P2T: C Programming under Linux

Lab 4: Functions and Pointers

Introduction

The purpose of this laboratory is to help you become familiar with the concepts of functions and pointers in a C programming. This material has been covered in Lecture 5 of the C component of P2T: C Programming under Linux.

Functions

In this course, you have written only short pieces of code so far. In general, as the amount of code in a project gets larger, it becomes much harder to understand it without adding additional structure (just as we structure prose into sentences, paragraphs, chapters and volumes as a written work becomes bigger to allow us to process it more easily). The next unit of "code structure" in C, after blocks, is the *function* (sometimes called a "procedure" or a "subroutine" in other languages). Functions allow us to encapsulate commonly used code into compact, named, units that can be reused. C programs are typically written by combining new code the programmer writes with "pre-packaged" functions available in the C standard library, or in other libraries from other sources. You've already met many functions (**printf**,**scanf**,etc) and you've written at least one in every piece of code you've produced: the `main()` function.

The generic syntax for a function definition is:

```
return_type function_name(parameters list) {
  declarations;
  statements;
  return result;
}
```

The function definition has two parts.

1. Function declaration - this is the first line of the function definition. It specifies the return type of a function (the type of the returned value), function

name and the types and names of the arguments that function expects to receive (this combination of things also defines the "function signature", which is the "type" of the function).

2. Function body - this is the block of code that is executed each time the function is called. This block contains all the declarations, statements, etc. that define how a function operates. Notice that a function body has to be enclosed in the curly braces . The function body is a lot like a self-contained "subprogram" that is executed each time the function is called (some languages call functions "subroutines" for this reason). A `return` statement causes the function to end immediately, with the value after the statement being given as the "result" of the function. (There can be more than one `return` statement in a function, if the function has multiple possible ways to finish.)

It is important to realise that *functions* in C are *not* mathematical functions - while a function that takes arguments and returns a value may resemble a mathematical function externally, the philosophy behind them is entirely different. For example, it is sometimes desirable to have a function which does not return a particular value but rather does some other things internally (e.g. prints a new line character or gives some other output to the screen, changes value of its argument, etc.). Such a function might have a return type `void`. In this case the `return` statement is not necessary (unless you want to leave the function early) as there is nothing to return. We call the effects of a function which are not reflected in its return type *side effects*.
We can, of course, define a function which has both side effects *and* returns values. For example, the `printf` function from the C standard library returns a value equal to the number of characters it passed on to the screen and also outputs a string on the screen (which is its main feature in most uses).

A better analogy for functions may be drawn from cookery. There are many recipes, for example, which use one or more of the classic "mother sauces" of French cuisine. Describing how to make a *béchamel* every time you need it would get tiresome, however; so you simply direct the reader to the page where you've already described how to make one. Much as with functions, every time you make a *béchamel* sauce, it's a different one, made with a different measure of milk, butter and flour (but always with milk, butter and flour) - just as a function always might require an int and two floats (but each time you call it, you can use different ones). The side effects of following the recipe for *béchamel* are the dirty saucepan, spoon etc which were created in the process of making the actual "return value" - the sauce itself.

A function is “called” by invoking its name, followed by a list of all the values to be given to it (one for each parameter) enclosed in parentheses. The result of the function - its return type - is treated as essentially its value, as if the entire function call was just a single variable. So, given this partial function definition, the code in main would invoke f twice:

```
int f(int a, int myint, float prec) {
  int x;
  (some statements here, which may use a, myint and prec
   as well as x)
  return x;
  }

int main(void) {
  int y, z;

  y = f(9,34,5e-5);
  z = f(y, 36, 5e-5);

  return y*z;
}
```

Logically, this does something almost the same as:

```
int main(void){
  int y,z;
  {
    int a=9;
    int myint=34;
    int prec=5e-5;
    int x;
    (some statements here)
    y = x;
  }
  {
    int a=y;
    int myint=36;
    int prec=5e-5;
    int x;
    (some statements here)
    z = x;
  }
  return y*z;
}
```

Notice that, in particular, the separate calls to the functions are like separate blocks - the variables in one call are not persistent to the next call, nor are they accessible outside the function. In fact, functions are *more* restrictive than blocks - not only are their variables inaccessible outside the function (or between function calls to the same function), but all variables outside the function (unless in file scope) are also out of scope inside the function, as the scope of the function is drawn from where in the file it is *defined*, not where it is called.

(The values that we “pass” to a function are *copied* to the *new* variables inside the function when it is called, as we’ve represented at the start of the logical blocks.)

Specifically, that means that the two “y = x” and “z = x” statements don’t reflect what actually happens when a function returns - the function `f` doesn’t know about `y` or `z`, it just throws back a value when it returns (via a CPU register), and the continuing code that is returned to is responsible for putting the value in the correct place.

This is a logical requirement of the fact that a function is a “bottling up” of a set of statements so they can be used anywhere - clearly, they can’t depend on any values not used in the algorithm itself. This does present an immediate issue in using functions to modify values in variables... which we will address in the next section.

Similarly to variables, functions have to be “declared” in the code before their first use - the compiler needs to know what their signature looks like so it can appropriately arrange for values to be passed to and from them. However, they do not have to be *defined* before they are used. Thus, you can declare the function without including the function body in a “function prototype”, as long as you later provide a full function definition (declaration plus body) later on. That is:

```
//function prototype
return_type function_name(parameters list);

/*
  Lots of Code here
*/

//function definition
```

```
return_type function_name(parameters list) {
  declarations;
  statements;
  return result;
}
```

You can think of this as being a bit like how you can declare a variable without giving it an initial value, with the caveat that you're not allowed to change the "value" of a function after the first definition.

Usually, the function prototype is declared at the beginning of a file (before the `main` function), providing the external interface for the function to be used, and the function is defined later in the code (possibly after the `main` function). In bigger projects that contain several files, functions are declared in separate file(s) and their prototypes are *included* before their calls. In both cases, this is mostly pragmatic, allowing you to more easily read a "precis" of the code, and get to the "important" core of `main`, without having to scroll through potentially many other function definitions first.

Pointers

A *pointer* is a variable that stores a memory address of another variable (that is, it *points* at the location used to store that variable). Each memory address is simply a number which uniquely identifies that location (much as the houses on a street usually have unique numbers which identify them). In this sense, a variable name directly references a value and a pointer indirectly references a value. Pointers, like any other variables, must be declared before they can be used. The declaration

```
int *ptr_count, count;
```

declares the value of type pointer to type int (i.e. `ptr_count` is a pointer to an integer value). Notice the dereference (or indirection) operator (`*`) before the `ptr_count`. In this statement it indicates that `ptr_count` is a pointer. Note that (`count`) is here declared as type `int` here, *not* pointer to int. Each pointer must be declared with the * prefixed to the name.

The next step is to initialise a pointer variable. This can be done either when a pointer is declared or in an assignment statement:

```
\\at the declaration time:
```

```
int var, *ptr_var = &var;
\\or separate assignment statement:
ptr_var = &var;
```

The &, or address operator, is a unary operator that returns the address (id of the location in memory) of its operand. The statements above assign the address of the variable `var` to pointer `ptr_var`. Variable `ptr_var` points to `var`. There is nothing stopping you from assigning the address of a variable of type A to a pointer to type B, but this should be avoided unless you know exactly what you're doing (the pointer doesn't know that the address you've given it is the wrong type, and will treat the value as the type it is intended to point at).

Outside of a variable declaration, the * operator has another use. It returns the value which is stored in the memory location with the given number. You can rephrase the declaration of a pointer: *"If I dereference `ptr_var` I will get an `int`"*. For example, the printf statements in the code

```
int x = 5, *ptr_x = &x;
printf("%d\n", *ptr_x);
printf("%d\n", x);
```

would output the same value. Below is the table of the pointer operators in use.

Example Code	Meaning
int thing	declare an integer variable called "thing"
&thing	address of variable "thing" (a pointer)
∗thing	the contents of the memory location with number "thing"
int *thing_ptr	declare a "pointer to an integer" variable
∗thing_ptr	integer variable at the address "thing_ptr" points to
&thing_ptr	the memory location in which thing_ptr is stored

Challenging Questions (1)

Exercise 1: Pointers

This is a debugging question.

```c
#include <stdio.h>


int main(void) {

  int i,*myPtr, **myPtrPtr;
  int a[2] = {1,2};

  i = 6;
  myPtr = i;
  myPtrPtr=myPtr;
  {
    int i = 7;
    myPtr = 45;
    printf("The value %d is stored in location %p\n",
   myPtr,myPtr);
    printf("The value %d is stored in location %p\n", i, i
   );
  }


  printf("The value %p is stored in the location %p\n",
   myPtrPtr, myPtrPtr);

  printf("The value %d is stored in location %p\n", i, i);


  return 0;
}
```

The above code does quite a lot of work with pointers (and the addresses of variables). Unfortunately, within the `main` function, most of the `&` and `*` have gone missing! With reference to the expectations of the `printf` statements, appropriately restore them to the right places.

What do the `printf` statements inside the inner block in `main` print? Why? (Specifically, consider where the values modified are, and what ”variable names” they associated with. Add comments to explain this.)

What does myPtrPtr contain? How would you modify the `printf` involving it to also print the value of `i` (in the outer scope of `main`)? (Add comments to the code, and alter the printf appropriately.)

You also have an array, a, declared and initialised. Write one or more printf statements to print the values of a, &a, *a, a[0] and a[1]. What do you notice about the output of these? Write comments to describe what you think is happening. (The warnings you may get from the compiler might help; feel free to talk to a demonstrator as well.)

Pointers as Function Arguments

The function arguments we used in the previous section were always non-pointer variables. In C, variables are passed by value (call by value) meaning that their contents were *copied* to the variables named in the function scope. Thus, a function cannot modify the values in the variables used to call it. However, pointers can be passed as function arguments as well. This allows us to use a method referred to as “call by reference” as opposed to “call by value”. The value in the pointer is a reference to a location in memory - thus, the function gets to know the same location in memory and can directly modify the contents of it (even if the variable name referring to it is not in scope in the function).

Dangers of Pointers

Care needs to be taken when dealing with pointers. As you know, pointers contain an address of a memory location but C does not provide any guarantee about the validity of the memory they point at. In other words, the contents or use of an address may change, without warning. For example, if we use the address of some variable and then it reaches the end of its allocation lifetime (we exit the block in which it has scope), the memory attached to it is no longer associated to it and may be used for other things. For example:

```c
#include <stdio.h>

int *f(int b) {
  int a = 2 * b;
  return &a;
}

double f2(double a, int b) {
  double d = a * b;
10   return d / 2;
11 }
12
13 int main() {
14   int c = 3;
15   int *p = f(c);
16   f2(5.0, c);
17   printf("%d\n", *p);
18   return 0;
19 }
```

The result of the printf statement on the 17th line most likely will be unexpected. This happens because, the memory location pointed by the pointer `p` gets overwritten by the function `f2`. Even though, the pointer `p` points to the same location in memory, the contents of that memory has been altered as the variable `a` in the first function is not in scope in the main function. (Your compiler would usually give a warning about this, if you turned on warnings with -Wall).

Pointers and Arrays

Pointers and arrays are very closely related. You do not have to explicitly declare a pointer to an array because the bare name of an array is a pointer to the memory location where the first element of an array is stored. This is very convenient if we want to pass an array as a function argument. So now a function prototype might look like:

```
void function(int n, int array[]);
```

or...

```
void function(int n, int * array);
```

The `function` call in the `main` function would look like:

```
function(n, array);
```

If you absolutely need get an address of an individual element of an array, you can use the `&` operator (e.g. `&array[3]`).

Passing multidimensional arrays is a lot easier in C99 (and later) than in earlier versions of C. Because of the extra structure in a multidimensional array, you must pass integers for each of the dimensions of the array - and use a special form of the array declarator in the function definition to let the function know that these actually are the dimensions of the array. If you think back to the last lab, you know that C considers arrays to be “built” from right to left, so the left-most index is the “optional” one which is turned into a pointer, so you can do:

```
void function(int n, int m, int p, int array[][m][p]);
```

or

```
void function(int n, int m, int p, int (*array)[m][p]);
```

(the parentheses are needed around the *array in the latter example to force the ”pointer” bit to happen first - this is a ”pointer to an array of m arrays of p integers”, not an ”array of m arrays of p pointers to integers” which is what you’d get without them). We still call this in the same way:

```
void function(n, m, p, array);
```

but inside the function, the array is immediately reinterpreted in the correct format that we’ve specified. (Obviously, you need to make sure that you get the values for the dimensions you pass the function correct!)

Command Line Arguments

Now that we know more about functions and pointers we can introduce a new prototype for the `main` function. We started with a signature that looks like

```
int main(void)
```

However, the alternative, full, signature is

```
int main(int argc, char *argv[])
```

You will notice that this signature adds two parameters to the `main` function. This version of `main` allows us to make use of command line arguments (i.e. now you can write some arguments after the program name on the command line: `mycode 1 3 4` where 1, 2 and 3 are three arguments passed to the program “mycode”).

The variable `argc` keeps track of how many command line arguments, including the name of the program, were passed on to the program (e.g. if your program accepts one command line argument the `argc` will be equal to 2).

The `argv` is an array of strings that contains, in order, the text of each argument. (This does not use the multidimensional array scheme for functions, as each string might be a different length: `argv` is strictly, *not* a 2 dimensional character array, but is more like an array of pointers.) So, for example, `argv[0]` is a string containing the name of the program (including any path which was included), and `argv[1]` would be the first argument passed to it (if one was passed). You need to check the value of `argc` to determine how many strings `argv` contains in total...

The variable names `argc` and `argv` are not enforced, but they are called so by convention, and so changing them risks confusing other people reading your code.

Challenging Questions (2)

Exercise 2: Functions and CLA

The factorial of an integer is the product of all of the integers between 1 and it, inclusive.

Write a function, called `factorial` which, when passed an `int`, returns the factorial of that number, by using an appropriate loop inside the function.

Write a `main` function to allow you to test this: you should use the version of main which takes Command Line Arguments, and calculate the factorial for every integer parameter passed on the command line. (Prompting for input interactively is *not* the same thing!)

One feature of functions which is not as commonly used is that they can call themselves. We can use this recursion to calculate a factorial in a different way: For any given number N, $N!$ must be equal to $N \times (N-1)!$, by definition. (And since $1! = 1$, we can stop once we get to 1).

Write a second function, `rfactorial` which uses this feature to calculate the factorial of a number passed to it. Specifically: if the `int` passed to it is 1, it should return the value `1`. Otherwise, if it is passed the value X, it should call the function `rfactorial` itself (with a value X-1), and return the result, multiplied by X.

(You will need to declare a function prototype for `rfactorial`, before its definition, for this to work.)

By appropriately altering your `main` function, test that this works.

Exercise 3: Functions and Arrays and CLA

In Lab 3, you created a piece of code to output a PGM image. At present, all of your code is in one function (`main`).

Appropriately separate your code into three parts - one function which takes an array (and values for its dimensions) and fills it with the desired image, one function which takes an array (and values for its dimensions) and prints the corresponding PGM image to the console, and the remaining logic in main (which should just need to set up variables, and then call the two functions you just made).

By changing the signature for `main` to take Command Line Arguments, and then adding a simple test, enhance your code so that if it is run with the option “-h” on the command line, rather than printing the PGM file as output, it instead prints a few lines describing its purpose.