
```

Comments
The above example also displays the use of comments within c code. Any section preceded by two forward slashes (“//”)will be ignored by the compiler, and can be used for writing comments in your code. Comments can also be made within a block section. Start a comment block with /* and then end the comment block with */.

```
//This is a comment.

/* This is an example of a comment block.
 *  It can also be multiple lines!
 */
```

Mathematical Operations
Mathematical operations may be conducted on variables, such as addition (+), subtraction (-), division (/), multiplication (*), and modulation (%) [This will return the remainder amount of division]. Please refer to the following example:

```
int myVar1 = 5;
double myVar2 = 7.5;
float myResult;

myResult = myVar1 + myVar2 / (6 + myVar2);
myResult += myResult%2;
myResult++;
```

The code example above shows a number of elements. Multiple mathematical operations may be completed in a single line of code and they will follow the order of operations. First, parenthesis are handled to create a value. Then, the order of operations is followed from left the right, with multiplication and division happening simultaneously before addition and subtraction. The modulus (%) operation returns the leftover of a (a%b) when a is divided by b with integers. Aka, if a =3 and b = 2, 3%2 = 1 (3/2 = 1 Remainder 1). Additionally, notice that operations can be simplified by the +=, *=, -= etc. operators. This is a shorthand notation for writing:

```
myResult = myResult + myResult%2; //Shorthand: myResult += myResult%2;
```

Also, notice in the earlier example that the '++' operator can be used to increment variables by a value of 1. This can also be done with '--' to decrement by 1. These operators may be placed before or after a variable. If they are placed before, the decrement or increment is performed before any other statements in the line. If it is placed after, the increment/decrement is done after the line has executed.

Pointers

Pointers are another form of variable, but are worth mentioning further. A pointer variable is declared by the "*" specifier. All variables inherently have their own pointer, which can be accessed by the "&" operator. A pointer is simply an address of a variable. With the address to a variable, we can find the original variable. This is an extraordinarily similar concept to house addresses. If you have the address to your friend's house, you can send a letter to it. Without the address, you can't send them mail. Pointers are extremely useful when passing variables between functions to allow different functions to modify different variables. A pointer can be directly thought of as "giving directions to" a variable. Please refer to the following code example:

```
int myInteger = 5;
int *mtIntegerPointer = &myInteger;
```

Within this example, we are setting the myIntegerPointer variable to the address of myInteger. With the variable address, we can additionally access the original variable itself by again using the "*" operator. Now, if we execute the following code:

```
*myIntegerPointer = 4;
```

The variable myInteger will now also contain the value of 4 instead of the original 5. Pointers are extraordinarily powerful when programming with C. The following C statement would execute even though myInteger was never directly set to 4:

```
if (myInteger == 4)
{
        // This code executes
}
```

Functions:

Functions are used within C programming so that code can be reused throughout your program. A function can be called from anywhere as many times as you would like and it can react to inputs that you specify as arguments. Functions are extraordinarily powerful, as they cut down on the amount of code that you must write. Additionally, they help make code much more understandable.

Prototypes

A function prototype is a way of telling the compiler that a function will exist somewhere in the source code. A function prototype contains a return type, a function name, and a list of function inputs. Functions may only provide one output (this is why pointers are useful for functions, as it allows functions to return multiple values), but can take any number of inputs. Below is an

example of a function prototype:

```
int addTwoNumbers (int numberOne, int numberTwo);
```

In this prototype, we can see that the function is named addTwoNumbers and that it returns a value of type “int”. Additionally, this function accepts two input parameters, namely numberOne and numberTwo, and these inputs are both also integers.

Main

The main() function is a C program's main entry point when it begins execution. This is the very first function that will be called. A main function typically does not accept parameters, although it possibly can, and typically returns an integer upon completion. For microcontrollers, main functions return an integer and accept no parameters. The main function is the entirety of a program and any functionality that a program has must be contained within the main function. The main function does not need a prototype. Main function declarations typically are as follows:

```
int main(void)
{
	// Write your program code here.
}
```

Definitions

Once a a function has a prototype, it must then be defined. We must say what the function will actually do within our code. A function definition is simply a container of all the code that the function will do, and is similar to the way main() works. Please refer to the following definition:

```
int addTwoNumbers (int numberOne, int numberTwo)
{
	// This function will return the addition of two numbers.
	int myValue = 0;

	myValue = numberOne + numberTwo;

	return myValue;
}
```

Take note of the final line, where we specify “return myValue;”. The word “return” is a keyword in C that specifies that this is the value that the function will return. It then ends the function execution and returns the value specified to wherever the function was originally called from. If a function's return type is declared as “void”, it may not return a value at all.

Calling Functions

To utilize functions that we have written within our main function, it must be called with input parameters supplied.. Additionally, if the function outputs a value, this value should be assigned to an output variable. Please refer to the following example:

```
int addTwoNumbers (int numberOne, int numberTwo);

int main (void)
{
	int myValue1 = 5, myValue2 = 7, myAddedValue;

	myAddedValue = addTwoNumbers(myValue1, myValue2);
}
```

In this example, the function that was declared earlier is called (note, the definition must exist later or earlier in the file) to add myValue1 and myValue2 and it then stores the result into myAddedValue.

Conditionals:

In C programming, conditional statements are used for branching code based upon variable values. This is done using the if-elseif-else statement. It also utilizes rational operations, such as > (greater than), < (less than), == (equals), != (does not equal), <= (less than or equal), and >= (greater than or equal). Not all portions of an if statement must be used for it to function. Please refer to the following examples.

```
int myVar1 = 1, myVar2 = 2, myVar3 = 2;

if (myVar1 <= myVar2)
{
	// This code executes when myVar1 is less than or equal to myVar2.
}
else if (myVar2 == myVar3)
{
	// This code executes if the first condition fails and myVar2 is equal to myVar3.
}
else
{
	// If neither of the two above conditions is true, this code executes.
}
```

Note that in the above example, because myVar1 is less than or equal to myVar2, the first statement will execute. Because the first statement executed, the second and third blocks will not. Even though myVar2 is equal to myVar3, the code will not execute because the first block was true. Make sure to not make the mistake of only using a single equals sign! When this is done, you are actually assigning the first variable to the value of the second parameter, but you are not checking equivalence! Portions of the if statement may also be used in separate cases. The only portion that must always exist is the if (condition) portion.

```
	int a = 1, b = 2;

	if (b <= a)
	{
		// Execute this block of code is b is less than or equal to a.
	}
```

This code will execute just fine without the else if and else portions of the if block.

Loops:

C code also has two useful structures for looping through code multiple times. This can simplify the amount of code that you have to write within your program.

For Loops

For loops are useful for looping through a section of code a specific number of times. A for loop declaration includes the specification of a counting variable (integer), a conditional, and a post-loop operation. Please look at the following example:

```
int index = 0;
int tempVariable = 5;

for (index = 0; index < 100; index++)
{
	tempVariable += index;
}
```

The above example will execute the loop 100 times, incrementing the value of index by 1 after each execution. In the code, we specify with the first portion of the for-loop that we are using index as our counting variable, the conditional for ending the loop is index < 100, and that after each execution of the loop, we increment index by 1.

While Loops

While loops are loops that continue executing until a condition fails. They are commonly used for continually executing code until some state has been achieved. Within microcontrollers, while-loops are used to ensure that a program will never end its execution. Please refer to the following for defining a while-loop:

```
int myFlag = 0;
int someNumber = 5;
while (myFlag == 0)
{
	myFlag = myFlag + someNumber;
}
```

This particular while-loop will execute once. After it does, the value of myFlag will no longer be zero, and the while-loop will finish. For microcontrollers, we typically replace the conditional with '1' so that the while-loop will execute forever.

```
while (1)
{
	//Embedded system loop
}
```

Example of a basic C program:

Header: (myHeader.h)

```
#ifndef MY_HEADER
#define MY_HEADER
//above is the guard code to ensure that the header is compiled only once.

//Declare our function prototypes
double myFunction(int parameterOne, double parameterTwo);

//Below is the end of the guard code.
#endif
```

Source: (mySource.c)

```
#include "myHeader.h"

int main(void)
{
	int myNumber = 7;
	double myOtherNumber = 6;
	double myResult;

	myResult = myFunction(myNumber, myOtherNumber);

	//Enter the embedded loop.
	while (1)
	{
		myNumber++;
		myResult = myFunction(myNumber, myOtherNumber);
	}
}

double myFunction(int parameterOne, double parameterTwo)
{
	double myValue;
	myValue = --parameterOne + (parameterTwo%3);
	myValue = myValue%4 * (parameterOne + parameterTwo);

	return myValue;
}
```

Note that this program does not actually do anything useful, it is just an exercise to understand the C programming language.