Crash Introduction to C Programming

1 Introduction

Many computer programs in use today were written in the C programming language, or its more complex younger sibling C++. In fact, most of the code for the most familiar operating systems (Windows, MacOS, and Linux/Unix) is written in C or some varient of it. The development of C occurred in tandem with Unix in the early 70's at Bell Labs. The name "C" came about because the language was the successor to two prior attempts at building a programming language, which were creatively named "A" and "B."

At its core, C is a very basic language, consisting of a small set of basic data types and commands that use an easy-to-learn syntax. To extend the power of C, there are a tremendous number of libraries that add more complex functions, such as reading and writing strings, files, and mathematical functions. Without these libraries, getting anything done in C would take a huge amount of coding. You will quickly come to think of these libraries as part of C, which for all practical purposes they are. The main stumbling block is knowing what functions exist, and in which library they reside. Fortunately, for basic scientific programming we will need only to use a few libraries, with which you will quickly become familiar.

This document is far from comprehensive, and is intended to provide only the basic elements of C that you will need for this course. There are an innumerable number of sources you can explore to learn C, and, fortunately, there are many that are freely accessible on the web. I encourage you to consult these if you need help:

- http://publications.gbdirect.co.uk/c_book/ – this is a complete online version of *The C Book*, a pretty good introductory text published first in 1991.
- http://www.cprogramming.com/tutorial.html
- http://randu.org/tutorials/c/

2 Basic Structure of a C Program

Before we cover the data types, commands and syntax of C (though some of it necessarily appears in this section), we first want a big picture view of how a C program is structured. Generally, a C program consists of a collection of subroutines and functions, which may be stored in a single file, or spread across multiple files. In all cases, when the source code is compiled and executed, the execution will start with a subroutine call `main`; for any C program to function, this routine must exist. Let's illustrate by an example:

```c
/* Anything bracketed this way is a comment.
   Compilers will ignore this text. */

/* The library headers we need for the functions we will call */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Other preprocessor directives, if needed */
#define MY_FAV_NUMBER 32768

/* Function declarations */
double my_function(double p);

/* the main routine where execution always starts */
int main(int argc, char **argv)
{
  /* variable declarations */
  int i, n, m;
  double p,q;

  /* begin the executable statements */
  m = 13;
  n = MY_FAV_NUMBER;
  i = n/m;
  p = (double)n/(double)m;

  /* here is a function call that returns a value */
  q = my_function(p);

}

/* Here is the function that main calls;
   it is only executed if called! */
double my_function(double p)
{
  double result;

  result = p*p*p;

  return(result);
}
```

This little example does not really do much. In particular, it would never give any feedback to the user if ran, so it is of little practical use, but hopefully shows the basic structure of every code we will write this semester.

A few comments on style should be made at this point. Any discussion of style is subjective, but if we agree that clarity of the code should be a priority (at least for this course), there are some basic guidelines we should follow. For example, notice that in `main` I have mixed the use of basic commands and functions calls. Such mixing is unavoidable to some degree, but you should try to minimize mixing, so that the `main` routine consists primarily of calls to functions or subroutines that do the work. Note there is no operational difference between a function and subroutine. By naming these subroutines in a useful manner, a reader of your code should be able to follow the logic of your program without ever looking at the technical details of you code.

For example, if you were trying to evolve the equations of motions of a gravitational system, `main` might make a series of call to functions/subroutines like `initialize_planet_positions`, `initialize_planet_velocities`, which could be followed by a loop containing commands like `calculate_forces`, `update_positions`, `update_velocities`. In this way you provide a clear indication of the logic of your code. Similarly, it helps to make your variable names clear. For example, a good variable to hold the positions my be called `position`.

There is disagreement on how wordy one should be. For example, people often use a single letter variable (such as `i`) for a loop counter. Others argue for clarity you should use a variable name like `loop_counter`. Given the trivial nature of the variable, the single letter variable will suffice for clarity in most cases, and is simple to use or modify. However, please be wary of over-use of such simplifications. In the end, the choice is your own, but keep in mind the philosophy of making the code easy to follow.

Obviously there are a lot of details to explain even in this simple example, and we will discuss these details in the remainder of the document, and supplement with further examples.

3 Syntax

3.1 Comments

Anything between the symbols `/* */` will be treated as a comment and ignored by the compilers. This is the traditional comment indicator. Many compilers also respect the use of `//` as an indication that anything until the next line should by considered a comment. To be safe, you are best using the traditional comment symbol.

3.2 Character Set

You can use more or less all the keys on your keyboard. Any variable or function you reference will be delineated by spaces. Hence you cannot have a space in a variable or function name. The common way to provide clarity is to use a _ to separate words. I used this in my example `calculate_forces`. Moreover, some characters have special meaning, like `+`, `-`, `*`, and `/`. These are the basic mathematical operations and hence cannot be used in variable or functions names. Additionally, {} , `()`, `:`, and `;` are reserved. Please refer to one of the online manuals for a complete listing of the special characters – but 98% of the time, avoiding the basic math symbols and punctuation characters will keep you out of trouble. The one place where people tend to forget this is trying to use a hyphen to separate words, like `calculate-forces`. This would not work as it would be interpreted as trying to take the difference between variables `calculate` and `forces`.

3.3 Blank Spaces and Line Termination

Except for the preprocessor directives (*i.e.* lines that start with `#`), C does not care much are about the number of blank spaces or carriage returns. There is no difference between 1 space and 100 blank spaces with 12 carriage returns. As a result, a single instruction in C can extend over many lines if you wish. To terminate the instruction, you use the semicolon `;`. Given these facts, if you neglect to put a semicolon at the end of a line where one should be, it will likely cause big problems (a.k.a. a bug), since the two lines would be considered part of the same instruction.

Note that since you can leave as much "white space" as you like, you can make things look very strange if you wish. For clarity, this is a bad idea. However, using additional white space to indent code can be very helpful to indicate the hierarchy that exists within the code – for example indenting the contents of a loop. Note that some text editors (including emacs and vim) understand C syntax, and will automatically indent code as you work. This can be very handy.

3.4 Preprocessor Directives

Preprocessor directives are examined by the compiler before the compiler "processes" the compilation; hence the name! Preprocessor directives are things like including the header files for the libraries. Preprocessor directives always start with the `#` symbol, and do *not* use a semicolon at the end. For example,

```
#include <stdio.h>
#define MY_FAV_NUMBER 32768
```

3.5 Function Declarations

The declaration of a function is just that – you declare it exists. With the exception of the special function `main`, you should do this for all functions you use. A function declaration consists of three mains parts:

(i) the data type of the function,

(ii) the name of the function, and

(iii) the variables the function will need.

The declaration should be terminated by a semicolon `;`

3.6 Functions

Having declared a function you eventually will write the function itself! The first line of the function will be identical to the declaration, except that you will not have a semicolon at then end. Instead, you will have the code of the function contained between `{ ... }` . For example:

```
int my_function(int var1, double var2)
{
        /* insert code here */
}
```

Note that you can use the curly braces { and } as much as you like to set apart code.

A special note on the arguments of the `main` function: `main` always takes two arguments, specifically `main(int argc, char **argv)`. `argc` is the number of arguments that were given on the command line when the program was started, and `argv` is an array of strings that contain the actual text command used to launch the program.

4 Data Types

In this section we describe the basic data types for the variables and functions. Like the rest of the document, this list is not comprehensive, but should cover the important types you will need for most computational applications.

4.1 Integer Numbers: `int`, `long int`, and `short`

If you need an integer variable, you will normally use type `int`. Thus an integer declaration should looksomething like:

```
int my_int_variable;
```

Note the terminating semicolon. You can also declare multiple integer variables on a line by separating them with a comma. The other two types – `short` and `long int` – are used if you only need small numbers (`short`) or if you need larger numbers (`long int`) than `int` can handle. The exact values of the maximum value the types can handle depends on machine and compiler, but in most cases `int` is a good choice.

It is important to realize that integers are always integers! Similarly, mathematical operations between integers result in an integer answer. This is particularly important for division.

For example, integer division `6/3 = 2`, but `5/3 = 1`, or `1/2 = 0`. If you ever need a decimal number do not use integers. However, you can force decimal evaluation by "typecasting". Specifically, if you have two integer variables `i` and `j`, you can treat them as decimals by doing `(double)i/(double)j` – where `double` is the next data type we discuss.

4.2 Real Numbers: `double`, `long double`, and `float`

`double` is the most commonly used variable for storing real numbers. Like `int`, `double` has variants that affect the largest size number that can be stored (which also depends on machine and compiler), but in most cases `double` should be fine.

As you would expect, mathematical operations between `double` variables results in a `double` answer. Where you must be very careful is when you mix variables of type `double` and `int`. Depending on the use and the compiler, you results may be of either `int` or `double` type! In general, you know the type of the answer you want, so it is best to be explicit by typecasting. For example:

```
/* declare variables */
int m, n;
double p, q;

/* insert code that assigns values... */

```

```
p = m/n;                        /* scary -- is this integer
                                   or double arithmetic? */
p = (double)m/(double)n;        /* ahh, much better */

q = n*p;                        /* again scary, mixing types! */
q = (double)n*p;                /* better, we are specific */

q = 2*p;                        /* careful!  2 is an integer! */
q = 2.*p                        /* better -- 2. has a decimal,
                                   and thus is real */
```

Pay particular attention to the last 2 lines. The number "2" is an integer and will be treated as such, while the number "2." or "2.0" is a real number. This is an easy place to make mistakes!

4.3 Characters: `char`

The variable type used to store characters is `char`. Note that a `char` variable holds only a single character. If you need more than one character – such as to store a word – you either need more than one variable, or more likely you need an array of characters.

4.4 Arrays

You can make an array from any variable type with a simple declaration like

```
double myArray[10];
```

This would declare an array of type `double` called `myArray` with 10 elements. The elements are numbered from 0 through 9 – *not* 1 through 10. Hence to assign the first element, you would type something like

```
myArray[0] = 14.0;
```

Note that the number of elements in an array must be explicitly indicated; you cannot specify a variable number of elements. This requires dynamically allocated arrays, which we will not discuss in this introduction.

To use good form, it is best to use a `#define` to avoid having seemingly random numbers floating around in your code. For example,

```
#define ARRAY_SIZE 10

int main(int argc, char **argv)
{
   double myArray[ARRAY_SIZE];
}
```

4.5 `void` Data Type

What if you have a function that is not returning a value? This is the reason for the `void` type. Generally, you will never declare a variable to be of this type. Normally only functions with no return value have type `void`.

4.6 Pointers

Pointers are a subject that can cause great confusion and trouble in C. From a practical computational mindset, it is somewhat unfortunate that they exist, since they can cause so much trouble. Here we will limit ourselves to only the most basic use when it simply cannot be avoided.

Crudely speaking, for every variable, there is a pointer to that variable. The pointer is the memory location of the variable. If we have a variable `bob`, we identify the memory location (or pointer) as `&bob`.

Why should you ever care or need this? The reason is that when you pass a single variable (like `bob`) to a function, C makes a copy of the variable and sends that copy to the function. As a result, if you change the value of the copy in the function, nothing happens to the original variable. This is fine, but what if we want changes to `bob` made by the function to be preserved? Then we must pass to the function the memory location, or pointer to the variable, *i.e.* `&bob`.

Since a pointer is a memory location, how can we access the contents of that memory location if we actually want to work with it? If we have a pointer called `alice`, then the contents of the pointer are given by `*alice` – not to be confused with multiplication.

We will avoid this as much as possible, but there will be a few occasions where it comes up. Note that this is never an issue with arrays. If you pass an array to a function and make changes, those changes are preserved. This is because arrays are themselves actually pointers — but this is a detail we will not discuss. Visit an outside resource if you want the ugly details on these things.

4.7 File Pointers

The data type `FILE` is used to access files. The use of this will be discussed in the Input/Output section.

4.8 Initialization

The initialization of variables depends on your compiler; some compilers may set numbers to zero by default, while other may simply grab the space in memory, and whatever is there is what your get! Hence it is a good idea to always be explicit and assign values.

5 Operators and Basic Commands

5.1 Operators

You are hopefully already familiar with basic mathematical operators `+`, `-`, `*`, and `/`. There are several other operators you will commonly need.

Assignment Operators: Fairly obvious, but `=` is for assignment. It is *not* for comparing if two things are equal. It will always assign! Other important assignment operators are:

`+=`	shortcut for adding to; eg. `a += 4;` is the same as `a = a+4;`
`-=`	same as above, but for subtraction
`++`	shortcut for adding `1`; eg. `a++;` is the same as `a += 1;` useful for loops
`--`	same as `++`, but for decrementing by `1`.
`*=`	same as `+=`, but for multiplication
`/=`	same as `+=`, but for division

Comparison Operators:

`==`	equal to?
`!=`	not equal to?
`<`	less than?
`<=`	less than or equal to?
`>`	greater than?
`>=`	greater than or equal to?

Logic Operators:

`&&`	Logical AND
`||`	Logical OR

These logic operators should not be confused with the bitwise operators `&` and `|`. We will not discuss bitwise operators, or some of the other more esoteric operators that you should not need here.

These logic operators normally come into play when you want to combine conditions, such as

```
(a<b) && (a>c)
```

5.2 `if` construction

This is the basic statement for conditionals. If can also be coupled with `else if` and `else`, but they are not required. An example of the syntax is:

```
if (a*b > c)
{
   /* insert code */
}
else if (a*b > d)
```

```
{
    /* insert code */
}
else
{
    /* insert code */
}
```

Note that there is no limit on the number of `else if` statements you can combine.

5.3 `for` loops

This is the most common construction for a task you wish to repeat. It works when you know exactly how many times the task must be done. For example, to sum up the elements of the array we declared in the array section:

```
sum = 0.;
for (i=0; i<ARRAY_SIZE; i++)
{
    sum += myArray[i];
}
```

This performs a loop over values of `i` from 0 up to `ARRAY_SIZE-1`. Note that want to stop our loop before `ARRAY_SIZE` since the elements of the array have indices 0 through `ARRAY_SIZE-1`. Because we use the `i++` notation, the value of `i` only changes at the end of each cycle of the loop. Had we used `++i`, the value would change at the beginning. I suggest using the first method and forget about the second, so that you avoid confusion. Note that we explicitly zeroed my variable `sum`, since I am not sure about its initial value.

5.4 `while` and `do while` loops

`while` and `do while` loops are ideal for situations where you do not know how many iterations are needed. The only difference between the two statements is that `while` will check the condition at the beginning of the loop, and hence the loop may not execute if the condition is not met; `do while` will check at the end of the loop, and hence there will always be at least one iteration. In most cases, the `while` loop suffices. Example syntax is

```
while(i<ARRAY_SIZE)
{
    /* insert code here */
    i++;
}
```

For the `do while`:

```c
do
{
   /* insert code here */
   i++;
}
while(i<ARRAY_SIZE);
```

Note that you must be sure that you are doing something inside the loop that will affect your conditional test. Otherwise you have an infinite loop, and no one likes an infinite loop.

6 Library Functions

6.1 Input and Output

While unimportant from an algorithmic point of view, input/output (I/O) is obviously extremely important from a practical point of view. We will discuss only text input and output. We will start with output since it is slightly easier.

6.1.1 Output

Screen Output: The most basic output is to print to the screen. This is achieved by means of the `printf` function. In its most basic use, you can print a string the the screen, for example

```
printf("Hello world!\n");
```

You will notice the `\n`; this means to add a newline, so that the next print will occur on a new line. Without `\n`, a subsequent print would appear immediately following the first on the same line.

Naturally, there are times when you will want to print out information about variables in your program. This is also best illustrated by example:

```
1 printf("The value of a = %d\n", a);
```

This will print the value of `a` where you see the `%d`. In this example, the variable `a` is assumed to be of type `int`. The letter you use in formatting the print statement depends on the variable type. The most common you will encounter are:

`%d`	`integer` variable
`%lf`	`double` variable
`%le`	`double` variable, using scientific notation
`%f`	`float` variable
`%c`	`char` variable
`%s`	a string, *i.e.* an array of `char`

You can also easily print many variables in a single print statement, for example:

```
1 printf("The value of a = %d; Here is an array element: %lf\n",
        a, myArray[3]);
```

It should be clear that here `a` is of type `int` and `myArray` is our array of type `double`. With the exception of strings (`character` arrays), you can only print out one element of the array at a time. Also notice my use of white space – the carraige return is irrelevant!.

File Output: Output to a file works almost the same way, but you use the function `fprintf`, which requires one additional argument, the pointer to the file that you

are writing to. Hence you also need to know how to open a file pointer, which requires the functions `fopen` and `fclose`. As always, we use an example:

```
FILE *fp;

fp = fopen("myfile", "w");   /* opens file for writing */

fprintf(fp, "I am writing to a file!\n'');
fprintf(fp, "And now I print a string: %s\n", mystring);

fclose(fp);
```

The `"w"` in the `fopen` statement is for opening the file in write mode. This will open a new file, and if a file with that name exists, it will overwrite it. If you want to append to an existing file, open using `"a"`, append mode. Also, the string `"myfile"` can be replaced with a string variable (character array) that holds the desired name.

6.1.2 Input

Input is slightly more tricky because it requires the use of a pointer. Fortunately, if you stick to the same basic method, it is not hard.

Screen Input: This covers the case where you want a user to enter input information. There is an analogous function to `printf`, called `scanf`. They are normally used together, so that you can communicate what you are expecting from the user, for example:

```
printf("Enter an integer value:" );
scanf("%d", &a);
```

In this example, `a` in an `integer` variable. Note the use of the `&` symbol. Since `scanf` is going to modify the value of `a`, it needs the memory address of `a`. You must remember to include the `&` or your code will bomb at that point when you execute it!

Like `printf`, you can use `scanf` to read multiple values in a single line if you wish.

File Input: Just as we can print to a file, we can scan from a file. Not surprisingly, the name of the function we use is `fscanf`. The use of `fscanf` is a logical extension of what we have introduced so far. An example use is:

```
char myString[ARRAY_SIZE];
double myArray[ARRAY_SIZE];
FILE *fp;

fp = fopen("myfile", "r");        /* opens file for reading */

fscanf(fp, "%d", &a);             /* read an integer */
fscanf(fp, "%lf", &myArray[0]);   /* read a double into
                                     array element 0 */
```

```
10 fscanf(fp, "%s", myString);      /* read a string */

12 fclose(fp);
```

The file must have exactly the expected items for this to work. Notice the special case when I read a string. Since `myString` is a `char` array, we do not need to use `&`, we simply give the variable name. We cannot play this trick with the `double` array.

To be careful, it is helpful to be sure that a file exists before trying to open it for reading. Such care is not required for writing, unless you want to avoid overwriting a file. When reading, it is best to replace the simple `fopen` command with the more complex construction:

```
if ((fp = fopen("myfile", "r")) == NULL){
2   printf("Error opening file\n");
    exit(1);
4 }
```

6.2 Mathematical Functions

The mathematical functions are named more or less as you would expect. For example, the cosine function is called `cos` and it takes as an argument a `double` variable. If you are unsure of a function's name name or arguments, Pelles C's HELP menu has information on many common functions. This applies to I/O functions as well. Of course, Google is always helpful as well.