Week 2

OK, so now we know how to write simple C programs that contain loops. Now we move on to two powerful features. The first is *functions*, a way to write a piece of code that can be "reused" any number of times in the program. Another is *arrays*, which provide a way to operate upon large "chunks" of data.

2.1 Functions

Understanding functions is essential to writing useful programs in any language. A function in C is not much different from a function in math — it is a piece of code that has a name, and can be reused to perform the same function over and over again. Imagine you have a program that does some sort of exponential decay calculation. Certainly, somewhere in this program you will have to calculate the value of some variable raised to the power of some other variable. This would be done with a function that had as its "arguments" the base and the exponent, and would "return" the computed value. All this will become clearer in a minute.

Arguments

A function performs its calculations on some data that you must specify. For example, if you wanted to write a function that raised some number to some integer power, you would have to specify the number and the power to the function for it to have enough information to do its task. These pieces of input data are called **arguments** to the function. They behave just like local variables inside the function.

Return values

Once a function has done its calculation, it may want to return the result to the function that invoked it. In our power function example, the function would return a float. This is accomplished with the `return` statement, as we will see in a moment.

Note that both return values and arguments could be done as global variables — Functions do not have to take arguments and do not have to explicitly return anything. It is possible for you to put values into global variables, have the function do its computations assuming that those global variables have been set up, and put its result in another global variable that will be available to the caller. This is VERY bad practice; it makes your code hard to read, and can introduce bugs that are very difficult to track down. In general, global variables are frowned upon if they can be avoided. There are many situations where they can't be, but if something can be an argument, make it so.

Declaring functions

Now, we are ready to write a function. Let's stick with the power example for now. This function will need to take as its arguments a floating point base and an integer exponent, and will need to return its computed floating point number to the caller. The declaration is shown in figure 2.1.

```
float power(float base, int exponent);
```

Figure 2.1: A sample function declaration

Note that the type of the return value comes before the function name. If the function does NOT return anything, the return type is **void**. Similarly, if the function does not take any arguments, its argument list is **void**. This should remind you of the main function: `void main(void)`.

There are several important things to note about function declarations. First, if the function takes more than one argument of the same type, it is still necessary to explicitly give the type of each argument. That is, it is not possible simply to have a comma separated list of variable names as it was with variable declarations. Second, it doesn't matter what you call the arguments. I see many students who think that the arguments have to have the same name as the variable that will be passed to the function. This is not so.

Functions should be declared earlier in the C file than they are called. The actual code for the function (the definition, which we will talk about in a minute), can be anywhere as long as the declaration is before the actual call. This will be discussed in more detail in a moment.

Defining functions

The distinction between function definitions and function declarations is important. A function declaration simply tells the compiler what the type of the return value is, and what the types of the arguments are. This is so that the compiler can do **type checking**. If you pass the wrong type of argument to a function, the compiler will let you know about it. If, however, you don't declare your function ahead of time, the compiler will issue a warning but will assume you know what you are doing and continue.

Important note: If you define a function earlier in the C file than you ever use it, you don't need to declare it. The definition will be checked against the declaration for types, and if no declaration exists, the types specified in the definintion will be used.

So what is a definition, anyway? Well, let's write out the whole code for the power function. That code is in figure 2.2.

```
 1: float power(float base, int exponent)
 2: {
 3:   int i;
 4:   float result;
 5:
 6:   result = 1.0;
 7:
 8:   for (i=0; i < exponent ; i++)
 9:     {
10:       result *= base;
11:     }
12:   return result;
13: }
```

Figure 2.2: The definition of the `power` function

Let's look at this definition carefully.

Line 1 looks just like the declaration, but **there is no trailing semicolon**. If you put a semicolon after the first line of a function definition, the compiler thinks you are declaring it again, and will spit out tons of completely incomprehensible syntax errors when it hits the block of function code. So DON'T DO IT!

Lines 2-13 are the function code.

Lines 3 and 4 are declarations of variables that are *local* to this function — that is, they have no meaning outside of the function `power`.

Line 6 initializes the variable `result` to be 1.0. This is done because we are going to multiply the result by `base` many times and we want the initial value to be one.

Lines 8-11 are the meat of the calculation. The for loop is one of the most common kinds - it uses the dummy variable `i` for nothing other than to count how many times we have gone through the loop. So the loop executes `exponent` times, and each time it multiplies `result` by `base`.

At the end of the loop, the variable `result` holds the desired value, so we return it to the calling function.

Note that this function isn't nearly as general as you want it to be. The exponent should be able to be a non-integral value, and should also be able to be negative. Try to figure out how the `power` function would behave in each of these cases.

Calling functions

Finally, we look at how you would use a function. Let's look at an entire C file that uses the `power` function several times, shown in figure 2.3.

This program makes use of most of the things we have talked about so far. Note that you don't have to assign the return value to a variable, but you can just use it in any kind of expression where its type is valid.

Scope

A frequent source of error when beginning programmers use functions is that they misunderstand when a variable can be legally used and when it cannot. Here we summarise the scope rules using the above program as an example. A detailed discussion of scope can be found in the reader; the lectures and classes will also discuss it.

Here is a summary of the scope rules.

The caller function's variables are "invisible" to the callee function. In the above example, the variable `mypower` is local to `main()`. It is an error to try to use it in an expression inside the function `power()`. (Note that it is legal to declare a *new* variable named `mypower` inside the code for the `power()` function, However, such a variable would be distinct from the identically named variable that was declared in `main()`; modifications to one of them will not affect the other.)

For example, even though the `main()` function calls `power()` using the statement `power(2.5, j)`, the variable `j` is still "invisible" inside the body of `power()`. (The general name for such a scope rule is *call by value*.)

Note further that if the variable named `j` inside `main()` were renamed `i`, then this variable would be utterly distinct from the variable named `i` that is present inside the body of the `power()` function. Modifying the variable `i` inside `power()` would not affect the variable `i` in `main()`.

Local vs. Global

Given the above scope rules, how can we design functions that operate upon the same piece of information? One way is to use *global variables*. These are variables that are declared outside of any functions. Global variables have a scope consisting of the entire C file beneath where they are declared. A local variable is one that is declared within a block of code, and its scope is the entire block. If a variable is declared with the same name as one that is already declared, the new variable takes precedence until its scope ends. For example, consider the code in figure 2.4, with the declaration of the first "a" moved outside the main function.

The global variable "a" is valid everywhere in this C file, even in other functions (functions are discussed in depth in section 2.1).

```
#include <stdio.h>

float power(float base, int exponent)
{
  int i;
  float result;

  result = 1.0;

  for (i=0; i < exponent ; i++)
    {
      result *= base;
    }
  return result;
}

void main(void)
{
  float mypower;
  int j=10;

  mypower = power(2.5,j);
  printf ("2.5^10 = %f\n",mypower);
  printf ("5.2345^3 = %f\n",power(5.2345,3));
  if (power(2,3) > power(3,2))
    {
      printf ("Wow!  2^3 > 3^2!\n");
    }
  else
    {
      printf ("Wow!  2^3 < 3^2!\n");
    }
}
```

Figure 2.3: Using functions

```
int a;

void main(void)
{

  a = 10;
  {
    int a;

    a = 20;
  }

}
```

Figure 2.4: A global variable

Important Note: Use global variables with caution. The fact that they are "visible" to all functions also means that you could inadvertently modify it where you don't want to. In general, it is bad idea to give global variables generic-sounding names like `i, j` etc.

2.2 Arrays

Arrays provide you with a way to conveniently store lots of related variables. You can think of an array as a bunch of variables (all of the same type) lined up next to each other. From now on, we will refer to each of these variables as a "slot".

Imagine you were writing a program to count the number of occurrences of each letter of the english alphabet. One way to do this would be to have 26 variables, one for each letter, and each time you saw a letter, you would increment the appropriate variable. This approach is extremely cumbersome, wasteful, and prone to error. A *much* better way to do this would be using arrays.

Declaring arrays

Arrays are declared just like any other variable, except that you must specify their *size*. This is done using square brackets, as shown in figure 2.5.

Notice that, just like any other variable, an array has a type (integer), and a name (letters). The size of our array is 26. Now, when you want to access one slot in the array, you use the brackets in the logical way. Figure 2.6 shows how to add an occurrence of a letter.

```
int letters[26];
```

Figure 2.5: Declaring an array

```
int letters[26];  /* The letter count */

char c;           /* The letter */

/* Let´s assume that the letter has already been read,
 * and is in the character ´c´ */

letters[c - ´a´]++;
```

Figure 2.6: Counting letters

Take a look at the weird syntax inside the bracket: `c - ´a´`. Remember how we said that character variables could be added and subtracted just like integers? This is what we're doing. If you subtract an 'a' from a letter, you get its position in the alphabet, starting with a = 0, b = 1, etc.

Array bounds

This brings us to an important point: **ARRAYS START AT ZERO**. What this means is that the first slot in an array has index zero, not index one. Some beginner programmers get around this by declaring their arrays one bigger than they need, and starting at one. DON'T DO THIS. It's terrible practice, and you will get graded down for it.

Note that this ALSO means that the last element in our `letters` array is 25, not 26. So, even though our array has 26 slots, remember that the indicies go from 0 to 25, not 1 to 26.

C is too stupid to check that the number you use as your array index is "in bounds", that is, within the declared size of the array. You can even use a *negative* number! The results of out of bounds array accesses are undefined, and can cause your program to crash in horrible, unreproducible ways. Be careful to make sure that if your array is 10 elements wide, you only access from elements 0 to 9, not 0 to 10 or 1 to 10[1].

[1]If you do this, the TA on duty will thrash you with a wet noodle

Symbolic constants

Sometimes you will want to be able to define a descriptive name for some "magic number". For example, if you are writing a graphics program that deals with images that are always 1024 pixels wide, you would like to use the constant `IMAGE_WIDTH` rather than 1024 in your program for two reasons: First, it makes the code more readable both by you and anyone else who might have to look at it (the graders?). Second, it allows you to change the definition of the constant in one place and have that change reflected in many places..

Constants are defined with the `#define` command. You can use the `#define` command anywhere in your code, but it is convention that all your `#define`'s should appear directly after your `#include` statements. So, for our example, you use the directive shown in figure 2.7.

```
#define IMAGE_WIDTH 1024
```

Figure 2.7: Defining a constant

The `#define` directive sets up strict textual equivalence between the symbolic name for your constant and its value. That is, wherever you type `IMAGE_WIDTH` in your C file, the compiler will see the value 1024. In fact, it has no idea that the number is a symbolic constant. In this respect, constants are different from variables in that you cannot change them at run-time.