Lecture 03
Bits, Bytes and Data Types

In this lecture

- **Computer Languages**
- **Assembly Language**
- **The compiler**
- **Operating system**
- **Data and program instructions**
- **Bits, Bytes and Data Types**
- **ASCII table**
- **Data Types**
- **Bit Representation of integers**
- **Base conversions**
- **1's compliment, 2's compliment and negative numbers**
- **Variable and storage classes – static, register, auto and extern**
- **Functions – pass by value, pass by reference**
- **Reading and Writing files**
- **Exercises**

Computer Languages

A computer language is a language that is used to communicate with a machine. Like all languages, computer languages have syntax (form) and semantics (meaning). High level languages such as Java are designed to make the process of programming easier, but programmer typically has little control over how efficient the code will run on the hardware. On the other hand, Assembly language programs are harder to write but are designed so that programmer can optimize the performance of the code. Then there is the machine language, the language the machine really understands. All computer languages are designed to communicate with hardware at the end. But programs written in high level languages may go through many steps of translations before being executed. Programs written in C are first converted to an assembly program (designed for the underlying hardware), which then in turn is converted to the machine language, the language understood by the hardware. There may be many steps in between. Machine language "defines" the machine and vice versa. Machine language instructions are simple. They typically consist of very simple instructions such as adding two numbers or moving data or jumping from one instruction to another. However, it is of course very difficult to write and debug programs in machine language.

Assembly Language

Programs written in a high level language such as C go through a process of translations that eventually leads to a set of instructions that can be executed by the underlying hardware. One layer of this program translation is the assembly language. A high level language is translated into assembly language. Each CPU/processor has its own assembly language. Assembly code is then translated into the target machine code. Assembly languages are human readable and contains very simple instructions. For example,

instructions such as Add two numbers, or move memory from one place to another or jump from one place to another etc.

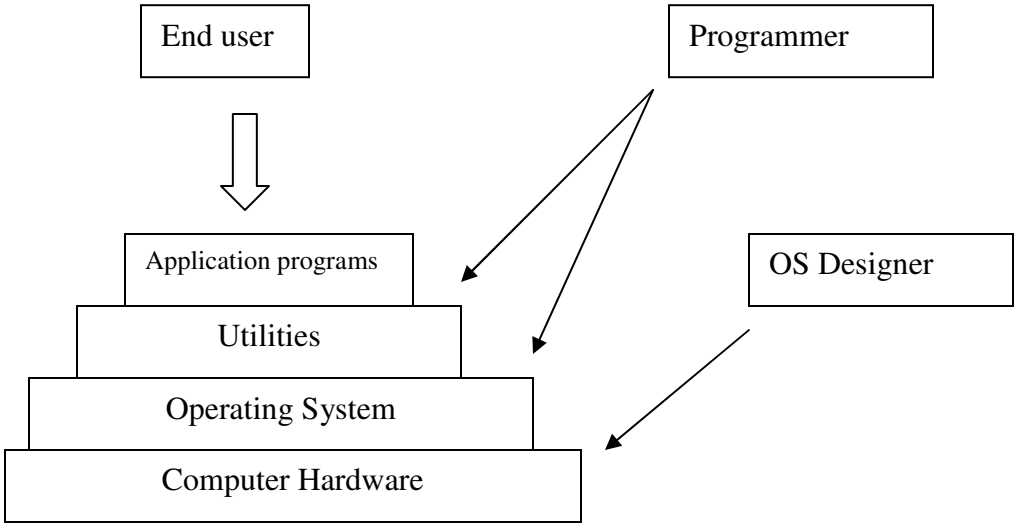
A high level instruction written in C such as
A = A + 1 could be translated into (hypothetical) Assembly as follows.

```
Mov R1, A          // move A to register 1
Inc R1             //  increment R1 by 1
Mov A, R1          // move R1 to A
```

Eventually this assembly code is mapped into the corresponding machine language so that the underlying hardware can carry out the instructions.

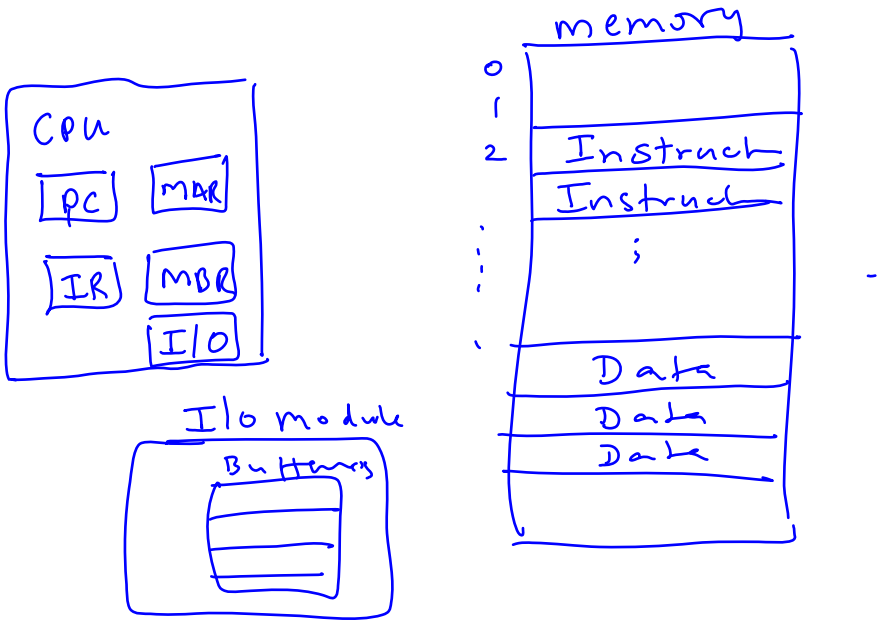
The Compiler
A compiler (such as gcc – GNU C compiler or lately GNU compiler collection) translates a program written in a high level language to object code that can be interpreted and executed by the underlying system. Compilers go through multiple levels of processing such as, syntax checking, pre-processing macros and libraries, object code generation, linking, and optimization among many other things. A course in compiler design will expose you to many of the tasks a compiler typically does. Writing a compiler is a substantial undertaking and one that requires a lot of attention to detail and understanding of many theoretical concepts in computer science.

Operating System
Each machine needs an Operating System (OS). An operating system is a software program that manages coordination between application programs and underlying hardware. OS manages devices such as printers, disks, monitors and manage multiple tasks such as processes. UNIX is an operating system. The following figure demonstrates the high level view of the layers of a computer system. It demonstrates that the end users interface with the computer at the application level, while programmers deal with utilities and operating system level. On the other hand, an OS designed must understand how to interface with the underlying hardware architecture.



Data and Program Instructions

All data and program instructions are stored as sequences of bytes in the memory called Random Access Memory (RAM). Typically data and instructions are stored in specific parts of the RAM as directed by the OS/compiler. As programs are executed, each instruction is fetched from memory and executed to produce the results. To increase the speed of execution of a program, a compiler may use fast accessed memory locations such as registers and cache memory. There could be 8 registers in the machine with one called the zero register (containing the value zero for initializations). The following figure demonstrates the architecture of a uni-processor machine that contains a CPU, memory and IO modules.



Bits, Bytes and Data Types

A bit is the smallest unit of storage represented by 0 or 1. A byte is typically 8 bits. C character data type requires one byte of storage. A file is a sequence of bytes. A size of the file is the number of bytes within the file. Although all files are a sequence of bytes,m files can be regarded as text files or binary files. Text files are human readable (it consists of a sequence of ASCII characters) and binary files may not be human readable (eg: image file such as bitmap file).

If you have a UNIX shell, you can type
> ls -l filename // to find out the size of the file(and many other things).
Eg: -rw-r--r-- 1 guna staff **11977** Feb 24 2004 joel.txt

Standard units of memory
1000 bytes = 1 Kilobytes(KB)
1000 KB = 1 megabyte (MB)
1000MB = 1 Gigabyte(GB)
1000 GB = 1 Terabyte(TB)
1000 TB = 1 Petabyte(PB)

Each data byte can be represented using an ASCII (or extended ASCII) value. An ASCII table is given below. Standard ASCII table assigns each character to a numerical value. For example 'A' = 65 and 'a' = 97. Printable standard ASCII values are between 32 and 126. The 8^{th} bit in the byte may be used for parity checking in communication or other device specific functions.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Each ASCII value can be represented using 7 bits. 7 bits can represent numbers from 0 = 0000 0000 to 127 = 0111 1111 (total of 128 numbers or 2^7)

Data Types

C has all the standard data types as in any high level language. C has **int, short, long, char, float, double**. C has no boolean data type or string type. C has no Boolean type but 0 can be used for false and anything else for True. A C string is considered a sequence of characters ending with null character **'\0'**. We will discuss more about strings later. You can read more about data types in K&R page 36.

An integer is typically represented by 4 bytes (or 32-bits). However this depends on the compiler/machine you are using. It is possible some architectures may use 2 bytes while others may use 8 bytes to represent an integer. But generally it is 4 bytes of memory. You can use **sizeof(int)** to find out the number of bytes assigned for int data type.

For example:
printf("The size of int is %d \n", sizeof(int));
prints the size of an integer in the system you are working on.

Find out the sizes of data types in unix.andrew.cmu.edu.

Bit Representation of Integers

If you take a low level look at an integer, this is how integer with value 10 is represented using 4 bytes (32-bits) in the memory:

00000000 00000000 00000000 00001010

Highest bit is the signed bit. Unsigned numbers uses the highest order bit as well to store the value of the number and hence doubling the range of values. To understand this concept, assume a signed number represented using 8 bits.

0111 1111 - What is the value of this?

1111 1111 – What is the highest value that can be represented if all 8 bits are used for the number?

Base Conversions

Understanding different bases is critical to understanding how data is represented in the memory. We consider base-2 (binary), base-8 (octal), base-10(decimal) and base-16(hexadecimal). A number can be represented in any of the bases stated above. The following digits are used in each base.

Base-2 - 0, 1
Base-8 - 0, 1, 2, 3,…, 7
Base-10 – 0, 1, 2, …, 9
Base-16 - 0,1, 2,…., 9, A, B, C, D, E, F

A number that is in base-10 such as 97 can be written in base-2 (binary) as follows. To convert the number to binary, find the sums of the powers of 2 that makes up the number, that is 97 = 64 + 32 + 1, and then represent this number using a binary pattern such as follows.
97 = 01100001
Each number can be converted from binary(base-2) to any other base such as octal(base-8), decimal(base-10) or hex (base-16).
Examples: 0000 1010 =
0101 1110 =

Examples: 70 =
300 =

Exercise: Write 456 in base-2, base-8, and base-16

Negative Numbers, One's Complement and Two's Complement

Signed data is generally represented in the computer in their two's complement. Two's complement of a number is obtained by adding 1 to its one's complement. So how do we find one's complement of a number? Here is the definition
One's complement of x is given by **~x**. Obtain the one's complement of a number by negating each of its binary bits. For example one's complement of 30 is (16-bit short)
30 = 16 + 8 + 4 + 2 = 00000001 11100000 ← binary 30

~30 = 11111110 00011111 ← its one's complement

The two's complement of the number is obtained by adding 1 to its one's complement. That is, the two's complement of 30 is obtained as follows.

11111110 00011111
+ 1

11111110 00100000

Hence -30 is represented as its two's complement, that is **~30 + 1 = 11111110 00100000**

Exercise: Perform binary addition of 34 + (-89) using two's complement of the negative number.

Variables

A C variable can be declared as follows

int x = 10;

This defines a variable x of type int (32-bits) and assigns 10 as the initial value. It is possible to find both the value and a pointer to (address of) the variable x. The value and the address of the variable can be found using:

printf("The value is %d and the address is %x \n", x, &x);

The format statements **%d (decimal)** and **%x(hexadecimal)** are used to format the output. A variable declared as **int x** is called an **automatic** variables. Automatic variables are not initialized, and given a place in the runtime stack, and it is the programmer's responsibility to initialize the variables. When a variable goes out of scope, the memory is given back. In addition to automatic storage class, C variables can be declared to be **static, extern or register** variables.

Storage Class Specifiers

Auto, Static, Extern and Register Variables

Any variable defined inside a function or file are considered to be an auto variable unless specified otherwise. That is, the scope of the variable is within the function or file it is declared. For example, consider a file program.c that contains the following.

```
#include <stdio.h>
int n;      ← file scope below this line

int main() {
  int n;   ←main function scope only

}
```

```
int foo( ) {
  int n;   ← foo function scope only
}
```

Let us look at the scope of the variables declared in this file. The first **int n** that is declared just below the #include statement can be seen by any function below it. In other words, the location of the variable determines where the variable is meaningful. Any function declared above first **int n** will not be able to see the variable n. The variables n defined inside functions main and foo are only meaningful within the functions.

STATIC VARIABLES

Static variables can be declared **externally**, that is, outside of any function, or **internally**, that is inside a function. An **external static** variable declared outside of functions is visible to users of the file, but not to the functions in other files. An **internal static** variable declared inside a function **retains** its value during all function call to the function. For example, consider the file **program.c** that contains the following declarations.

```
#include <stdio.h>
static int n;     ← external static variable – not visible to any other file

int main() {
  foo();
  foo();
}

int foo( ) {
  static int n;   ← internal static variable – retains its value during multiple calls to foo
}
```

Functions can also be declared static, making them visible only to the source file where it is declared. Normally functions are visible to any part of the program. But by making them static, we can limit the scope of the function only to the source file where it is declared. For example,

```
static int foo ( ) {
    ----
}
```

Only has the scope of the file, where it is declared and cannot be seen by any other file.

EXTERN VARIABLES

Extern variables can be used to share value of a variable among many functions. Typically, all variables declared inside a function are local variables and they are created when function is invoked and released when function is exited. However, if we are to

share the same variable among many functions, then we can define the variable to be "extern" as follows.

int n; ← allocates storage for the external variable

```
int foo ( ) {
   extern int n;   ← refers to the n defined above
}

int foo2( ) {
  extern int n;  ← refers to the same n as above
}
```

It should be noted that the use of "extern" inside the function can be avoided if all functions using n are defined below the external definition of n.

The use of extern is more evident when "extern" variables can be shared across multiple files. For example, if program.c defines an external variable n, then program2.c and program3.c can refer to the value of the external variable by using the qualifier "extern". Extern variables therefore are useful for sharing variables among several files.

REGISTER VARIABLES
Some variables that may be accessed by the program frequently can be specified to be a "register" variable.

register int x;

This will request the compiler to consider allocating a register location, a fast access memory location, for the variable x. However, compiler may completely ignore this request. The use of a register variable depends on hardware restrictions such as number of registers available etc. If a register cannot be allocated for a particular variable, then the directive is ignored. However, the address of a register variable cannot be accessed regardless of whether it is placed in a register or not.

Functions

Functions are important part of C programming. Functions allow us to break the program into manageable units that can be individually developed and tested. Procedural decomposition or breaking the program into small procedures is the best way to develop C programs. It allows easy debugging by localizing debugging problems. We have seen many examples of using functions to make C programs better. C function prototypes are defined in the beginning of a file, so the source code can be compiled without having the function fully developed. For example,

```
void foo(int);

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

```
   int ptr;
   foo(ptr);
   …….
}
```

The above code can be compiled just with the prototype.

- **gcc –c main.c**

Passing Arguments

Arguments to functions can be passed **by value** or **by reference**. A value parameter is a safe way to pass the value (or copy) of any type of a variable. When an address of a variable is passed into a function, the reference parameter provides direct access to the "global" variable that was passed into the function. This is a very powerful concept as we can give access to a large data structure without actually making a copy of it in the runtime stack. Let us look at the concepts of pass by value and pass by reference in more detail.

Pass by value

A copy of the variable can be passed to a function by value. The original variable is not changed. A copy is placed on the runtime stack for use during the execution.

Example:

```
void foo(int x) { x++; }

int a = 12;
foo(x);
```

Although x is incremented inside the function foo, x will not affect the original value of a as x is a copy of the variable a.

Pass by Reference

The address of a variable can be passed to a function. This allows us to manipulate the original variable directly from the function. We will discuss this after we study more about pointers.

Writing a Program in C

Now you are ready to write a simple program in C (using emacs or vi editor or any other editor), compile using gcc, and run and debug the program. We will give you some notes on a C and Unix primer, but there is a good tutorial introduction to C in K&R chapter 1. Please read and do some of the activities. Note that C uses header files (.h) to link to predefined libraries. You can get a list of the header files used in the course from Bb → header files. I will continue to update this list as semester progresses. C header files are equivalent to Java interfaces where you can learn what specific standard C functions that you can get from a predefined C library.

A typical C program given in a single file looks like the following.

- Include all libraries
- Include your libraries
- Define macros
- Define any global variables
- Define Function Prototypes
- The main program
- The function definitions

Here is an example:

```
#include <stdio.h>
#include "mylib.h"
#define n 10
int size = 10;
int  foo(int , double);
…
int main(int argc, char* argv[]) {

    ….
  return 0;
}

int  foo(int x , double y) {
   …..

}
```

Reading and Writing with Files

Input and output is essential in any programming language. C has three standard files, stdin, stdout and stderr. Other files opened as follows.
Typing: **man fopen** -- produces the following

NAME
fopen, fdopen, freopen - stream open functions

SYNOPSIS
#include <stdio.h>

*FILE *fopen(const char *path, const char *mode);*
*FILE *fdopen(int fildes, const char *mode);*
*FILE *freopen(const char *path, const char *mode, FILE *stream);*

DESCRIPTION
The fopen function opens the file whose name

is the string pointed to by path and associates a stream with it.

Here is an example of how to read values from a file and write the output to stdout.

```
FILE* fp  = fopen("filename", "r");
while (fscanf(fp, "%d", &num) > 0)  {
   fprintf(stdout, "%d", num);
}
```

We used two functions, fscanf and fprintf to read and write values to/from file. To find more information about fscanf and fprintf type:

% man fprintf
% man fscanf

For the following exercises, you may want to understand how to do following activities in C

- Read an int from stdin (keyboard)
- Write something to stdout (terminal)
- Formatting output - %d (decimal), %c (char), %s (string or char*) – see page 154 K& R – table 7.1

Further Readings

[1] K & R – chapters 2.1 – 2.6, 4.1

Exercises

3.1. Write a C program that can show the number of bytes used by each of standard data types, such as int, double, etc (hint: use sizeof). Use this information to understand the size of the maximum and minimum signed integers. (hint: INT_MIN, INT_MAX). Look for same in other data types. Include <limits.h> and <float.h>. See Bb header files links.

3.2. Write a C program to find the storage locations (or addresses) of 3 variables (int, double, string or char*) you declare. The values will be given in hexadecimal notation.

3.3 Write a program that reads an integer from stdin and output its decimal, binary, hexadecimal and octal representations.

See: democode/exerercise code to see the solutions