C Class Topic 1

Basic Data types

C has a concept of '*data types*' which are used to declare a variable before its use.

The declaration of a variable will AUTOMATICALLY assign storage for the variable and define the type of data that will be held in the location and removed when the program ends or the function ends if defined in a function. This is termed and AUTOMATIC variable.

int - data type

int is used to define integer numbers.

```
{
    int Count;
    Count = 5;
}
```

float - data type

float is used to define floating point numbers.

```
{
    float Miles;
    Miles = 5.6;
}
```

double - data type

double is used to define BIG floating point numbers. It reserves twice the storage for the number. On PCs this is likely to be 8 bytes.

```
{
    double Atoms;
    Atoms = 2500000;
}
```

char - data type

char defines characters.

```
{
    char Letter;
    Letter = 'x';
}
```

Modifiers

The three data types int, float and double have the following modifiers.

- short
- long
- signed
- unsigned

The modifiers define the amount of storage allocated to the variable. The amount of storage allocated is not cast in stone. ANSI has the following rules:

```
        short int <=    int <= long int
            float <= double <= long double
```

What this means is that a 'short int' should assign less than or the same amount of storage as an 'int' and the 'int' should be less or the same bytes than a 'long int'. What this means in the real world is:

Type	Bytes	Bits	Range	
short int	2	16	-32,768 -> +32,767	(16kb)
unsigned short int	2	16	0 -> +65,535	(32Kb)
unsigned int	4	16	0 -> +4,294,967,295	(4Gb)
int	4	32	-2,147,483,648 -> +2,147,483,647	(2Gb)
long int	4	32	-2,147,483,648 -> +2,147,483,647	(2Gb)
signed char	1	8	-128 -> +127	
unsigned char	1	8	0 -> +255	
float	4	32		
double	8	64		
long double	12	96		

These figures only apply to 32 bit generation of PCs. Newer 64bit PC's can allocate even larger data types definitions using compiler switches.

You can find out how much storage is allocated to a data type by using the “sizeof” operator.

&:

The prefix of & with a variable returns the address of the variable in memory. For example an integer i would have the address returned by &i.

The FOR keyword.

Say you wanted to print all the numbers between 1 and 10, you could write:

```
main()
{
    int count=1;

    printf("%d\n", count++);
    printf("%d\n", count++);
    printf("%d\n", count++);
    printf("%d\n", count++);
```

```
    printf("%d\n", count++);
    printf("%d\n", count++);
    printf("%d\n", count++);
    printf("%d\n", count++);
    printf("%d\n", count++);
    printf("%d\n", count++);
}
```

As you can see this program would NOT be very practical if we wanted 1000 numbers. The problem can be solved with the **for** statement as below.

```
main()
{
    int count;

    for ( count=1 ; count <= 10 ; count++) printf("%d\n", count);
}
```

The **for** statement can be broken down into 4 sections:

count=1
: is the initialization.

count <= 10
: An expression. The for statement continues to loop while this statement remains true (not equal to zero)

count++
: increment or decrement.

printf("%d\n", count)
: the statement to execute.

Repeating several lines of code. The previous example showed how to repeat ONE statement. This example shows how many lines can be repeated.

```
main()
{
    int count, sqr;

    for ( count=1 ; count <= 10 ; count++)
    {
        sqr=count * count;
        printf( " The square of");
        printf( " %2d", count);
        printf( " is %3d\n", sqr);
    }
}
```

The **{** and **}** following the **for** statement define a block of statements.
The **for** statement performs the following functions while looping.

```
for (expression_1 ; expression_2 ; expression_3) statement ;
```

1. Executes expression_1.
2. Executes statement.
3. Executes expression_3.
4. Evaluates expression_2.
    - If TRUE, Jumps to item 2.
    - If FALSE, Stops looping.

Any of the three expressions can be missing, if the first or third is missing, it is ignored. If the second is missing, is is assumed to be TRUE.

The following example is an infinite loop:

```
main()
{
    for( ; ; ) puts(" Linux rules!");
}
```

malloc (memory allocation) is used to dynamically allocate memory at run time vs Automatically. Possible uses for this function are:

- Read records of an unknown length.
- Read an unknown number of database records.
- Linked Lists

The simplest way to reserve memory is to code something like: (using AUTOMATIC variables)

```
main()
{
    char string[1000];

    strcpy (string, "Some text");
}
```

The example above has three problems:

- If the data is less than 1000 bytes we are wasting memory.
- If the data is greater than 1000 bytes the program is going to crash by overwriting other memory possibly code.
- The 1000 bytes are reserved though out the life of the program. If this was a long running program that rarely used the memory, it would again be wasteful.

malloc allows us to allocate exactly the correct amount of memory and with the use of free only for the time it is required.

```
Library:   stdlib.h
Prototype: void *malloc(size_t size);
Syntax:    char * String;

                String = (char *) malloc(1000);
```

Looking at the example syntax above, 1000 bytes are reserved and the pointer **String** points to the first byte. The 1000 bytes are NOT initialized by malloc. If the memory is NOT available, a NULL pointer is returned. Please note, the cast is required to return a pointer of the correct type.