
Chapter 1

Systems Programming and C Basics

What is in This Chapter ?

This first chapter of the course explains what **Systems Programming** is all about. It explains how it is closely linked to the operating system. A few basic tools are explained for use with the **gcc compiler** under a **Linux Ubuntu** environment running within a **VirtualBox** application. It then introduces you to the **C programming language** in terms of the basic syntax as it is compared to JAVA syntax. A few simple programs are created to show how to **display information**, compute simple **math calculations**, deal properly with **random numbers**, getting **user input**., using **arrays** and calling **functions**.



1.1 Systems Programming and Operating Systems

In COMP1405 and COMP1406, you developed various programs and applications. The goal was to write programs that “accomplished” something interesting in that it provided a service for the user ... usually resulting in an “app” that interacted with the user. Examples of common applications are internet browsers, word processors, games, database access programs, spreadsheets, etc.. So, what you have been doing in your courses has been:

Applications Programming *is the programming of software to provide services for the user directly.*

Systems Programming, on the other hand is different. It has a different focus ... and can be defined as follows:

Systems Programming *is the programming of software that provides services for other software ... or for the underlying computer system.*

So, when you are doing systems programming, you are writing software that does not typically have a front-end GUI that interacts with the user. It is often the case where the software runs “behind-the-scenes”... sometimes as a process/thread running in the background. Some examples of systems programs are:

1. **Firmware** (e.g., PC BIOS and UEFI).
2. **Operating systems** (e.g., Windows, Mac OSX, GNU/Linux, BSD, etc...).
3. **Game Engines** (e.g., Unreal Engine 4, Unity 3D, Torque3D)
4. **Assemblers** (e.g., GNU AS, NASM, FASM, etc...).
5. **Macro Processors** (e.g., GNU M4).
6. **Linkers and Loaders** (e.g., GNU ld which is part of GNU binutils).
7. **Compilers and Interpreters** (e.g., gcc, python, Java VM).
8. **Debuggers** (e.g., gdb).
9. **Text editors** (e.g., vim).
10. **Operating system shell** (e.g., bash).
11. **Device Drivers** (e.g., for Bluetooth, network cards, etc..)

Systems software involves writing code at a much more lower level than typical application software. It is often closely tied to the actual hardware of the machine that it is running on. In fact, it often uses the operating system directly through system calls. So, when you write systems software, it is important to have a good understanding of the machine that it will be running on.

Applications programming is at a higher level than systems programming, and so it is closer to the way we think as humans. It is more natural and can make use of high level programming languages and specialized libraries.

System software is the layer between the hardware and application software. It can deal directly with the hardware and usually controls it. It is therefore, more naturally written in a lower level programming language. In general, it provides “efficient” services to applications.



The goal of writing systems software is to make **efficient** use of resources (e.g., computer memory, disk space, CPU time, etc..). In some cases, performance may be critical (e.g., like a fast game engine). In fact, it is often the case that small improvements in efficiency can save a company a lot of money.

Therefore, in this course, we will be concerned about writing efficient code ... something we didn't focus on too much in COMP1405/1406.

In this course, we will also be trying to get a better grasp of the computer's operating system.

*An **Operating System** is system software that manages computer hardware and software resources and provides common services for computer programs.*

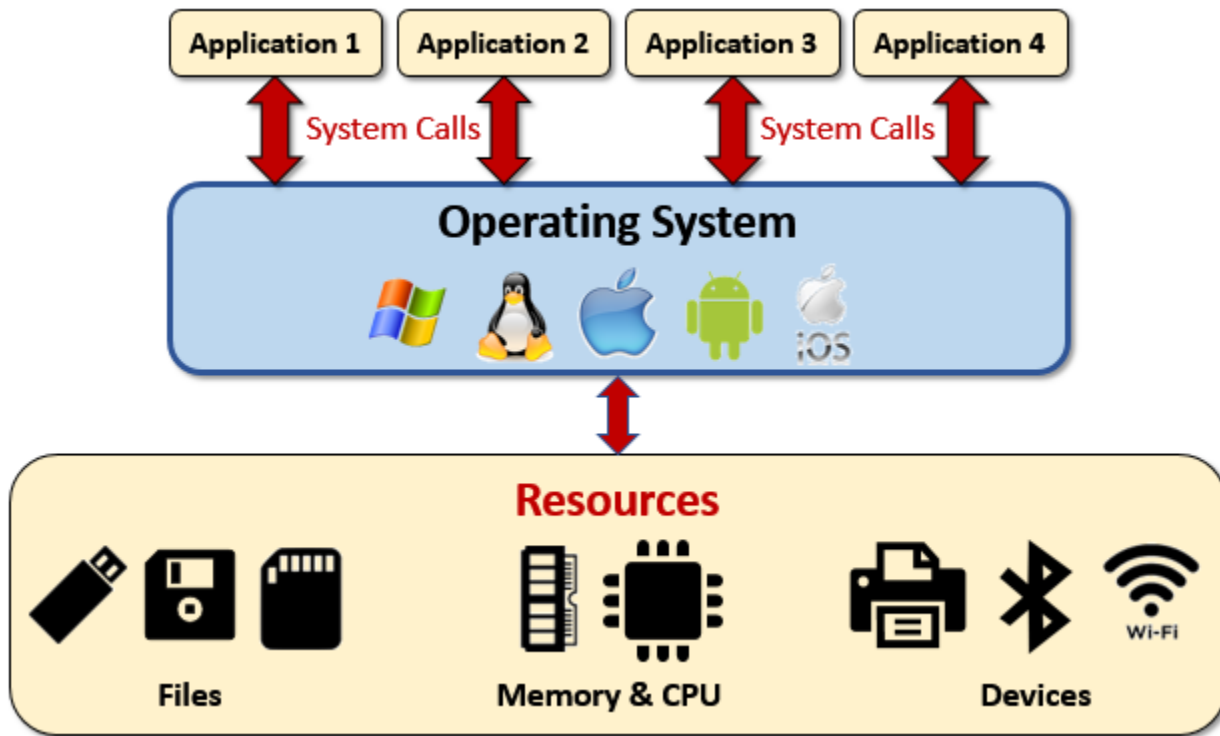
Operating systems are the layer of software that sits between the computer's hardware and the user applications. It is considered the "boss" of the computer ... as it manages everything that is going on visibly, as well as behind the scenes.

Some operating systems provide time-sharing features that schedule tasks at various times depending on how the computer's resources (e.g., memory, disk storage, printers and devices) are allocated at any different time.

There are various operating systems out there of which you may have heard of:

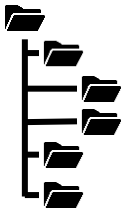
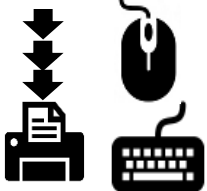
- Windows
- Mac OSX
- Unix
- Linux
- Android
- Chrome OS




The operating system acts as the intermediary between the applications and the computer's resources. Applications perform system calls to gain access to the resources:



So, in a sense, the operating system provides *services* for the applications. It provides some core functionality to users as well as through programs.

Here is some of the functionality that is provided by the operating system, although there is a lot more than this:

| | |
|---|---|
| <p>File I/O</p> <ul style="list-style-type: none"> • provides file system organization and structure • allows access to read and write files • provides a measure of security by allowing access permissions to be set |  |
| <p>Device I/O</p> <ul style="list-style-type: none"> • allows communication with devices through their drivers • e.g., mouse, keyboard, printer, bluetooth, network card, game controllers... • manages shared access (i.e., printer queue) |  |

| | |
|--|---|
| <h2>Process Management</h2> <ul style="list-style-type: none"> allows the starting/stopping/alteration of running executables can support multitasking <ul style="list-style-type: none"> (i.e., multiple processes running concurrently) allocates memory for each process <ul style="list-style-type: none"> needed for program instructions, variables & keeping track of function calls/returns. |  |
| <h2>Virtual memory</h2> <ul style="list-style-type: none"> provides memory to every process <ul style="list-style-type: none"> dedicated address space, allocated when process starts up appears (to the process) as a large amount of contiguous memory but in reality, some is fragmented in main memory & some may even be on disk programs use <i>virtual</i> memory addresses as opposed to <i>physical</i> memory addresses |  |
| <h2>Scheduling</h2> <ul style="list-style-type: none"> allows applications to share CPU time as well as device access |  |

Operating systems vary from one to another.



The **Windows** operating system:

- expensive** when compared to some others
- has limitations that make it **harder to work with**
- is a **closed system** that has very **restricted access to OS functions**
- was designed to make the computer simpler to use ... primarily for business users and the average home user who are not computer savvy. Hence, the system was designed to allow **access at a higher level**.



Unix-based operation systems:

- free ... **open source**
- a more open system that allows **broad access to OS functions**
 - "root" or *super-user* can do **anything** (extremely dangerous)
- family of options** (Linux, Solaris, BSD, Mac OS X, many others)
- OS of choice for complex or scientific application development

The language of choice for programming Unix-based operating systems is **C**.

- closer to hardware
- used to write Unix OS, device drivers
- very fast, with negligible runtime footprint

1.2 Tools for Systems Programming

We will now discuss 4 tools that are essential for systems programming:

Shells

*A **Shell** is a command line user interface that allows access to an operating system's services.*

A shell allows the user to type in various commands to run other programs. It also serves as a command line interpreter. Multiple shells can be run at the same time (in their own separate windows).

In Unix, there are three major shells:

- **sh** - Bourne shell
- **bash** - Bourne-again shell (default shell for Linux)
- **cs**h - C shell



The shells differ with respect to their command line shortcuts as well as in setting environment variables.

A shell allows you to run programs with **command line arguments** (i.e., parameters that you can provide when you start the program in the shell). The parameters (a.k.a. arguments or options) are usually preceded by a dash **-** character.

There are some common shell commands that you can make use of within a shell:

- e.g., **alias**, **cd**, **pwd**, **set**, **which**

There are some common system programs that you can make use of within a shell:

- e.g., **grep**, **ls**, **more**, **time**, **sort**

You can even get some help by accessing a kind of “user manual” through access of what are called **man pages** with the **man** command.

We will make use of various shell commands and programs throughout the course.

Text Editors

A **Text Editor** is a program that allows you to produce a text-based file.

There are a LOT of text editors out there. They are basic. Some are built into standard Operating System packages. There is a big advantage to knowing how to use one of these common editors ... as they are available on any machine.

It is good to choose one and “stick with it”, as you will likely develop an *expertise* with it.

There are some common ones such as: **vi/vim**, **emacs** and **gedit**. You will need to use one for writing your programs in this course and for building **make files** (discussed later).



There is a bit of a learning curve for these editors, as they all require you to use various “hot keys” and commands in order to be quick and efficient at editing. The commands allow you to write programs without the use of a mouse ... which is sometimes the case on some systems when you don't have device drivers working/installed.

Compilers

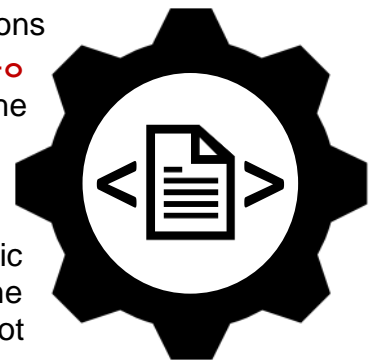
A **Compiler** is computer software that transforms source code written in one programming language into another target programming language.

In this course, we will make use of the **GNU** compiler.

GNU is a recursive acronym for "GNU's Not Unix!". It was chosen because GNU's design is Unix-like, but differs from Unix by being free software and containing no Unix code.

The command for using the compiler is **gcc**. There are many options that you can provide when you run the compiler. For example, **-o** allows you to specify the output file and **-c** allows you to create the object code. You will learn how to use these command options (and others) as the course goes on.

Compilers produce code that is meant to be run on a specific machine. Therefore, your code **MUST** be compiled on the same platform that it runs on. Linux-compiled code, for example, will not run on Windows, Unix nor MacOS machines.



Debuggers

A **Debugger** is a program that is used to test and debug other programs.

There are two main advantages of using a debugger:

- It allows you to control the running (i.e., execution) of your code.
 - can **start/stop/pause** your program
 - good to slow things down in time-critical and resource-sharing scenarios
- It allows you to investigate what is happening in your program
 - can **view your variables** in the midst of your program
 - can **observe the control flow** of your program to find out whether certain methods are being called and in what order



The goal is always to debug ... to find out where your program is going wrong or crashing.

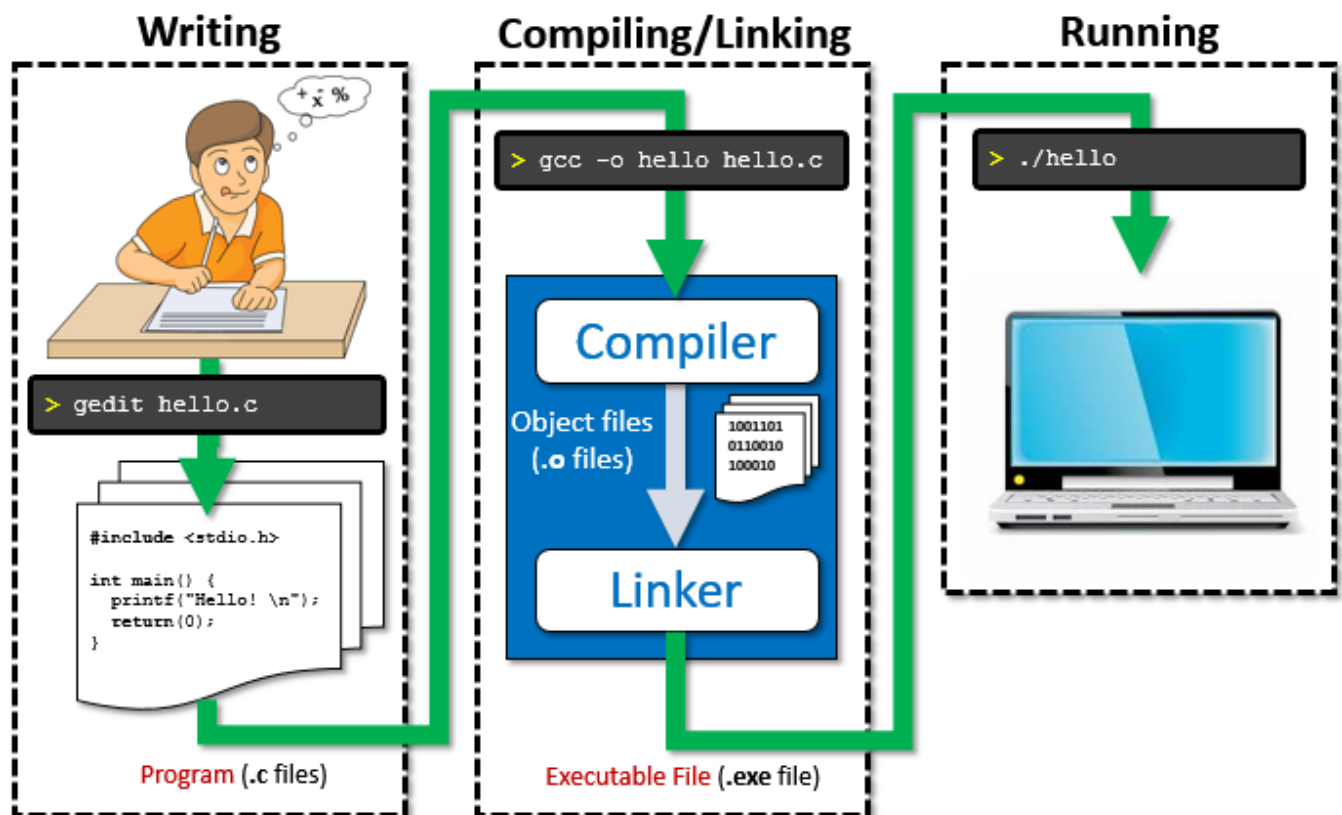
The command for using the compiler is **gdb**. You use it to run the program ... after it has been compiled. However, to use it, the program must have already been compiled with the **-g** option. When using the debugger, there are various commands that you can apply such as **run**, **break**, **display**, **step**, **next** and **continue**.

1.3 Writing Your First C Program

The process of writing and using a C program is as follows:

1. **Writing:** write your programs as `.c` files.
2. **Compiling:** send these `.c` files to the `gcc` compiler, which will produce `.o` object files.
3. **Linking:** the `.o` files are then linked with various libraries to produce an executable file.
4. **Running:** run your executable file.

Getting your C programs to run requires a little more work than getting a JAVA program to run. As with JAVA, your source code must be compiled. Instead of producing `.class` files, the C compiler will produce `.o` files which are called **object files**. These object files are “linked” together, along with various library files to produce a runnable program which is called an **executable file**. This executable file is in **machine code** that is meant to be run on a specific computer platform. It is not portable to other platforms.



Our First Program

The first step in using any new programming language is to understand how to write a simple program. By convention, the most common program to begin with is always the "hello world" program which when run ... should output the words "Hello World" to the computer screen. We will describe how to do this now.

In this course, you will use either **emacs**, **vim** or **gedit** to write your programs. I will be using examples that make use of **gedit**. Once you have your Terminal window open (you will learn how to do this in the first tutorial), then you start up your editor by specifying the editor name followed by the name of the file that you want to write ... in this case it will be **helloWorld.c**.

You use the **&** symbol at the end of the command line to indicate that you want to run the editor in the background. This allows you to keep the editor open while you are compiling and running/testing your code. If you don't use the **&** character, then you must close the **gedit** editor in order to continue to work again in the Terminal window. Of course, you can always work with a second Terminal window if you'd like, but it is easiest to simply run the editor in the background.

```
student@COMPBase:~$ gedit helloWorld.c &
```

Here is a window that shows the editor with some code in it:



```

#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}

```

When compared to **JAVA** (shown on the right), you will notice some similarities as well as some differences.

Below, on the left, is our first **C** program that we will write:

| C program | JAVA program |
|--|---|
| <pre> #include <stdio.h> int main() { printf("Hello world\n"); return 0; } </pre> | <pre> public class HelloWorldProgram { public static void main(String[] args) { System.out.println("Hello World"); } } </pre> |

Here are a few points of interest in regard to writing **C** programs:

1. The `#include <stdio.h>` on the first line tells the compiler to include a **header file**.

A **Header File** is a file containing C declarations and macro definitions to be shared between several source files.

This this case, `stdio.h` is the name of the header file. This is a standard file that defines three variable types, several macros, and various functions for performing input and output. We need it here because we will be printing something out to the screen using `printf`.

2. Unlike JAVA, we don't need to define the name of a class. So we begin right away with the `main()` function. There are no **public/private/protected** access modifiers in **C**, so we leave those out. You will notice that the `main()` function returns an `int`, whereas in JAVA it was `void`. Also, we are not required in **C** to specify that there will be command-line arguments ... so we do not need to declare that as a parameter to the `main()` function.
3. The procedure for printing is simply `printf()`, where we supply a string to be printed ... and also some other parameters as options (more on this later). If we want to ensure that additional text will appear on a new line, we must make sure that we include the `\n` character inside the string.
4. The `main()` function should return an integer. By convention, negative numbers (e.g., -1) should be returned when there was an error in the program and `0` when all went well. However, by allowing a variety of integers to be returned, we can allow various **error codes** to be returned so as to more precisely what had gone wrong.
5. Finally, notice that **C** uses braces and semi-colons in the same way that JAVA does.

So ... to summarize, our **C** programs will have the following basic format:

```
#include <...>
#include <...>
#include <...>

int main() {
    _____;
    _____;
    _____;
}
```

You should ALWAYS line up ALL of your brackets using the **Tab** key on the keyboard.

Now that the program has been written, you can compile it in the same Terminal window by using the `gcc -c` command as follows:

```
student@COMPBase:~$ gedit helloWorld.c &
student@COMPBase:~$ gcc -c helloWorld.c
student@COMPBase:~$
```

This will produce an *object file* called **helloWorld.o**. You can view the file by using the **ls** command:

```
student@COMPBase:~$ gedit helloWorld.c &
student@COMPBase:~$ gcc -c helloWorld.c
student@COMPBase:~$ ls
helloWorld.c  helloWorld.o
student@COMPBase:~$
```

Then, we need to link it (with our other files and standard library files) to produce an executable (i.e., runnable) file. We do this as well with the **gcc -o** compiler as follows:

```
student@COMPBase:~$ gedit helloWorld.c &
student@COMPBase:~$ gcc -c helloWorld.c
student@COMPBase:~$ ls
helloWorld.c  helloWorld.o
student@COMPBase:~$ gcc -o helloWorld helloWorld.o
student@COMPBase:~$
```

After the **-o** is the name of the file that we want to be the runnable file. We follow it with a list of all object files that we want to join together. In this case, there is just one object file. This will produce an *executable file* called **helloWorld**. You can view the file by using the **ls** command:

```
student@COMPBase:~$ gedit helloWorld.c &
student@COMPBase:~$ gcc -c helloWorld.c
student@COMPBase:~$ ls
helloWorld.c  helloWorld.o
student@COMPBase:~$ gcc -o helloWorld helloWorld.o
student@COMPBase:~$ ls
helloWorld  helloWorld.c  helloWorld.o
student@COMPBase:~$
```

You can then run the **helloWorld** file directly from the command line, but we need to tell it to run in the current directory by using **./** in front of the file name:

```
student@COMPBase:~$ gedit helloWorld.c &
student@COMPBase:~$ gcc -c helloWorld.c
student@COMPBase:~$ ls
helloWorld.c  helloWorld.o
student@COMPBase:~$ gcc -o helloWorld helloWorld.o
student@COMPBase:~$ ls
helloWorld  helloWorld.c  helloWorld.o
student@COMPBase:~$ ./helloWorld
Hello World
student@COMPBase:~$
```

Notice that when you run your program, any output from the program will appear in the shell window from which it has been run. As a side point, you can link the files without first compiling. The linking stage will compile first by default:

```
student@COMPBase:~$ gcc -o helloWorld helloWorld.c
student@COMPBase:~$
```

1.4 C vs. Java

C code is very similar to JAVA code with respect to syntax. Provided here is a brief explanation of a few of the similarities & differences between the two languages. To begin, note that commenting code is the same in C as it is in JAVA:

Commenting in C and JAVA

```
// single line comment

/* a multiline comment
   which spans more
   than one line.
*/
```



Displaying Information to the System Console:

| | |
|-------------|---|
| JAVA | <pre>System.out.print("The avg is " + avg); System.out.println(" hours"); System.out.println(String.format("Mark is %d years old and weighs %f pounds", a, w));</pre> |
| C | <pre>printf("The avg is %d", avg); printf(" hours\n"); printf("Mark is %d years old and weighs %f pounds\n", a, w);</pre> |

Notice that the print statement is easier to use in C. In JAVA, things like integers and objects could be appended to strings with a + operator. We cannot do that in C.

Instead, we do something similar to the **String.format()** function in JAVA by supplying a list of parameters for the string to be printed. Inside the string we use the % character to indicate that a parameter is to be inserted there.



There are many possible *flags* that can be used in the format string for the **printf**. You will want to look them up. The general format for each parameter is:

```
%[flags][width][.precision][length]<type>
```

Here is a table showing what the various values may be for **type**:

| type | Description | Example | Output |
|------|------------------|---|-------------------|
| %d | integer | printf("%d", 256) printf("%d", -256) | 256 -256 |
| %u | unsigned integer | printf("%u", 256) printf("%u", -256) | 256 4294967040 |

| | | | |
|------------------------|-------------------------------------|--|------------------------------|
| %f | floating point (6-dec precision) | printf("%f", 3.14159265359) printf("%f", 314159265.359) | 3.141593 314159265.359000 |
| %g | floating point (exp. precision) | printf("%g", 3.14159265359) printf("%g", 314159265.359) | 3.14159 3.14159e+08 |
| %c | character | printf("%c", 65) | A |
| %s | string | printf("%s", "Hello") | Hello |
| %x %X | hexadecimal | printf("%x", 250) printf("%X", 250) | 0fa 0FA |
| %o | octal | printf("%o", 250) | 372 |

The **width** parameter allows us to specify the minimum number of “spaces” that the output will take up. We can use this **width** parameter to have things lined up in a table. If the **width** is too small, and the number has more digits than the specified **width** ... the **width** parameter will not affect the output.

Here are some examples:

```
printf("%5d", 256)           // 256
printf("%5f", 3.14159265359) //3.141593      ... No effect
printf("%5c", 65)           //   A
printf("%5s", "Hello")      //Hello

printf("%10d", 256)          //          256
printf("%10f", 3.14159265359) //   3.141593
printf("%10c", 65)           //          A
printf("%10s", "Hello")     //          Hello
```

The **precision** parameter works differently depending on the value being used. When used with floating point flag **f** it allows us to specify how many digits we want to appear after the decimal place. When used with floating point flag **g** it allows us to specify how many digits in total we want to be used in the output (including the ones to the left of the decimal):

```
printf("%2.3f\n", 3.14159265359); //3.142      ... rounds up
printf("%2.3g\n", 3.14159265359); //3.14
printf("%2.3f\n", 3141592.65359); //3141592.654
printf("%2.3g\n", 3141592.65359); //3.14e+06
```

When used with string flag **s**, the **precision** parameter allows us to indicate how many characters will be displayed from the string:

```
printf("%2.1s\n", "Hello"); // H
printf("%2.3s\n", "Hello"); //Hel
printf("%2.5s\n", "Hello"); //Hello
printf("%10.1s\n", "Hello"); //          H
printf("%10.3s\n", "Hello"); //          Hel
printf("%10.5s\n", "Hello"); //          Hello
```

When used with integer, unsigned integer, octal and hexadecimal flags **d**, **u**, **o**, **x**, **X**, the **precision** parameter allows us to indicate how many leading zeros will be displayed:

```
printf("%6.1X\n", 250); // FA
printf("%6.2X\n", 250); // FA
printf("%6.3X\n", 250); // 0FA
printf("%6.4X\n", 250); // 00FA
printf("%6.5X\n", 250); // 000FA
printf("%6.1d\n", 250); // 250
printf("%6.2d\n", 250); // 250
printf("%6.3d\n", 250); // 250
printf("%6.4d\n", 250); // 0250
printf("%6.5d\n", 250); // 00250
printf("%6.5d\n", -250); // -00250
```



When used with integer, unsigned integer, octal and hexadecimal flags **d**, **u**, **o**, **x**, **X**, the **flags** parameter also allows us to indicate how many leading zeros will be displayed. When used with numbers, the **0** flag allows leading zeros to be inserted and the **+** allows a plus sign to be inserted for positive numbers (normally not shown):

```
printf("%6d\n", 250); // 250
printf("%06d\n", 250); //000250
printf("%+6d\n", 250); // +250
```

When used with numbers or strings, the **-** flag allows everything to be left-aligned:

```
printf("%-6d\n", 250); //250
printf("%-+6d\n", 250); //+250
printf("%- .1s\n", "Hello"); //H
printf("%- .3s\n", "Hello"); //Hel
printf("%- .5s\n", "Hello"); //Hello
```



There are more options than this, but we will not discuss them any further. You may google for more information.

In **C**, there are 4 main primitive variable types (but **int** has 3 variations). Variables are declared the same way as in **JAVA** (except literal float values do not need the 'f' character).

| <i>Variables in C</i> | <i>Variables in JAVA</i> |
|--|------------------------------------|
| int days = 15; | int days = 15; |
| char gender = 'M'; | char gender = 'M'; |
| float amount = 21.3; | float amount = 21.3f; |
| double weight = 165.23; | double weight = 165.23; |
| char age = 19; | byte age = 19; |
| short int years = 3467; | short years = 3467; |
| long int seconds = 17102397834; | long seconds = 17102397834; |
| char hungry = 1; | boolean hungry = true; |

Interestingly, there are no **booleans** in the basic C language. Instead, any value which is non-zero is considered to be **true** when used in a **boolean** expression (more on this below).

Also, **char** is used differently since it is based on ASCII code, not the UNICODE character set. So the valid ranges of char is **-128** to **+127** (more on this later).

Fixed values/numbers are defined using the **#define** keyword in **C** as opposed to **final**. Also, the **=** sign is not used to assign the value. Typically, these fixed values are defined near the top of the program file. You should ALWAYS name them using uppercase letters with multiple words separated by underscore characters.

| <i>Fixed Values in C:</i> | <i>Fixed Values in JAVA:</i> |
|--|--|
| <pre>#define DAYS_IN_YEAR 365 #define RATE 4.923 #define NEWLINE '\n'</pre> | <pre>final int DAYS_IN_YEAR = 365; final float RATE = 4.923f; final char NEWLINE = '\n';</pre> |

In **C**, both **IF** and **SWITCH** statements work “almost” the same way as in **JAVA**:

| <i>IF statements:</i> | <i>SWITCH statements:</i> |
|--|---|
| <pre>if ((grade >= 80) && (grade <= 100)) printf("Super!\n"); if (grade >= 50) { printf("%d", grade); printf(" Passed!\n"); } else printf("Grade too low.\n");</pre> | <pre>switch (someIntegerVariable) { case 1: printf("Case1"); break; case 2: printf("Case2"); break; case 3: printf("Case3"); break; case 4: printf("Case4"); break; default: printf("Default"); }</pre> |

However, there are differences, since there are no **boolean** types in **C**. We need to fake it by using integers (or one-character bytes as a **char**). Consider this example in **C** and **Java**:

| <i>Booleans “faked” in C:</i> | <i>Booleans in JAVA:</i> |
|---|--|
| <pre>char tired = 1; char sick = 0; if (sick && tired) printf("I give up\n");</pre> | <pre>boolean tired = true; boolean sick = false; if (sick && tired) System.out.println("I give up");</pre> |

In the **C** example, **tired** is considered to be **true** since it is non-zero, whereas **sick** is considered to be **false**. When the **&&** is used, the result is always either **1** or **0**, indicating **true** or **false**. So, if both **sick** and **tired** were set to **1**, then (**sick && tired**) would result in **1**, not **2**. The same holds true for the **||** operator which is used for the **OR** operation.

The boolean negation character **!** will change a non-zero value to **0** and a zero value to **1**.

So ...

```
char tired = 5;
char sick = 0;

tired = !tired;    // tired will now be 0
sick = !sick;     // sick will now be 1
```

If it makes your code easier to read, you can always define fixed values for TRUE and FALSE that can be used in your programs:

```
#define TRUE 1
#define FALSE 0
```

Then you can do things like this:

```
char tired = TRUE;
char sick = FALSE;
```

There is also another kind of conditional operator called the *ternary/conditional* operator which uses the **?** and **:** character in sequence. It is a short form of doing an **IF** statement:

```
tired ? printf("tired\n") : printf("not tired\n");
```

It does the same as this:

```
if (tired)
    printf("tired\n");
else
    printf("not tired\n");
```

However, the **?:** is often used to provide a returned value that can be used in a calculation:

```
int hoursWorked = 45;
int bonus = (hoursWorked > 40) ? 25 : 0;
printf("%d\n", bonus);
```

In C, the **FOR** and **WHILE** loops work the same way:

| FOR loops: | WHILE loops: |
|---|--|
| <pre>int total = 0; for (int i=1; i<=10; i++) { total += i; } printf("%d\n", total);</pre> | <pre>int speed = 0; int x = 0; while (x <= 100) { speed = speed + 2; x = x + speed; printf("%d, %d\n", speed, x); }</pre> |

There is also a **DO/WHILE** loop ... in which the condition is checked at the end of the loop. This guarantees that the loop is evaluated at least once:

DO/WHILE loops:

```
int speed = 0;
int x = 0;
do {
    speed = speed + 2;
    x = x + speed;
    printf("%d, %d\n", speed, x);
} while (x <= 100);
```

In the first-year courses, we were interested in getting our code working. In this course, however, we will also be interested in getting our code to run quickly and efficiently. One example of improving efficiency is to make use of the **break** and **continue** statements when doing loops.

For example, here is code that determines all the prime numbers from 1 to 100,000:

```
#include <stdio.h>

int main() {
    int isPrime;

    printf("The prime numbers from 1 to 100,000 are:\n");
    for (int i=1; i<=100000; i++) {
        isPrime = 1; // Assume that i is prime
        for (int j=2; j<i; j++) {
            if (i%j == 0)
                isPrime = 0; // if factor found, not prime
        }
        if (isPrime)
            printf("%d\n", i);
    }
}
```



The code goes through all the numbers **i** from 1 to 100,000 and then checks all values from 2 to **i** to see if they divide evenly into **i**. If they do, then **i** is not a prime number.

The code took about **22** seconds to run in my virtual environment! However, we can speed this up drastically by using the **break** statement.

Consider the program logic. In the case where **i** is 100, for example ... as we search in the inner FOR loop, we find immediately that 2 divides into 100 evenly, so 100 is not prime. The code, however, will continue to search through remaining numbers from 3 through 99. But this is pointless, because we have already determined that 100 is not prime.

If we therefore insert a **break** statement when we find out that **i** was not prime, then we break out of that loop and don't check the remaining numbers from 3 through 99.

You can imagine how this cuts back on running time even more as **i** increases to **100,000,000!**

```

for (int i=1; i<=100000; i++) {
    isPrime = 1;
    for (int j=2; j<i; j++) {
        if (i%j == 0) {
            isPrime = 0;
            break;           // quit the inner loop right away
        }
    }
    if (isPrime)
        printf("%d\n", i);
}

```



As a result of this simple addition, the code runs in less than 3 seconds!!!

In a similar way, we can use the **continue** statement, not to *quit* the loop, but to go on to the next item in the **FOR** loop.

For example, consider having to search 500 resumes to hire a worker. You may be looking for a particular attribute (e.g., previous administrative experience). You may go through each application and browse it quickly. If you do not see the desired attribute right away, you may want to quickly move on (i.e., **continue**) to the next resume instead of spending a lot of time examining this candidate's resume any further. That would save you a lot of time...



```

for (int i=0; i<numResumes; i++) {
    if (/* this candidate has less than 2 years experience */)
        continue;
    // ... code to determine if there is administrative experience ...
    if (/* this candidate does not have administrative experience */)
        continue;
    // ... otherwise check the resume further...
}

```

In this course, you should keep an eye out for any opportunities to be able to speed up your code.

When dealing with mathematical expressions, logic statements and variable assignment ... C and JAVA are almost exactly the same. Notice how similar the code is:

```

a = 15;           // a assigned value of 15
b = a % 4;       // b assigned remainder of 3
b++;             // b is increased to 4
c = 15.3;        // don't need an f character as with JAVA
d = c / 3;       // d assigned value of 5.1

```



Here is a table showing the basic operators:

| Arithmetic Operators | Assignment Operators | Relational Operators |
|-----------------------------|-----------------------------|------------------------------|
| + - * / % ++ -- | = += -= *= /= | == != < > <= >= |
| Logical Operators | Bitwise Operators | Conditional Operators |
| && ! | ~ & ^ >> << | ?: |

The order in which expressions are evaluated is very similar to JAVA:

| Precedence | Operator | Description | Associativity | |
|------------|--------------|---|---------------|---------------|
| 1 | ++ -- | Suffix/postfix increment and decrement | Left-to-right | |
| | () | Function call | | |
| | [] | Array subscripting | | |
| | . | Structure and union member access | | |
| | -> | Structure and union member access through pointer | | |
| | (type){list} | Compound literal(C99) | | |
| 2 | ++ -- | Prefix increment and decrement | Right-to-left | |
| | + - | Unary plus and minus | | |
| | ! ~ | Logical NOT and bitwise NOT | | |
| | (type) | Type cast | | |
| | * | Indirection (dereference) | | |
| | & | Address-of | | |
| | sizeof | Size-of | | |
| | _Alignof | Alignment requirement(C11) | | |
| 3 | * / % | Multiplication, division, and remainder | Left-to-right | |
| 4 | + - | Addition and subtraction | | |
| 5 | << >> | Bitwise left shift and right shift | | |
| 6 | < <= | For relational operators < and ≤ respectively | | |
| | > >= | For relational operators > and ≥ respectively | | |
| 7 | == != | For relational = and ≠ respectively | | |
| 8 | & | Bitwise AND | | |
| 9 | ^ | Bitwise XOR (exclusive or) | | |
| 10 | | Bitwise OR (inclusive or) | | |
| 11 | && | Logical AND | | |
| 12 | | Logical OR | | |
| 13 | ?: | Ternary conditional | | Right-to-Left |
| 14 | = | Simple assignment | | |
| | += -= | Assignment by sum and difference | | |
| | *= /= %= | Assignment by product, quotient, and remainder | | |
| | <<= >>= | Assignment by bitwise left shift and right shift | | |
| | &= ^= = | Assignment by bitwise AND, XOR, and OR | | |
| 15 | , | Comma | Left-to-right | |



Many math functions in JAVA are also available in C. To use them however, you must include the `<math.h>` header file (and `<stdlib.h>` for the `rand()` function):

| Math in C | Math in JAVA | Trig. in C | Trig. in JAVA |
|-------------------|-------------------------|------------------|----------------------------|
| N/A | Math. <i>min</i> (a, b) | sin (r) | Math. <i>sin</i> (r) |
| N/A | Math. <i>max</i> (a, b) | asin (r) | Math. <i>asin</i> (r) |
| ceil (a) | Math. <i>ceil</i> (a) | cos (r) | Math. <i>cos</i> (r) |
| floor (a) | Math. <i>floor</i> (a) | acos (r) | Math. <i>acos</i> (r) |
| round (a) | Math. <i>round</i> (a) | tan (r) | Math. <i>tan</i> (r) |
| pow (a, b) | Math. <i>pow</i> (a, b) | atan (r) | Math. <i>atan</i> (r) |
| sqrt (a) | Math. <i>sqrt</i> (a) | atan2 (r) | Math. <i>atan2</i> (r) |
| fabs (a) | Math. <i>abs</i> (a) | N/A | Math. <i>toDegrees</i> (r) |
| rand () | Math. <i>random</i> () | N/A | Math. <i>toRadians</i> (d) |

Here is an example that makes use of some of the operators:

| Code from MathOps.c | Output |
|--|---|
| <pre> #include <stdio.h> int main() { int x, y, z; x = 4; y = x; z = y + 2 * x - 3; printf("%d %d %d\n", x, y, z); if (x == y) printf("equal\n"); else printf("not equal\n"); printf("%s\n", ((y == z) ? "equal" : "not equal")); x = y = 7; z -= x; y *= z; printf("%d %d %d\n", x, y, z); printf("prefix: %d\n", ++x); printf("postfix: %d\n", x++); printf("next: %d\n\n", x); printf("y: %d\n", y); printf("y + 1: %d\n", y + 1); printf("y: %d\n", y); printf("++y: %d\n", ++y); printf("y: %d\n\n", y); y += 1; printf("after y += 1: %d\n\n", y); y + 1; printf("after y + 1: %d\n\n", y); } </pre> | <pre> 4 4 9 equal not equal 7 14 2 prefix: 8 postfix: 8 next: 9 y: 14 y + 1: 15 y: 14 ++y: 15 y: 15 after y += 1: 16 after y + 1: 16 </pre> |

Here is another example that makes use of some math functions. To use the math functions, however, we must make sure to add `-lm` (which means “include the math library”) to the end of our `gcc` compiling line so that it is linked with the math library:

```
student@COMPBase:~$ gcc -o trig trig.c -lm
student@COMPBase:~$
```

| Code from <code>trig.c</code> | Output |
|---|--|
| <pre>#include <stdio.h> #include <math.h> #define PI 3.14159265 // or defined as M_PI in math.h int main() { double angle = 90; // in degrees printf("sin(90): %f\n", sin(PI*angle/180)); printf("cos(90): %f\n", cos(PI*angle/180)); printf("tan(90): %f\n", tan(PI*angle/180)); printf("ceil(235.435): %g\n", ceil(235.435)); printf("floor(235.435): %g\n", floor(235.435)); printf("pow(2, 8): %g\n", pow(2,8)); printf("fabs(-15): %g\n", fabs(-15)); }</pre> | <pre>sin(90): 1.000000 cos(90): 0.000000 tan(90): 557135183.943528 ceil(235.435): 236 floor(235.435): 235 pow(2, 8): 256 fabs(-15): 15</pre> |

Regarding the `rand()` function, it returns a pseudo-random integer (i.e., not truly random) in the range of 0 to `RAND_MAX` each time. `RAND_MAX` is defined in `<stdlib.h>` and has a value of **2,147,483,647** in the virtual box that we will be running in. So, to get the number into a value between 0 and 1, as JAVA gives us, we need to divide by `RAND_MAX`:

```
double value = rand() / (double) RAND_MAX;
```

Unfortunately, if you are not careful ... the sequence of random numbers generated will always be the same each time that you run your program.



The following code, for example, always prints out the same 10 randomly-generated numbers every single time that you run it !

| Code from <code>randTest1.c</code> | Output |
|--|---|
| <pre>#include <stdio.h> #include <stdlib.h> int main() { for(int i = 0; i<10; i++) printf("%g\n", rand() / (double) RAND_MAX); }</pre> | <pre>0.840188 0.394383 0.783099 0.79844 0.911647 0.197551 0.335223 0.76823 0.277775 0.55397</pre> |

Why? That does not seem *random* at all! Well, the random number generator is really just choosing a sequence of random numbers based on some starting point in the sequence. In order to get a variety of pseudo-random numbers, we need to set the starting point for the series of numbers generated. We do this by setting the **seed** of the generator.

The **srand(s)** function allows us to set the starting point in the sequence to be integer **s**.

| Code from randTest2.c | Output |
|--|--|
| <pre>#include <stdio.h> #include <stdlib.h> int main() { srand(1); for (int i = 0; i<5; i++) printf("%d\n", (int) (rand() / (double) RAND_MAX * 100)); srand(2); for (int i = 0; i<5; i++) printf("%d\n", (int) (rand() / (double) RAND_MAX * 100)); srand(683); for (int i = 0; i<5; i++) printf("%d\n", (int) (rand() / (double) RAND_MAX * 100)); }</pre> | <pre>84 39 78 79 91 70 80 8 12 34 75 10 18 10 91</pre> |

Of course, if you always use the same seed, you are stuck again with a fixed sequence.

Therefore, what is often done is to set the seed to be somewhat unique each time you run the program. One way to do this is to incorporate the time of day into the seed. The time of day can be obtained using the **time(NULL)** function defined in the **<time.h>** header. This function returns the number of seconds that have elapsed since the Epoch (00:00:00 UTC, January 1, 1970). The code below “always” produces different/unique numbers.

| Code from randTest3.c | Output |
|--|--|
| <pre>#include <stdio.h> #include <stdlib.h> #include <time.h> int main() { srand(time(NULL)); for (int i = 0; i<10; i++) printf("%d\n", (int) (rand() / (double) RAND_MAX * 100)); }</pre> | <pre>32 28 60 83 37 26 36 46 64 70</pre> |

There is an advantage of having a fixed sequence of random numbers when you run your program. Imagine, for example, that your program uses a lot of random numbers to make choices in your program (e.g., a game character makes random decisions to turn left or right or go straight on a gameboard). If your program crashed or caused some undesirable situation in your game at some point, you will need to debug your code.

If you set the sequence of random numbers to be fixed, then each time you run your code, the character ends up in the same location. This allows you to trace the steps carefully, make some changes and re-run the same set of steps.

Arrays:

In **C**, arrays work in a similar way as those in JAVA, but the syntax is not as flexible. If you want to make arrays fully populated with data, here is what you can do:

| Code from <code>arrays.c</code> | Output |
|---|---|
| <pre>#include <stdio.h> int main() { int ages[] = {34, 12, 45}; double weights[] = {4.5, 23.6, 84.1, 78.2, 61.5}; char vowels[] = {'a', 'e', 'i', 'o', 'u'}; printf("\nHere is the ages array:\n"); for (int i=0; i<3; i++) printf("%2d: %d\n",i, ages[i]); printf("\nHere is the weights array:\n"); for (int i=0; i<5; i++) printf("%2d: %g\n",i, weights[i]); printf("\nHere is the vowels array:\n"); for (int i=0; i<5; i++) printf("%2d: %c\n",i, vowels[i]); return(0); }</pre> | <pre>Here is the ages array: 0: 34 1: 12 2: 45 Here is the weights array: 0: 4.5 1: 23.6 2: 84.1 3: 78.2 4: 61.5 Here is the vowels array: 0: a 1: e 2: i 3: o 4: u</pre> |

Although the code looks similar to JAVA, there are some differences. We cannot, for example, move the square brackets over to the type side as follows:



```
int[]    ages    = {34, 12, 45};
double[] weights = {4.5, 23.6, 84.1, 78.2, 61.5};
char[]   vowels  = {'a', 'e', 'i', 'o', 'u'};
```

Also, we can specify the maximum capacity of the array even if we do not want to fill it up:

| Code from <code>arrays1.c</code> | Output |
|---|---|
| <pre>#include <stdio.h> int main() { int ages[8] = {34, 12, 45}; char vowels[8] = {'a', 'e', 'i', 'o', 'u'}; printf("\nHere is the ages array:\n"); for (int i=0; i<8; i++) printf("%2d: %d\n",i, ages[i]); }</pre> | <pre>Here is the ages array: 0: 34 1: 12 2: 45 3: 0 4: 0 5: 0 6: 0 7: 0</pre> |

| | |
|--|--|
| <pre>printf("\nHere is the vowels array:\n"); for (int i=0; i<8; i++) printf("%2d: %c\n",i, vowels[i]); return(0); }</pre> | <pre>Here is the vowels array: 0: a 1: e 2: i 3: o 4: u 5: 6: 7:</pre> |
|--|--|

There are some dangers to be aware of with respect to array boundaries. There is no checking for “Index Out of Bounds” errors as with JAVA. So, C will allow us to access beyond an array’s boundaries without warning us. This could be a problem:

| Code from arrays2.c | Output |
|---|---|
| <pre>#include <stdio.h> int main() { int ages[8] = {34, 12, 45}; char vowels[8] = {'a', 'e', 'i', 'o', 'u'}; printf("\nHere is the ages array:\n"); for (int i=0; i<15; i++) printf("%2d: %d\n",i, ages[i]); // Notice how data beyond boundary is invalid -----> printf("\nHere is the vowels array:\n"); for (int i=0; i<8; i++) printf("%2d: %c\n",i, vowels[i]); ages[8] = 78; // This goes beyond bounds!! // This will overwrite/corrupt // any variables declared after it printf("\nHere is the vowels array:\n"); // Notice how data in vowels array is corrupt -----> for (int i=0; i<8; i++) printf("%2d: %c\n",i, vowels[i]); return(0); }</pre>   | <pre>Here is the ages array: 0: 34 1: 12 2: 45 3: 0 4: 0 5: 0 6: 0 7: 0 8: 1869178209 9: 117 10: -397217792 11: -1216973860 12: -1076233904 13: 0 14: -1218652617 Here is the vowels array: 0: a 1: e 2: i 3: o 4: u 5: 6: 7: Here is the vowels array: 0: N 1: 2: 3: 4: u 5: 6: 7:</pre> |

We will be using arrays throughout the course. Be aware of these boundary issues, as it can cause problems in your code in places that you would never had expected it.

1.5 Getting User Input

Getting user input is also quite straight forward in C. Recall that in JAVA, we made use of the **Scanner** class to get user input:

```
import java.util.Scanner;

public class GreetingProgram {
    public static void main(String[] args) {
        System.out.println("What is your name ?");
        String name = new Scanner(System.in).nextLine();
        System.out.println("Hello, " + name);
    }
}
```

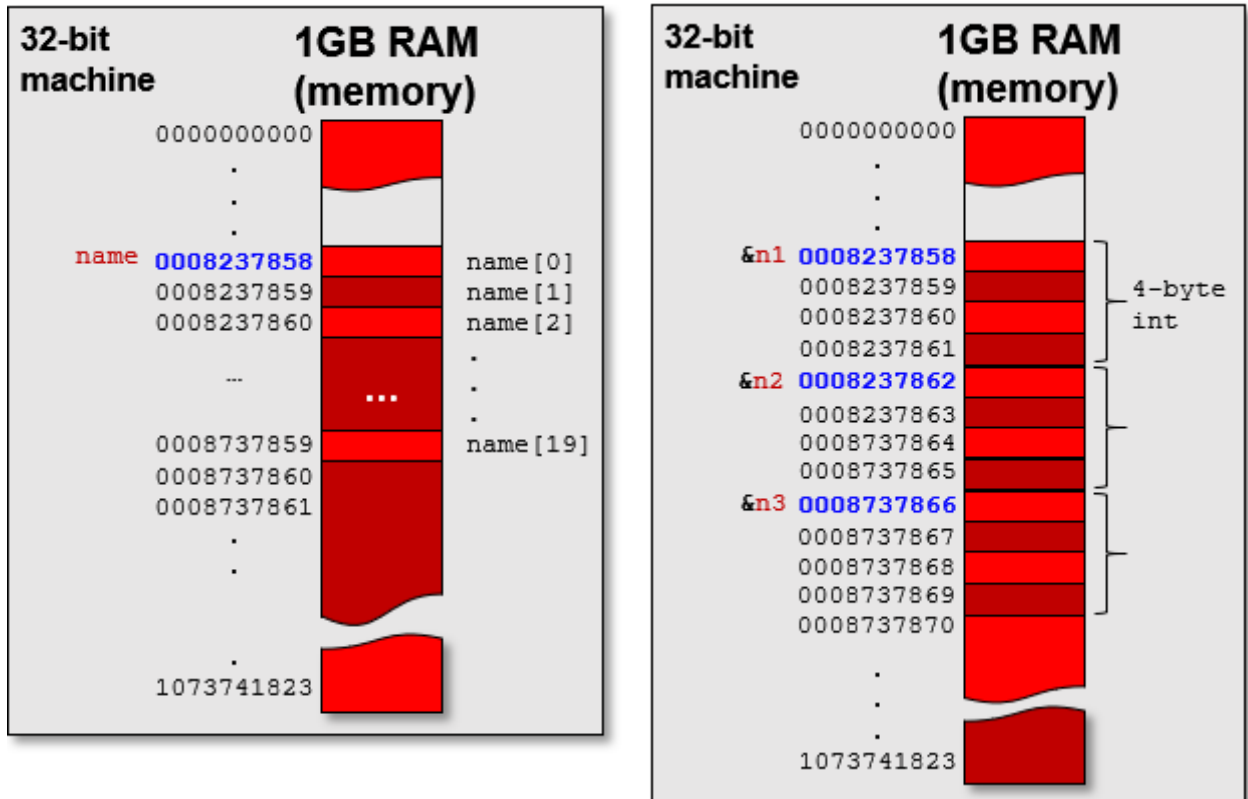
In C, we make use of the **scanf()** function ... which has a similar format to that of **printf**:

| Code from <code>userInput.c</code> | Output |
|---|---|
| <pre>#include <stdio.h> int main() { char name[20]; printf("What is your name ? \n"); scanf("%s", name); printf("Hello, %s\n", name); return 0; }</pre> | <pre>What is your name ? Mark Hello, Mark</pre> |

Notice that the **scanf()** takes a parameter string, like the **printf()**. As well, for each parameter it takes a variable. The variable must actually be a **pointer** (more on this later). That is ... we need to tell the compiler the memory location that we want the incoming information to be stored into. In the above example, the variable **name** is actually a pointer to a **char** array. If we want to simply store a value into an **int**, **float**, **double** or single **char**, then we need to use the **&** symbol in front of the variable to indicate that we want the **address** of the variable (i.e., its location in memory).

| Code from <code>numberInput.c</code> | Output |
|--|---|
| <pre>#include <stdio.h> int main() { int n1, n2, n3; printf("Enter three numbers:\n"); scanf("%d", &n1); scanf("%d", &n2); scanf("%d", &n3); printf("The average is %g\n", (n1+n2+n3)/3.0); return 0; }</pre> | <pre>Enter three numbers: 34 89 17 The average is 46.6667</pre> |

Here are some memory diagrams that show how the **char** array **name** is stored in the memory as well as the integers **n1**, **n2** and **n3**. Each **char** takes 1 byte of space. We do not need the **&** symbol when we use it because it is implied with arrays that we are talking about the address of the first item in the list. When using the numbers though, the **&** is required so that we give the **scanf()** function the memory location of where we want to store the incoming value. Each **int** takes 4 bytes of space (more on this later).



There are lots of parameters for the **scanf()** function, just as with **printf**. We will not discuss them all here. Instead, we will just mention a few more issues at this time, related to reading in strings of characters. Note that in the above code, we set the **name** array to be size **20**. Since we will be treating the char array as a string ... most of the string functions expect there to be a **null-terminating character** (i.e., a **0**) at the end of the string. Therefore, we can only use **19** of the chars for the name, leaving the last one as **0** to indicate the end of the string.

Thus, our **name** can be at most 19 characters long. That means if we enter anything more than this, we will have crossed our boundary line and will be overwriting the memory locations that come after it (i.e., 0008737866, 0008737867, 0008737868, etc.). This could be horrible as we will likely be overwriting important information stored in other variables. So, we need to prevent this from happening.



To prevent the `scanf()` from accepting more characters, we need to supply the *width* limit, as we did when using `printf()`. So, we would do this:

```
scanf("%19s", name);
```

As a result, our code will be safe (i.e., won't overflow) for names with more than 19 chars (e.g., **Hatmaguptafratarinagarosterlous** ... which is an actual first name):

| Code from <code>userInput2.c</code> | Output |
|---|---|
| <pre>#include <stdio.h> int main() { char name[20]; printf("What is your name ? \n"); scanf("%19s", name); printf("Hello, %s\n", name); return 0; }</pre> | <pre>What is your name ? Hatmaguptafratarinagarosterlous Hello, Hatmaguptafratarina</pre> |

The other issue when reading a series of items using `scanf()` is that sometimes the carriage return (i.e., `\n`) is not read in and remains in the buffer. This could cause a problem when reading in more than one item from the program. In our earlier code, where we read in three integers, there was no problem because reading in using the `%d` will ignore the `\n` character that is left over from the previous read. However, reading in a character using `%c` would be a problem because it will consider the `\n` from the previous number as being the character that we want to read in. Consider this code, for example:

```
int age;
char gender, hand;

printf("How old are you? \n");
scanf("%d", &age);

printf("What gender are you (M or F)?\n");
scanf("%c", &gender);

printf("What hand do you write with (L or R)?\n");
scanf("%c", &hand);

printf("You are a %d year old %c who writes with hand %c\n", age, gender, hand);
```

Notice the output that would result:

```
How old are you?
65
What gender are you (M or F)?
What hand do you write with (L or R)?
L
You are a 65 year old
    who writes with hand L
```

You can see that the `\n` character is read in as the **gender** ... which was left over from the **age** input. To solve this issue, whenever you read from the keyboard using `scanf()`, you can call `getchar()` to read the carriage return character:

```
int age;
char gender, hand;

printf("How old are you? \n");
scanf("%d", &age);
getchar();
printf("What gender are you (M or F)?\n");
scanf("%c", &gender);
getchar();
printf("What hand do you write with (L or R)?\n");
scanf("%c", &hand);
getchar();
printf("You are a %d year old %c who writes with hand %c\n", age, gender, hand);
```

Then the code will work fine:

```
How old are you?
65
What gender are you (M or F)?
M
What hand do you write with (L or R)?
L
You are a 65 year old M who writes with hand L
```

There can be many such issues that pop up. It is important to be aware of them. For example, if we read in a name using this:

```
scanf("%10s", firstName);
scanf("%15s", lastName);
```

Then we need to realize that if we enter more than 10 characters in the first name, the unused characters will be still in the standard keyboard input buffer and will therefore be read in as the first few letters of the last name! Sometimes, we are simply left with no solution except to empty out the buffer each time manually like this:

```
while(getchar() != '\n'); // The ; indicates no body for the loop
```

which can be used as follows:

```
printf("What is your first name? \n");
scanf("%10s", firstName);
while(getchar() != '\n');
printf("What is your last name?\n");
scanf("%15s", lastName);
while(getchar() != '\n');
printf("How old are you?\n");
scanf("%d", &age);
while(getchar() != '\n');
printf("Hi %s %s ... you are %d years old.\n", firstName, lastName, age);
```

Example:

Let us write a program (that you may have already done in JAVA) that displays the following menu:

```
Luigi's Pizza
-----
                S (SML)  M (MED)  L (LRG)
1. Cheese           5.00    7.50    10.00
2. Pepperoni       5.75    8.63    11.50
3. Combination     6.50    9.75    13.00
4. Vegetarian      7.25   10.88    14.50
5. Meat Lovers     8.00   12.00    16.00
```

The program should then prompt the user for the type of pizza he/she wants to order (i.e., 1 to 5) and then the size of pizza 'S', 'M' or 'L'. Then the program should display the cost of the pizza with 13% tax added.

Once we have the **kind** and **size** from the user, we will need to compute the total cost. Notice that the cost of a small pizza increases by \$0.75 as the kind of pizza increases. Also, you may notice that the cost of a medium is 1.5 x the cost of a small and the cost of a large is 2 x a small. So, we can compute the cost of any pizza based on its kind and size by using a single mathematical formula.

A small pizza would cost: **smallCost = \$4.25 + (kind x \$0.75)**
 A medium pizza would cost: **mediumCost = smallCost * 1.5**
 A large pizza would cost: **largeCost = smallCost * 2**

Here is the code:

```
#include <stdio.h>

int main() {
    int    kind;
    float  cost;
    char   size;

    printf("Luigi's Pizza\n");
    printf("-----\n");
    printf("                S (SML)  M (MED)  L (LRG)\n");
    printf("1. Cheese           5.00    7.50    10.00\n");
    printf("2. Pepperoni       5.75    8.63    11.50\n");
    printf("3. Combination     6.50    9.75    13.00\n");
    printf("4. Vegetarian      7.25   10.88    14.50\n");
    printf("5. Meat Lovers     8.00   12.00    16.00\n\n");

    printf("What kind of pizza do you want (1-5)?\n");
    scanf("%d", &kind);
    while(getchar() != '\n');

    printf("What size of pizza do you want (S, M, L)?\n");
    scanf("%c", &size);
    while(getchar() != '\n');
```

```
cost = 4.25 + (kind * 0.75);
if (size == 'M')
    cost *= 1.5;
else if (size == 'L')
    cost *= 2;

printf("\nThe cost of the pizza is: $%0.2f\n", cost);
printf("The price with tax is: $%0.2f\n", cost * 1.13);

return 0;
}
```

Here is the expected output when ordering a Medium Vegetarian pizza:

Luigi's Pizza

```
-----
                S (SML)  M (MED)  L (LRG)
1. Cheese           5.00    7.50   10.00
2. Pepperoni        5.75    8.63   11.50
3. Combination      6.50    9.75   13.00
4. Vegetarian       7.25   10.88   14.50
5. Meat Lovers      8.00   12.00   16.00
```

What kind of pizza do you want (1-5) ?

4

What size of pizza do you want (S, M, L) ?

M

The cost of the pizza is: \$10.88

The price with tax is: \$12.29

1.6 Functions & Procedures in C

When programming, it is often advantageous to write *functions* in order to modularize our code and to break complicated code into easily understood pieces that are also maintainable.

In procedural languages, such as C, functions & procedures are the basic building blocks of our programs. In OO languages, *classes* are the basic building blocks in that the object definitions are what structure the code, while functions are “hidden” inside the classes.

When designing functions, if you want to do it “correctly”, you should keep these points in mind. A function should ...

- take data in, do something and then return a result
 - results can be returned as return value or through parameters
- be single-purpose
 - have a single goal, do one thing only
- encapsulate (i.e., hide) functionality
 - user knows what function does, not how it does it
- be reusable
 - in the same program or other programs

In C, it is important to understand how data is shared between functions. In some ways, it can be different in C than in JAVA. A parameter to a function will have one of three purposes:

- **input** – data passed in and required for the function to complete its task
- **output** – data that is set or “filled-in” by the function ... the results of the function’s work
- **input/output** – data both required by the function and resulting from it

At ALL times, it is vital that we understand the purpose of each parameter before we write the code. Interestingly, the return value of a function is almost never used to return data! It is often used only to return the function’s status ... that is ... to indicate success or failure.

Consider the following coding example that prompts the user for some integers until either 10 are entered or until a -1 is entered. It then displays the entered values and computes the maximum.

| Code from <code>arrays3.c</code> | Output |
|--|--------|
| <pre> #include <stdio.h> #define MAX_SIZE 10 // maximum numbers allowed int main() { int array[MAX_SIZE]; // Get the numbers from the user int totalNums = 0; int num; </pre> | |

| | |
|---|---|
| <pre> do { printf("Enter a number (-1 to quit): "); scanf("%d", &num); if (num != -1) array[totalNums++] = num; } while ((totalNums < MAX_SIZE) && (num != -1)); // Compute the maximum of the array int max = 0; for (int i=0; i<totalNums; i++) { if (array[i] > max) max = array[i]; } // Print the array printf("\nHere is the array:\n"); for (int i=0; i<totalNums; i++) printf("%2d: %d\n",i, array[i]); // Print the maximum printf("\nMax is %d\n", max); return(0); } </pre> | <pre> Enter a number (-1 to quit): 12 Enter a number (-1 to quit): 43 Enter a number (-1 to quit): 65 Enter a number (-1 to quit): 23 Enter a number (-1 to quit): 8 Enter a number (-1 to quit): -1 Here is the array: 0: 12 1: 43 2: 65 3: 23 4: 8 Max is 65 </pre> |
|---|---|

We are now going to see how to make three functions from this code. One to get the input, one to display the array and one to compute the maximum. To begin, let us make a procedure that simply takes in the array and displays it. We will call the function **displayArray()**. It will take two parameters ... the array itself and the number of elements in the array. There is no return value for the procedure, so we use **void**. Here is the template:

```

void displayArray(int theArray[], int theArraySize) {
    // ...
}

```

Notice how the array is passed in as a parameter by using the **[]** characters. We actually aren't passing the **whole** array into the procedure ... we just passing in the address of (i.e., a pointer to) where the array is stored in memory. The ***** character means that we are passing in a pointer (i.e., reference) to a memory location. This is the same as in JAVA, in that the object is passed in as a reference to its memory location.

So in our example, to call the procedure in C, we do this:

```
displayArray(array, totalNums);
```

Here, the **array** parameter doesn't have brackets. Whenever we pass just an array name, it represents the memory location (i.e., reference) of the beginning of the array in memory. The **totalNums** parameter represents the value of the number stored in the **totalNums** variable.

When passing in parameters to functions and procedures in C, we can do one of two things:

Pass-by-value

- value is copied into function
- function works on the local copy
- copy is lost when function returns
- value in calling function cannot be changed



Pass-by-reference

- address of value is passed into function
- value in calling function can be changed

The code for the procedure is straight forward:

```
void displayArray(int theArray[], int theArraySize) {
    printf("\nHere is the array:\n");
    for (int i=0; i<theArraySize; i++)
        printf("%2d: %d\n",i, theArray[i]);
}
```

The only change that we made is to use the new incoming parameter names instead of the names of the variables that were passed in.

We can do something similar for computing the maximum value in the array:

```
int findMaximum(int theArray[], int theArraySize) {
    int m = 0;
    for (int i=0; i<theArraySize; i++) {
        if (theArray[i] > m)
            m = theArray[i];
    }
    return m;
}
```

Notice now that this is a *function* with an **int** being returned. The code is straight forward from our existing code that we had written previously. Calling this from the **main** program is also easy:

```
int max = findMaximum(array, totalNums);
```

The computed maximum will be stored in the **max** variable when the function returns.

Finally, we can write the code that gets the array data. It will need to return the number of values entered so that we can use this in our main program to pass it to the other two functions:

```
int getArrayData(int theArray[]) {
    int count = 0;
    int num;
```

```

do {
    printf("Enter a number (-1 to quit): ");
    scanf("%d", &num);

    if (num != -1)
        theArray[count++] = num;
} while ((count < MAX_SIZE) && (num != -1));

return count;
}

```

Again, calling this is easy. We'll make sure to store the return value:

```
int totalNums = getArrayData(array);
```

Finally, we can put it all together.

Code from arrays4.c

```

#include <stdio.h>

#define MAX_SIZE 10 // maximum numbers allowed

// Function definitions
void displayArray(int[], int);
int findMaximum(int[], int);
int getArrayData(int[]);

int main() {
    int array[MAX_SIZE];

    int totalNums = getArrayData(array); // Get the numbers from the user
    displayArray(array, totalNums); // Print the array

    int max = findMaximum(array, totalNums); // Compute the maximum of the array
    printf("\nMax is %d\n", max); // Print the maximum

    return(0);
}

/*****
/* Display the values of the given array */
*****/
void displayArray(int theArray[], int theArraySize) {
    printf("\nHere is the array:\n");
    for (int i=0; i<theArraySize; i++)
        printf("%2d: %d\n", i, theArray[i]);
}

/*****
/* Find and return the maximum value of the given array */
*****/
int findMaximum(int theArray[], int theArraySize) {
    int m = 0;
    for (int i=0; i<theArraySize; i++) {
        if (theArray[i] > m)
            m = theArray[i];
    }
    return m;
}

```

```

/*****
/* Prompt the user for values and place them into the given */
/* array until the array is full or -1 is entered          */
/*****
int getArrayData(int theArray[]) {
    int count = 0;
    int num;
    do {
        printf("Enter a number (-1 to quit): ");
        scanf("%d", &num);

        if (num != -1)
            theArray[count++] = num;
    } while ((count < MAX_SIZE) && (num != -1));
    return count;
}

```

There is one addition that you may have noticed. Each of the functions must be declared before it is used. To declare the function, we simply supply its signature:

```

void displayArray(int[], int);
int findMaximum(int[], int);
int getArrayData(int[]);

```

If you do not include these declarations, you will get compile errors such as these:

```

arrays4.c: In function 'main':
arrays4.c:23:19: warning: implicit declaration of function 'getArrayData' [-Wimplicit-function-declaration]
    int totalNums = getArrayData(array);    // Get the numbers from the user
                    ^
arrays4.c:24:3: warning: implicit declaration of function 'displayArray' [-Wimplicit-function-declaration]
    displayArray(array, totalNums);        // Print the array
    ^
arrays4.c:26:13: warning: implicit declaration of function 'findMaximum' [-Wimplicit-function-declaration]
    int max = findMaximum(array, totalNums); // Compute the maximum of the array
              ^
arrays4.c: At top level:
arrays4.c:36:6: warning: conflicting types for 'displayArray'
    void displayArray(int *theArray, int theArraySize) {
        ^
arrays4.c:24:3: note: previous implicit declaration of 'displayArray' was here
    displayArray(array, totalNums);        // Print the array
    ^

```



We will talk more in depth later about addresses to arrays and we will do many examples in which we create functions and pass in parameters by value as well as by reference.

1.7 Coding Conventions/Style

This section just mentions a few things that you should keep in mind when coding in the C language.

Variable/Function Names:

Fixed values should be all uppercase letters with compound names separated by underscores:

```
PI
DAYS_OF_THE_WEEK
INTEREST_RATE
```



Variable names should ALWAYS begin with lowercase letters but when multiple words are used in a variable name, each word should be capitalized, except the first one. Alternatively, underscore characters can be used to separate the variable names.

Pick descriptive names, not lame ones that make no sense. Here are some good ones:

```
count                insurance_rate
average              time_of_day
insuranceRate        a_string
timeOfDay            latest_account_number
aString              weight_in_kilograms
latestAccountNumber
weightInKilograms
```

Loop counters often use single letters (but you may call them whatever you'd like):

| | |
|---|---|
| <pre>for (int i=0; i<10; i++) { // ... }</pre> | <pre>for (int r=0; r<10; r++) { for (int c=0; c<10; c++) { // ... } }</pre> |
|---|---|

Data types (more on this later) should have names that follow the style of variable names, except that they should begin with an uppercase letter ... just as class names in JAVA began with an uppercase letter.

```
PersonType
StudentType
BankAccountType
```

Finally, for functions and procedures, following the naming convention for variable names.

```
getBalance
calculateMonthlyPayment
```

If it is a hidden function though (i.e., meant to be private), the function name should begin with an underscore.

```
_performValidation
```

Indentation

As a computer programmer, it is important that you properly indent your code and use decent spacing. This will ensure that your code is readable. It helps the reader of the code to quickly understand which statements belong together. It also allows you to easily determine which brackets match. After all ... can you imagine trying to read poorly indented code like this?

```
#include <stdio.h>

int main() {
int    isPrime;

printf("Prime numbers from 1 to 100 are:\n");
for (int i=1; i<=100000; i++) {
isPrime = 1; // Assume that i is prime
for (int j=2; j<i; j++) {

if (i%j == 0) {
isPrime = 0; // if found, not prime
break;
}
}

if (isPrime)
printf("%d\n", i);
}
}
```



A **block** of code is a sequence of code between one pair of matching braces. A **nested block** is a block contained within another block. The general rules are:

- ALL statements within a block must be indented identically.
- ALL blocks nested at the same level must have their statements indented identically

```
#include <stdio.h>

int main() {
int    isPrime;

printf("Prime numbers from 1 to 100 are:\n");
for (int i=1; i<=100000; i++) {
isPrime = 1; // Assume that i is prime
for (int j=2; j<i; j++) {
if (i%j == 0) {
isPrime = 0; // if found, not prime
break;
}
}
if (isPrime)
printf("%d\n", i);
}
}
```



Be consistent with your bracketing style ... do not mix styles. There are 2 reasonable styles:

| | |
|--|--|
| <pre>int main() { int isPrime; printf("Prime numbers from 1 to 100 are:\n"); for (int i=1; i<=100000; i++) { isPrime = 1; // Assume that i is prime for (int j=2; j<i; j++) { if (i%j == 0) { isPrime = 0; // if found, not prime break; } } if (isPrime) printf("%d\n", i); } }</pre> | <pre>int main() { int isPrime; printf("Prime numbers from 1 to 100 are:\n"); for (int i=1; i<=100000; i++) { isPrime = 1; // Assume that i is prime for (int j=2; j<i; j++) { if (i%j == 0) { isPrime = 0; // if found, not prime break; } } if (isPrime) printf("%d\n", i); } }</pre> |
|--|--|

Commenting

Students often ask: “How much should I comment my code?”. This is VERY subjective indeed. Some like few comments, some like many comments. Some overly comment so that every line of code is commented ... this is too excessive.

As a rule of thumb, comments are usually needed to document:

- **The program** - comments in the file that contains the **main()** function.
 - purpose of the program
 - how to use the program (e.g., explaining command line arguments)
 - the author
 - revisions over time

```

/*****
/*
/* GraphGen.C
/*
/* This program generates a graph from a triangle.
/* The user must supply 3 coordinates for the triangle vertices.
/*
/* Usage:  graphgen <x1><y1><x2><y2><x3><y3><numSteiners> <outputFilename>
/*
/*****

#include <sys/types.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    FILE    *outFile;
    int     numVertices, numOuterVertices, numEdges, numOuterEdges;
    int     i, j, k;

    ...

```

- **Each function** - appearing before each function
 - describes purpose of function
 - each parameter, indicating whether input, output, or input/output
 - possible return values
 - side-effects

```

/* This function builds projections representing propagation */
/* from a vertex (S) of a TIN. It does this by adding a new */
/* node for each incident face of the vertex. Since TINs */
/* are non-convex polyhedra, we must also add projections */
/* through vertices. */
/*
/*      v1
/*      o
/*      /\
/*     /  \
/*    /    \
/*   /      \
/*  /        \
/* /          \
/* o/         \o
/* v2         e3 v3
*/
*/
/*      The projections from S are
/*      actually the three whole edges.
/*      Also, we allow propagating
/*      through the vertices of the
/*      face.
*/
/*
/* These projections are built and then stored in the tree
/* called the "Sequence Tree" (which can have nodes corres-
/* ponding to edge or vertex projections).
*/
/*
/* vertexNode = vertex S from which to project.
/* seqTree = sequence tree to be returned.
*/

static void CHSP_VertexPropagate(TreeNode *vertexNode, SeqTree *seqTree) {
    Tin_vertex      *aVertex;
    Tin_vertex2D    *nextVertex, *nextNextVertex;
    Tin_edge        *anEdge;

```

- **Variables and Structures** – before variable declaration or alongside

- main purpose of variable
- ranges of variables, or anything that clarifies a value

```

/* This structure represents the data stored in a vertex node in the */
/* sequence tree. The tree information is not stored here. */
struct VertexNodeInfoStruct {
    Tin_vertex2D    image;           /* The unfolded image vertex */
    Tin_vertex2D    vertex;         /* The vertex of projection */
    Tin_face2D      *pFace;         /* Face projected through */
    float          dist;           /* Shortest distance from src */
};
typedef struct VertexNodeInfoStruct    VertexNodeInfo;

/* This structure represents the data stored in an edge node in the */
/* sequence tree. The tree information is not stored here. */
struct EdgeNodeInfoStruct {
    Tin_edge2D      pEdge;          /* The edge of projection */
    Tin_face2D      pFace;         /* Face projected through */
    Tin_vertex2D    *image;        /* The unfolded image vertex */
    Tin_vertex2D    vLeft;         /* A projection endpoint */
    Tin_vertex2D    vRight;        /* A projection endpoint */
    float          dist;           /* Shortest dist from src */
};

/* For 64 color rasters, I provide some color masks here */
#define DM_BLUE_MASK    0x30    /* 00110000 */
#define DM_GREEN_MASK   0x0C    /* 00001100 */
#define DM_RED_MASK     0x03    /* 00000011 */
#define DM_WHITE_MASK   0x3F    /* 00111111 */
#define DM_ALL_MASK     0xFF    /* 11111111 */

```


- **Throughout the code** – before each chunk of complex/critical code
 - explain portion of algorithm that is accomplished by code that follows
 - keep it brief

```

SPPath* CHSP_PathToDestination(SeqTree *aSeqTree, int destID, Tin *aTin) {
    SPPath          *path;
    TreeNode        *node;
    Tin_vertex2D    *last2DVertex, *iPt;
    Tin_vertex      *Ie, *last3DVertex;
    EdgeNodeInfo    *nodeInfo;
    float           scale;
    VertexList      *newVertex;
    FILE            *chPathFile;

    /* Get the destination node in the sequence tree */
    node = aSeqTree->pathPtrs[destID];
    if (node == NULL) {
        printf("CHError: Destination is unreachable\n");
        return(NULL);
    }

    /* Open a file to store the resulting path */
    if (!(chPathFile = fopen("chenpath.dat", "a"))) {
        printf("Error: Cannot open chenhan.path output file\n");
        return;
    }

    /* Create a new path */
    path = (SPPath *)malloc(sizeof(SPPath));
    path->size = 0;
    path->length = aSeqTree->distances[destID];
    path->firstVertex = (VertexList *)malloc(sizeof(VertexList));
    path->lastVertex = path->firstVertex;
    path->firstVertex->vertex = ((VertexNodeInfo *) (node->data))->vertex.ref;
    path->firstVertex->next = path->firstVertex->prev = NULL;
    path->totalEdgeLength = 0.0;
    path->totalEdgeCount = 0;

    /* Store the first vertex in the path file */
    fprintf(chPathFile, "%d %d\n", path->firstVertex->vertex->id, path->firstVertex->vertex->id);

    /* Now go through and get the links of the path */
    last2DVertex = &(((VertexNodeInfo *) (node->data))->vertex);
    last3DVertex = last2DVertex->ref;
    while(node->parent != NULL) {
        path->size += 1;
        newVertex = (VertexList *)malloc(sizeof(VertexList));
        path->totalEdgeCount++;
        if (node->parent->type == VERTEX_NODE) {
            /* The parent projection was from a vertex */
            newVertex->vertex = ((VertexNodeInfo *) (node->parent->data))->vertex.ref;
            path->totalEdgeLength += TIN_MaxEdgeLengthFromVertex(aTin, newVertex->vertex);

            last2DVertex = &(((VertexNodeInfo *) (node->parent->data))->vertex);
            last3DVertex = newVertex->vertex;

            /* Store the intermediate vertex id in the path file */
            fprintf(chPathFile, "%d %d\n", newVertex->vertex->id, newVertex->vertex->id);
        }
        else {
            /* The parent projection was from an edge, */
            /* we need to find an intersection point. */
            nodeInfo = (EdgeNodeInfo *) (node->parent->data);
            scale = TIN_Edge2D_ScaleFromIntersect(&(nodeInfo->pEdge), last2DVertex,
                                                  nodeInfo->image);

            path->totalEdgeLength += TIN_EdgeLength(nodeInfo->pEdge.ref);
        }
    }
}

```

```
    /* Store the intermediate edge id in the path file */
    fprintf(chPathFile, "%d %d\n", nodeInfo->pEdge.ref->start->id,
           nodeInfo->pEdge.ref->end->id);

    Ie = TIN_Edge_PointAtScale(nodeInfo->pEdge.start.ref, nodeInfo->pEdge.end.ref, scale);
    Ie->id = -1;
    newVertex->vertex = Ie;
    last3DVertex = Ie;
}
/* Add the new vertex to the path */
path->lastVertex->next = newVertex;
newVertex->prev = path->lastVertex;
newVertex->next = NULL;
path->lastVertex = newVertex;

/* Get the next projection intersection */
node = node->parent;
}
fprintf(chPathFile, "%d %d\n", INFINITY, INFINITY);
fclose(chPathFile);

return (path);
}
```