Introduction to Functions

Functions are a key part to modular programming. They can save a lot of repetitive coding. Functions can be tricky however, particularly when passing arrays and pointers. We haven't talked about arrays and pointers yet, so we'll just deal with very simple functions right now. At this stage we'll only deal with functions that have simple variables in their arguments.

A function is declared by the data type of the value that it returns. For example, a function can be a float, int, double, etc. Many functions are of the type "void", meaning that they don't return anything. We have already seen that every C program starts with an integer function named *main*. This function returns an integer to the operating system when it is finished. In our examples, we simply have it return the value 0 to the operating system.

Some key points to remember about functions:

1. Functions must be declared properly. If they are not, serious memory issues could occur! So, if your function returns a double, make sure you declare it as a double function.
2. Functions must be written outside of the main() function. In your source code, functions can be written anywhere, but do not write them within each other. You can "call" functions within functions, but you can't write functions within functions. Hopefully, this will become more clear by example.
3. When variables are used as arguments in a function, they are **copied** to the function. If you change the variable within the function, it will not be changed outside of that function. Now, this will be quite different when we start dealing with *pointers*.
4. Functions may include local variables which are not seen outside of that function.

These may not make sense until later, after you have seen several examples.

Let's start with a simple function of type *void*. This type of function does not return anything. Once it has finished, it will return back to the function from which it was called.

code08.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main(void)
{

// I declare a function as a void function named tell_it_dude.
// The compiler sees that this is a function and not a variable
// because of the parentheses after its name.

void tell_it_dude();

double A=20.0;

      // Call the function "tell_it_dude".
      // The function has an argument that must be a double.

      tell_it_dude(A);

      // Lets prints something to screen to indicate
      // that we made it back here.

      fprintf(stdout,"Wow, that was a funky function\n");

return 0;
}

// TELL IT DUDE
// This is a function that says some cool stuff

void tell_it_dude(some_number)
    double some_number;
{

   fprintf(stdout,"Dude, that number is %f\n",some_number);


return;
}
```

The output should be:

```
Dude, that number is 20.000000
Wow, that was a funky function
```

Notice a few things here. First, we must declare *tell_it_dude* in the function from which we call *tell_it_dude.* We called *tell_it_dude* in our *main* function, so we had to declare it

there. There are multiple ways to declare and define functions, but we'll just do thing my way for now. When I declare a function, I leave contents within the parenthesis empty.

```
void tell_it_dude();
```

This is ok because the function knows what to expect as arguments because we'll define that in our function definition later.

In *main*, we call *tell_it_dude* and pass the double variable *A* into it. *tell_it_dude* is expecting a double, so don't make a mistake here! If you put a float, integer, or anything else in here, you're asking for serious trouble!

tell_it_dude is then called with the argument A. The code then jumps to where you have defined *tell_it_dude*. The definition of *tell_it_dude* can be anywhere in the source code, as long as it's not defined in a function itself. In this example, we define *tell_it_dude* after the *main* function.

The basic structure of a function definition is:

```
Type function_name(arguments)
 Declaration of arguments;
{
     Your code inside the function

return  whatever gets returned;
}
```

If you look at the *code08.c*, you will see that our function *tell_it_dude* has this general structure to it.

Lets copy *code08.c* to *code09.c* and make some modifications to *tell_it_dude*. We will make it a double function rather than a void function. Therefore, *tell_it_dude* will return a double to the main code.

code09.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main(void)
{

// I declare a function as a double function named tell_it_dude.

double tell_it_dude();

double A=20.0;
double what_dude_thinks;

      // Call the function "tell_it_dude".
      // The function has an argument that must be a double

      what_dude_thinks = tell_it_dude(A);

      // Lets prints something to screen to indicate
      // that we made it back here.

      fprintf(stdout,"Wow, that was a funky function\n");

      fprintf(stdout,"And the dude says the number should be %f\n",what_dude_thinks);

return 0;
}

// TELL IT DUDE
// This is a function that says some cool stuff

double tell_it_dude(some_number)
    double some_number;
{

   fprintf(stdout,"Dude, that number is %f\n",some_number);

   // I think the number should be plus one

   some_number = some_number+1;

return some_number;
}
```

The output should be:

```
Dude, that number is 20.000000
Wow, that was a funky function
And the dude says the number should be 21.000000
```

As another example, let's use the rectangle rule to integrate the function x^2 from x=0 to x=1. We'll use 100,000 intervals. We'll use three functions in this code. There are not too many comments in this code, but you should try to mentally reverse-engineer it. Make sure you see how the code flows through these 3 functions. Make it make sense to you!

code10.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main(void)
{

double upper_bound = 1.0;
double lower_bound = 0.0;
int number_of_intervals = 100000;
double rectangle();
double answer;

      //call the rectangle function with the appropriate arguments

      answer=rectangle(upper_bound, lower_bound, number_of_intervals);

      fprintf(stdout,"The answer is: %f\n",answer);


return 0;
}

// RECTANGLE
// This function computes the integral using rectangle rule
double rectangle(upper_bound, lower_bound, number_of_intervals)
   double upper_bound;
   double lower_bound;
   int number_of_intervals;
{

double h;
int ii;
double sum;
double my_function();
double x;
double answer;

    // h is the size of each interval
    h = (upper_bound - lower_bound) / (number_of_intervals*1.0);

    sum = 0.0;
    x = lower_bound;

    for (ii=0; ii <= (number_of_intervals -1); ii++)
    {
       sum = sum + h*my_function(x);
```

```
        // we increase x to the next value
        x = x + h;
      }
      answer = sum;

return answer;
}

// MY_FUNCTION
// This function returns the value of x squared for a given x.
double my_function(x)
  double x;
{
double value;

  value = x*x;

return value;
}
```

Your output should be:

```
The answer is: 0.333328
```

Which is pretty close to the right answer, right? We should always check such things when we can!

Now, with this code, you can change *my_function* and integrate most anything you want.

Ok, that's it with functions for now. We will certainly return to this topic, but there are other things we should learn first.

A final word on function – code is all about functions! The more you can compartmentalize things into modular format, the nicer your code will be. Don't go overboard though because that could make your code unreadable. Functions are particularly nice because they can be used over and over within a single code. I will often use my functions in multiple codes because they are easy to cut and paste. If you have a nice personal library of useful functions already typed out, you can put together code much more quickly and efficiently.