CS102: Introduction

A computer is a system that consists of hardware (physical equipment) that executes software (programs or instructions that control the hardware).

Hardware can be divided into 5 main components:

1) **Input Devices** – Allow programs and data to be entered into the computer. Examples: keyboard, mouse.
2) **Central Processing Unit** (**CPU**) – Responsible for executing instructions.
3) **Primary Storage or Main Memory** – Hardware that stores programs and data temporarily while processing. If you turn off your computer, whatever is currently in main memory is lost.
4) **Auxiliary Storage** – Used for permanent storage of programs and data. Is not affected when you turn off your computer. Examples: hard drive, disk.
5) **Output Devices** – Where the output of a process is shown. Examples: monitor, printer.

Software can be broken down into two main categories:

1) **System Software** – Provides a user interface and tools that allow the user to access and use efficiently different components of the computer. Examples: operating system, disk formatting programs, compilers.
2) **Application Software** - Helps users to solve problems and accomplish tasks. Examples: word processors, spreadsheets.

In order to write a program, you must use a computer programming language.
The evolution of computer languages proceded roughly in the following stages:
1940's and earlier – Machine Languages
1940's and 1950's – Symbolic Languages (assembly languages)
Late 1950's through the present – High-Level Languages

There are many different categories of **high level languages**. For example, languages can be interpreted or compiled; they can be functional, procedural, or object-oriented (or allow combinations of these styles); some languages are considered scripting languages; etc. (There are many other categories.) Ultimately, though, there must be a conversion to a **machine language** (**machine code**) in order for a program to be executed.

Machine code is what computers "understand". Every program that you write will be translated to machine code before it is executed by a computer. (For interpreted languages, you can think of instructions being translated one at a time.) A machine language program can be thought of as a stream of 0's and 1's. The reason for this is that the internal circuits of computers are made of switches, transistors, and other electronic devices that can be in one of two states: off or on. The off state can be represented by 0 and the on state can be represented by 1. Each 0 or 1 digit is referred to as a **bit**. A bit is a single 0 or 1 in binary code. A **byte** is a sequence of 8 bits. Each byte could also be represented as a decimal number from 0 to 255.

Not all computers share the same machine language. Each type of computer has its own. So each time a new chip comes out, it may have a new machine language. One of the problems with programming in machine language is that you would have to learn a new version of it for each type of chip. Some machine languages have constant length instructions but others have variable length instructions depending on the type of instruction. Some instructions also include data. For example, an instruction telling the computer to "add" two numbers together must also specify the numbers.

Before a computer can execute binary code, the code must be stored in main memory. The starting location of the binary code is stored in a computer register, which is similar to main memory but can be accessed faster. A register is a location in which the computer can store information, and some registers have special purposes. One register is just used for storing the current location within the code of an executing machine language program. After the first instruction is located, the special register gets updated so that it points to the second instruction. Then that gets executed. Etc. Normally, instructions get executed sequentially. Sometimes, special instructions might change the special register used to keep track of the current location with the code of the executing program. Such instructions cause the flow of a program to jump from one location to another.

Clearly, another problem with programming in machine language is that it is totally confusing and unreadable. Nobody wants to memorize all these sequences of 0's and 1's. Nevertheless, the first computers required programming to be done in this manner. They were machines the size of rooms, millions of times slower than today's desktops, and they didn't use keyboards but switches or punch cards.

In the 1950's, the first **symbolic languages** (a.k.a. **assembly languages**) were created. The idea was to have each type of instruction represented by some intuitive code word, and to allow data to be represented as decimal numbers. Here is a short snippet from a symbolic language program:

```
pushal 3(r2)
calls #2,SCANF
mull3 -8(fp),-12(fp)
pushal 6(r2)
```

An **assembler** is a special program used to translate symbolic code into machine code. In actuality, the program containing the segment above would be translated into machine code that would then be executed. But it might be simpler to think of it as the code that is actually getting executed. Think of the computer walking along these instructions executing each one.

Assembly languages are more intuitive and easier to program than machine languages. However, there is still approximately a one to one correspondence between assembly language instructions and machine language instructions, and there has to be a different assembly language for each type of computer. To avoid these problems, high-level languages were developed.

Three benefits of high-level languages:
1) More powerful instructions; each may translate to several machine language instructions.
2) Portable to many different computer platforms.
3) Simpler to understand and use.

One of the first high-level languages was FORTRAN, created in 1957. Soon after FORTRAN was COBOL.

Today, one commonly used high-level languages is C.

Here is an abbreviated, quick history of C:
1972 – Dennis Ritchie created the first version of C
1978 – The first edition of "The C Programming Language" by Kernighan and Ritchie was published.
1983-1988 – A committee of the American National Standards Institute (ANSI) established a modern, comprehensive definition of C.
1988 – The second edition of "The C Programming Language" by Kernighan and Ritchie was published.

Other versions of C have since been adopted (the standard was modified in 1999 and 2011), but C has been fairly stable since the second edition. Also note that C++, still one of the most commonly used programming languages in the world, builds upon C (adding support for object-oriented programming as well as various other additions). C itself is still a very important language as well, and it seems to be experiencing a resurgence, as C-like languages are being used for various embedded platforms.

As with any high level language, a C program must be converted to machine language before it can be executed. The act of translating your C source code to machine language is known as compiling. A **compiler** translates your C code, stored in one or more source files, into the appropriate machine code.

A C program consists of variables and functions. A function consists of a sequence of statements. Some statements are single instructions that are part of the C language itself. Other statements are calls to functions, either functions that you write yourself or functions that are provided with the C language. ANSI C (The version of C that was approved by the American National Standards Institute) includes several libraries of functions that are provided to programmers in order to provide important functionality and simplify programming. When you want to call such provided functions, you must tell the compiler which libraries you are going to use, and some program must attach these libraries to your executable. A linker is a program which combines libraries and the machine code generated from a programmer's code into a single executable. Compilers and linkers can be separate programs, but in most cases, they will be coupled together into a single program that performs both functions.

It has become quite standard, for some reason, that the first program that is often taught in any high level programming language is a program which prints the words "Hello World!" on the screen on a line by itself. This is what the program might look like in C:

```
#include <stdio.h>

int main(void)
{
     printf("Hello World!\n");
     return 0;
}
```

Technically speaking, the word "int", the word "void", and the instruction "return 0;" are optional in this program. The program would still work as follows:

```
#include <stdio.h>

main()
{
     printf("Hello World!\n");
}
```

You're not yet in a position to understand why the optional segments are included. For now, it is a good idea of getting into the habit of including the extra "int", "void", and "return 0;" in all your main functions. The "return 0;" statement is just returning control from the program to the operating system with the message that no error has occurred.

Every C program includes a function named "main". This is where the execution of a program begins. Every **function body** (including "main") in every program begins with a left curly brace and ends with a right curly brace, and in between these curly braces are zero or more **statements**. The statements of "main" are executed one at a time until the program is finished. Some statements are actually calls to other functions. These functions may be written by the programmer or provided in a library. In this program, "printf" is a function provided in a library. The first line of the program, "#include <stdio.h>", indicates, in a sense, that the program will use one or more functions in the standard input/output library, and "printf" is one of these functions. When a function is called, execution of the program jumps to that function. When the called function is finished, execution continues in the calling function at the point directly after the call to the called function. It is too early to really understand how "printf" is working, but for now, just consider it a way to print strings to the screen. (A string is a sequence of characters.) In actuality, the characters are printed one at a time.

An important thing to remember: Statements in C are separated by semicolons. Notice, for example, the semicolon at the end of the "printf" line. It is necessary. If it isn’t there, the program won’t compile correctly. Forgetting semicolons will be one of the most common errors that you make. Fortunately, compilers are good at detecting this sort of error and warning you about it, so such an error is usually easy to catch and fix.

Also note the "\n" at the end of the string "Hello World". This is interpreted as a single special character, the end-of-line character. It causes the print routine to output to the monitor (or to whatever the standard output currently refers) an end-of-line character. It's like pressing Enter while using a word processor. The next piece of output will start on the next line.

C is pretty flexible when it comes to whitespace. Whitespace includes spaces, tabs, newline characters that you can not see. C ignores most whitespace characters that are not part of strings; this leaves a programmer some choice of style. When it comes to decisions such as how to indent code and where to place braces, you should be consistent so that your code is readable to others.

Before you can run a C program you must compile the program, and before you can compile the program you must store the source code for that program in a source file. You can create such a source file using a **text editor**. Some popular text editors that might be available when you are using a Linux system include emacs, vi, and pico (the last of which is less powerful but simple). If you are using a Windows system, one potential text editor is Notepad++. An alternative to explicitly specifying a text editor is to use an integrated development environment (IDE) with a graphical user interface that includes a built-in text editor.

Typically, the name of a file containing source code for a C program will end with the extension ".c", and you should be in the habit of always naming your source files this way if programming in standard C. For example, a sensible name for the program above would be "hello.c". After typing in the code of your program, you must use the editor (or the IDE) to "save" the file. Once you have saved the file, you are ready to compile the code. (Note that more advanced programs often use multiple files for a single program, but this will not be discussed here.)

Let's say you have created the file "hello.c" and are ready to compile it. If you are compiling from the command line, you will likely be using the GNU Compiler Collection (GCC). To use GCC, assuming you are in the same directory as the program "hello.c", you would type:

```
gcc hello.c
```

Assuming that there are no errors in your code, the compiler will create an executable that you can run. If you do not provide a name explicitly, the name of the executable defaults to "a.out" on a Linux system (or "a.exe" on a Windows system). So, if you list the contents of the directory after running one of these commands, the file "a.out" will be there. You can execute it by typing "./a.out" at the command prompt. (The "./" at the start of the command indicates that you are running "a.out" from the current directory.) Assuming you haven't made any mistakes, the words "Hello World!" will be displayed. The new command prompt will be on the line below.

What happens if you forgot the "\n" at the end of the string? You will still see the words "Hello World!" on the screen, but the new command prompt will be directly after the exclamation mark, and it won't look good. (In some environments, the first version, with the newline character, might automatically include a blank line before the new command line, and the second version would then leave it out but still display the command prompt an a new line.)

You don't have to use the default name of "a.out" for your executable; you can specify the name of the executable yourself. For instance, let's say you want the name of the executable to be "hello" with no extension. You would compile as follows:

```
gcc –o hello hello.c
```

And then you can run the executable by typing "./hello" at the command prompt.

Let's say you want "Hello World!" to be printed to the screen three times instead of one, with each instance on a separate line. With C, there are often multiple ways to solve the same problem. Here are two possible solutions for this problem.

Solution 1:

```
#include <stdio.h>

int main(void)
{
     printf("Hello World!\n");
     printf("Hello World!\n");
     printf("Hello World!\n");
     return 0;
}
```

Solution 2:

```
#include <stdio.h>

int main(void)
{
     printf("Hello World!\nHello World!\nHello World!\n");
     return 0;
}
```

With the first solution, the computer will step through the instructions one at a time, and each call to "printf" will cause the string "Hello World!" to be printed out on its own line. With the second solution, there is just one call to "printf", and the string includes the text "Hello World!" three times and also three end-of-line characters. I would not say that either of these solutions is clearly better than the other. The second is slightly more efficient (since there is only one function call), but the first is more readable.

Here is the general format of a C program:

```
Preprocessor Directives

Global Declarations

int main (void)
{
     Local Declarations

     Statements
}

Optionally other functions with similar structure to "main"
```

Now take another look at the "Hello World!" program:

```
#include <stdio.h>

int main(void)
{
     printf("Hello World!\n");
     return 0;
}
```

Our "Hello World!" program has just one preprocessor directive, the "#include <stdio.h>" line. This line is telling the compiler that we are using one or more routines from the standard input/output library (we will understand this better after we cover functions). In this case, we need this library so that we can use the "printf" command. The reason this is called a "preprocessor directive" is that things like this are handled by the compiler before it converts the rest of the code to machine code.

The program has no global or local declarations. The meaning of "declaration" will be made clear when variables are covered.

The "main" function includes two statements (there are no local declarations). One is the call to the function "printf" and the other is the "return" statement that ends the program.

This program does not have any functions other then "main".

One optional addition to the components of a C program, not listed in the structure of a general C program above, is that of **comments**. Comments can appear almost anywhere in a C program, and they are an example if internal program documentation. Comments are a description by the programmer to make the code easier to read for other people, or they might serve as reminders for the programmer at a later time.

Many programs will have a comment at the start of the program telling what the program does, a comment at the top of each function telling what the function does, and comments within the code to describe anything that isn't obvious.

A comment starts with the symbols "/*" and ends with the symbols "*/". These symbols tell the compiler that all text in between is a comment and should be ignored.
A comment can take up one line, part of a line, or multiple lines. If a comment is confined to a single line, and it stretches to the end of that line, you can start the comment with the symbol "//" and you do not need an end symbol. (This second notation for comments was actually added to the C standard in 1999, but it is also common.) These symbols are examples of **tokens**; a token is one or more symbols understood by the compiler that help it interpret your code.

Now we will rewrite the "Hello World!" program with comments:

```
/*
 * The following program prints the phrase "Hello World!"
 * to the screen on its own line.
 * Written by: NAME
 */

#include <stdio.h>

/* Global declarations would go here */

int main(void)
{

/* Local declarations would go here */

  printf("Hello World!\n"); // Display output
  return 0; // Returns 0 to the operating system
}
```

Normally such a simple program wouldn’t require all these comments; maybe just the one at the very top. The rest are here only as examples, and in fact they would be considered extraneous in a typical program. Notice the first comment takes up multiple lines. The asterisks in the middle are just for style; they are not required. There are also two examples of comments that take up there own line and two examples that take up part of a line (these could use either style of tokens).