Lecture-2

Basic structure of a C program:

1. Including header files
2. Macro definition
3. Global declaration
4. Main declaration
   {
   local declaration
   statement sequence
   other function call
   }
5. User defined function

Statements are parts of your program that actually performs operation. Statements are contained within functions.

All C statements end with a semicolon. C does not recognize the end of the line as a terminator. This means there are no constraints on the position of statements within a line. Also you may place two or more statements on one line.
The standard library contains functions to perform I/O, string manipulation, mathematics and much more.

So, we can start our first C program that will print "This is our first C program" to the standard output device.

```
#include<stdio.h>
void main(void)
{
printf("This is our first C program");
}
```

1. include is a pre processor directive that tells the compiler to include library functions contained in stdio.h file with your program.
2. main function
3. { beginning of main function
4. library function printf, that prints string within " "
5. } end of a function.

Character that can be used in C:

1. Alphabets: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z.
   A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.
2. Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
3. Special symbols: ~ ` ! @ # $ % ^ & * ( ) _ - + = | \ { } [ ] ; : " ' < > , . ? /

White Space: blank, new line, tab.

In a program the smallest individual units are known as C-token

5 types of tokens in C: keywords, identifiers, constants, operators & punctuators.

Keywords: Keywords are specially reserved words that have strict meaning as individuals token in C.

ANSI standard: 32 keywords.

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	far	for
goto	if	int	long	near
register	return	short	signed	static
struct	switch	typedef	union	unsigned
void	while.			

Additional keywords like asm etc. are in turbo C.

Identifiers: Identifiers are user defined name of objects in a program. An identifier is composed of a sequence of letters, digits and underscores (_). An identifier must begin with a letter or underscore. Ex: sb_int, cse_aust, s_99, _x11.

Constants: C constants can be divided into 2 major categories.

1. Primary constants.
   a. Integer constants: 123, +234, -876 etc.
   b. Real constants: 23.4, -45.6, 0.56, 4.2e7, 9.3e-4, -8.1e4, -2e-4 etc.
   c. Character constants: 'A', 'b', 'x', 'S', '&' etc.

   * Every keyboard key is a character constant.

2. Secondary constants.
   a. Array
   b. Pointer
   c. Structure
   d. Union
   e. Enum

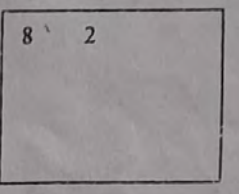
Lecture - 3

Operators:

1. Binary operator: + , - , / , * , %

Example:

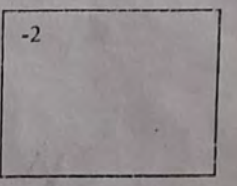
```
void main(void)
{
int a = 5, b= 3;
printf("%d\t%d",a+b,a-b);
}
```



2. Unary operator: + ,-

Example:

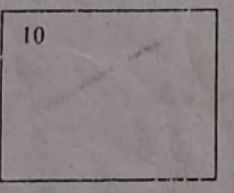
```
void main(void)
{
int a = 5, b= 3;
printf("%d",-a+b);
}
```



3. Relational operator: > , > = ,< , < = , = = , ! = (not equal)
   true will return 1 and false will return 0.

Example:

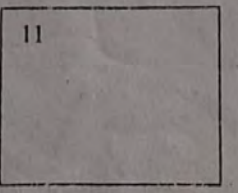
```
void main(void)
{
int a = 5, b= 3;
printf("%d%d",a>b,a= =b);
}
```



4. Logical operator: && (AND) , || (OR) , ! (NOT)
   true will return a nonzero value and false will return 0.

Example:

```
void main(void)
{
int a = 5, b= 3;
printf("%d%d",(a>b)&&(a!=b),!(b>a)||(b= =a));
}
```



5. Bitwise operator: & (AND), | (OR), ^ (XOR), <<, >>, ~

A	b	A AND b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

a	NOT a
0	1
1	0

Example:

```
void main(void)
{
int a = 40, b= 15;
printf("%d", ~(((a>>2) & (b<<3)) ^ (a | b)));
}
```

-40

6. Increment and decrement operator: ++, --

Example:

```
void main(void)
{
int i = 5;
printf("%d%d",i++,i--);
printf("%d",i);
}
```

455

7. Assignment operator: =, +=, *=, -=, /=, %=, &=, |=, ^=, <<=, >>=

Example:

```
void main(void)
{
int a=5,b=3;
a+=b; b*=a;
printf("%d%d",a,b);
}
```

824

Punctuators: ` @ # $ ( ) { } [ ] ; : " ' , . ?

Lecture - 4

C Instructions:

1. Type Declaration Instruction:
   ```
   int x ;
   float y ;
   char z ;
   ```

2. Input / Output Instruction
   ```
   Scanf(…);
   printf(…);
   inportb(…);
   outportb(…);
   ```

3. Arithmetic Instruction
   ```
   c = b + d ;
   p = m * n / q ;
   ```

4. Control Instruction
   a. Sequence control instruction
   b. Selection or decision control instruction
   c. Repetition or Loop control instruction
   d. Case control instruction

Data Types in C:

Type	Keyword	Size	Range
Void	void	0 byte	0
Character	char	1 byte	unsigned : 0 to (2^8-1) signed : (-2^7) to $+(2^7-1)$
Integer	int	2 bytes (for DOS) 4 bytes (for Unix)	unsigned: 0 to $(2^{16}-1)$ signed : (-2^{15}) to $+(2^{15}-1)$
Floating point	float	4 bytes	unsigned : 0 to $(2^{32}-1)$ signed : (-2^{31}) to $+(2^{31}-1)$
Double	double	8 byte	unsigned : 0 to $(2^{64}-1)$ signed : (-2^{63}) to $+(2^{63}-1)$

Type modifiers: signed, unsigned; long, short.

Size of short is given into the above data types table.

Size of long = 2n bytes, where short = n bytes.

*Exception: Size of long double = 10 bytes

CSE 101 Lecture 4

Constants:

integer: 234
long integer: 1234l / 1234L unsigned
integer: 1234u / 1234U unsigned long
integer: 1234ul / 1234UL float constant:
123.4f / 123.4F
double constant: 123.4
long double: 12.34l
hex constant: 0xff / 0XFF
octal constant: 077
character constant (ASCII): 'a'

integer: 234
long integer: 1234l/1234L unsigned
integer: 1234u/1234U unsigned long
integer: 1234ul/1234UL
float constant: 123.4f/123.4F
double constant: 1234
long double: 12.34l
hex constant: 0xff/0XFF
octal constant: 077
character constant (ASCII): 'a'

ASCII = American standard code of Information & Interchange.

Interpreting characters (scape sequence):

\a → alert (bell)
\b → backspace
\f → form feed
\n → new line
\r → carriage return
\t → horizontal tab
\v → vertical tab
\\ → back slash
\' → single quote
\" → double quote
\? → question mark
\0 → NULL

Example:

```
#include<stdio.h>
void main(void)
{
printf("\tHellow \nWorld");
}
```

```
    Hellow
World
```

Lecture 5

Format specifier: printf(...) converts, formats and prints its argument on the standard output under control of the format.

Format specifier	Printed as
%d , %i	Decimal
%c	Character
%o	Unsigned octal number
%x , %X	Unsigned hexadecimal number
%u	Unsigned integer
%ld	Long integer
%hd	Short integer
%lf	Double
%s	String
%Lf	Long double
%p	Address of variable
%%	To get output of an %
%f	Floating point number

Example 1:

```
void main(void)
{
int x,y;
printf("Enter two number");
scanf("%d%d",&x,&y);
printf("%d%d",x,y);
}
```

```
Enter two number5
6
56
```

Example 2:

```
void main(void)
{
int x = 1;
printf("Enter %d \n hellow ",x);
}
```

```
Enter 1
hellow
```

Example 3:

```
void main(void)
{
char x;
scanf("%c",&x);
printf("%d",x);
}
```

```
A
65
```

Example 4:

```
void main(void)
{
int a = 5;
printf("Address =  %p \nData = %d ",&a,a);
}
```

Address = FFF0
Data = 5

Example 5:

```
void main(void)
{
int x = 128;
char c ; c = x;
printf("%d",c);
}
```

-128

Example 6:

```
void main(void)
{
printf("%d ",32768);
}
```

-32768

Type conversion: (type) expression

Example 1:

```
void main(void)
{
int x ,y ;
x = 3;
y = 2;
printf("%d %d %f %d", x + y, x - y, (float) x / y, x % y);
}
```

511.5000001

Example 2:

```
char ch;
int i;
float f;
double d;
ch / i + f * d – f + i = double
```

Lecture – 6

Operator precedence and associativity:

1. ( ) , [] → Left to right
2. + + (postfix) , - - (postfix) → Right to left
3. ! (not) , ~ (1's complement) , + (unary) , - (unary) , + + (prefix) , - - (prefix) , & (address) , * (indirection) , sizeof → Right to left
4. * , / , % (modulus)→ Left to right
5. + (binary) , - (binary) → Left to right
6. << (shift left) , >> (shift right) → Left to right
7. < , < = , > , > = → Left to right
8. = = , ! = → Left to right
9. & (bitwise AND) → Left to right
10. ^ (bitwise XOR) → Left to right
11. | (bitwise OR) → Left to right
12. && (logical AND) → Left to right
13. || (logical OR) → Left to right
14. ? : (a ? x : y) → Right to left
15. = , * = , / = , % = , + = , - = , & = , ^ = , | = , << = , >> = → Right to left
16. , (comma) → Left to right

Dealing with expressions:

Example 1:

```
void main(void)
{
int x=2,n=2;
x=n++;
printf("%d",x);
x=++n;
printf("%d",x);
}
```

24

Example 2:

```
void main(void)
{
int x=2, y=3;
x*=y;
printf("%d",x);
x=x*y;
printf("%d",x);
x*=y+1;
printf("%d",x);
}
```

61872

void main(void)

Example 3:

```
void main(void)
{
int x=2, y=2;
printf("%d",x++>y);
printf("%d", x&077);
}
```

03

Example 4:

```
void main(void)
{
int x=2;
printf("%d",!(x)&&x++);
printf("%d", x);
}
```

02

!(x) is false thus it returns 0. If one operand of AND operation is false then the result will be false. In this case complier will return 0 without executing the full expression. So x++ will not execute in this case. This is called "short circuit evaluation".

Example 5:

```
void main(void)
{
int a,b,c,p,q; a=b=c=0;
printf("%d%d%d",a,b,c);
a=(p=2)+(q=3);
p*=c=a+5;
printf("%d%d%d", a,c,p);
}
```

000 1020

Example 6:

```
void main(void)
{
int x=5, y=2;
printf("%d",x+++y);
}
```

7

Mark operator: exp1 ? exp2 : exp 3

Example:

```
void main(void)
{
int x=5,y;
y= x>5 ? x-=2 : x+=2;
printf("%d%d",x,y);
}
```

77

Lecture – 7

Expression: An expression is the combination of functions, operators, commas, variables, identifiers etc. Ex: x = 0 , x + +.

Statement: An expression as x = 0 , x + + or printf(…) becomes a statement when it is followed by a semicolon(;) as in

```
x + + ; x =
0 ;
printf(…) ;
```

Braces { and } are syntactically equivalent to a single statement.

if … else: if … else statement is used to express decisions. Formally the syntax is :

```
if (expression)
        statement1
else
        statement2
```

Here the else part is optional. The expression inside if … else statement expects a value. If the value is nonzero then it is true, If the value is 0 then it is false.

Example 1:

```
void main(void)
{
int x ,y=0, a=2, b=3 ;
if(y>0)
     x = a;
else
     x = b;
printf("%d",x);
}
```

3

Example 2:

```
void main(void)
{
int x =1 ;
if(x - -)
     printf("Hellow");
printf(" friend");
}
```

Hellow friend

Example 3:

```
void main(void)
{
int x =1 ;
if(x - -)
    printf("Hellow");
if(x + +)
    printf(" everyone");
}
```

Hellow

else … if:

```
if(expression 1)
    statement 1
else if(expression 2)
    statement 2
else if(expression 3)
    statement 3
    .
    .
    .
else
    statement n
```

else … if checks its expression sequentially. When it found one expression is true then it will not check other expressions within that else … if block.

Example 1:

```
void main(void)
{
int x =2 ;
if(x - -)
    printf("1");
else if(x - -)
    printf("2");
else if(x - -)
    printf("3");
else
    printf("None");
printf("%d",x);
}
```

11

Example 2:

```
void main(void)
{
int x =1 ;
if(x - -)
    printf("1");
if(x - -)
    printf("2");
else if(x - -)
    printf("3");
if(x - -)
    printf("4");
printf("%d",x);
}
```

134 -3

Example 3:

```
void main(void)
{
int x =0,y =1 ;
if(x = 0)
    if(y = =2)
        printf("Hellow");
    else
        printf("Welcome");
printf("Nothing");
}
```

Nothing

Example 4:

```
void main(void)
{
int x =0 ;
if(x = = 0)
    printf("Hellow\n");
printf("Something\n");
}
```

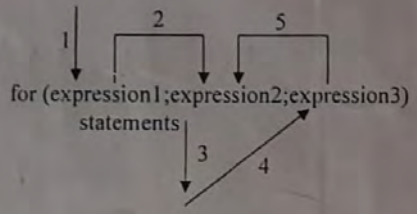
Hellow
Something

Lecture - 8

Loop: Loops are used in C for repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. There are three methods in C for repeating a part of program.

i) Using a for statement
ii) Using a while statement
iii) Using a do … while statement

for Loop: Syntax of for loop is given below:



Expression1, expression2 and expression3 are normally used for initialization, loop condition and loop counter increment respectively. Expression2 returns a value which may true or false. Any non-zero value means true and zero means false in this case. All three expressions (phase) are optional in for loop.

Example 1:

```
void main(void)
{
int x ,n ;
printf("Please enter a number ");
scanf("%d",&n);
for (x=1; x<=n; x=x+2)
    printf("%d",x);
}
```

```
Please enter a number 5
135
```

Example 2:

```
void main(void)
{
int x =1 ;
scanf("%d",&x);
for (x=5; x>=1; x - -)
    printf("%d",x);
}
```

```
54321
```


Example 3:

```
void main(void)
{
int x =1 ;
for (x=5; x - -; )
    printf("%d",x);
}
```

```
43210
```

Example 4:

```
void main(void)
{
int i ;
for ( i=2; i <= i; i++)
    printf("%d",i);
}
```

```
2345678910112131415
1617.....
.....
.....
.....
.....
.....
```

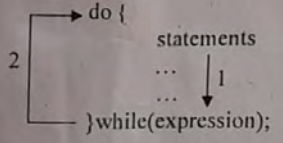
Programming will run infinite times in this case. This is called infinite loop. Be careful about infinite loops when programming using loops.

Example 5:

```
void main(void)
{
int x =5 ;
for (;x - -;)
    printf("%d",x);
}
```

```
43210
```

do … while loop: Syntax of do…while loop is given below:



Example 1:

```
void main(void)
{
int x =1 ;
do{
    printf("%d",x);
    x + +;
}while(x <= 3);
}
```

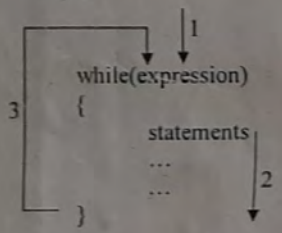
```
123
```

Example 2:

```
void main(void)
{
int x =3 ;
do{
      printf("%d",x);
}while(x - -);
printf("%d",x);
}
```

3210-1

while loop: Syntax of while loop is given below:



Example 1:

```
void main(void)
{
int x =1 ;
while(x <= 3)
{
     printf("%d",x);
     x = x+1;
}
}
```

123

Example 2:

```
void main(void)
{
int x = 5 ;
while(x-2)
{
     printf("%d",x--);
}
}
```

543

Lecture - 9

switch: The switch statement is used to test multi way decisions that tests whether an expression matches one of a number of constants. When a match is found a statement/ block of element is executed.

```
switch (expressions)
{
      case constexp1:
            statements
            break ;
      case constexp2:
            statements
            break ;
            .
            .
            .
      case constexpN:
            statements
            break ;
      default:
            statements
}
```

Example 1:

```
void main(void)
{
      int x ;
      scanf("%d",&x);
      switch(x)
      {
            case 1 : printf("One\n");
                     break ;
            case 2 : printf("Two\n");
                     break ;
            case 3 : printf("Three\n");
                     break ;
            default : printf("Other\n");
      }
}
```

3
Three

Example 2:

```
void main(void)
{
        char c ;
        scanf("%c",&c);
        switch(c)
        {
                case 'a' : printf("Vowel\n");
                           break ;
                case 'e' : printf("Vowel\n");
                           break ;
                case 'i' : printf("Vowel\n");
                           break ;
                case 'o' : printf("Vowel\n");
                           break ;
                case 'u' : printf("Vowel\n");
                           break ;
                default : printf("Consonent\n");
        }
}
```

P
consonent

Nested switch: We can use switch statement within a switch statement which is called nested switch.

Example:

```
switch(x)
{
        case 0 :
                switch(y)
                {       case 1: printf(" … ");
                                break ;
                        case 2 : printf(" … ");
                                break ;
                        default : printf(" … ");
                }
        case 1 :
                switch(z)
                {       case 0: printf(" … ");
                                break ;
                        case 1 : printf(" … ");
                                break ;
                        default : printf(" … ");
                }
        break ;
        default : printf(" … ");
}
```


break statement: break statement has two uses: The first is to terminate a case in the switch block and the second is to force immediate termination of a loop.

Example :

```
void main(void)
{
        int i ;
        for (i=1; i<=100; i + +)
        {
                printf("%d",i);
                if (i = = 10)
                        break ;
        }
}
```

12345678910

continue statement: Continue forces next iteration in a loop

Example :

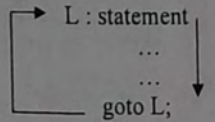
```
void main(void)
{
        int i ;
        for (i=1; i<=5; i + +)
        {
                if (i % 2 = = 0)
                        continue ;
                printf("%d",i);

        }
}
```

135

Lecture – 10

goto statement: We can jump from one statement to any statement by using **goto**.



Example:

```
void main(void)
{
int i = 1;
L :
printf("%d",i++);
if (i<=5)
    goto L ;
}
```

12345

Nested loop:

Example:

```
void main(void)
{
int i,j ;
for (i = 1; i<=3; i ++)
    for (j = 1; j<=2; j ++)
        printf("hellow\n");
}
```

hellow
hellow
hellow
hellow
hellow
hellow

More programs on loop:

Example 1:

```
void main(void)
{
int i ;
for (i = 1; i<=3; i ++)
    printf("%d",i);
}
```

123

Example 2:

```
void main(void)
{
int i =1;
while(i<=3)
    printf("%d",i ++);
}
```

123

Example 3:

```
void main(void)
{
int i =1;
do{
    printf("%d",i ++);
}while(i<=3);
}
```

123

Example 4:

```
void main(void)
{
int i ;
for (i = 1; i<=3; i ++);
    printf("%d",i);
}
```

4

Example 5:

```
void main(void)
{
int i ;
for (i = 32766; i<=32768; i ++)
    printf("%d",i);
}
```

3276632767-32768
.....
.....
.....
.....

It will be an **infinite loop** because maximum value of i can be 32767. So i<=32768 is an always true condition which will create the **infinite loop**.

Example 6:

```
void main(void)
{
int i,j ;
for (i = 1,j = 2; j<=3; i ++)
    printf("%d",i);
}
```

1234567.....
.....
.....
.....
.....

the variable j is not increased by the loop counter which will create **infinite loop**.

Example 7:

```
void main(void)
{
int i, j, x = 0 ;
for (i = 0; i < 3; i ++)
    for(j = 0; j < i;j++)
        switch (i + j -1)
        {
            case 1 : x + = 3;
                     break;
            case -1 : x + = 1;
                     break;
            case 0 : x - = 2;
                     break;
            default : x + = 1;
        }
        printf("%d",x);
}
```

2

Suppose ASCII code of 'A' is 65

Output

-2
1
2
2

Lecture – 11

User defined function: A function is a self contained block of statements that perform a coherent task of some kind. Every C program can be though of as a collection of these functions.

```
return_type fname (arguments);

void main (void)
{
    ...
    ...
}

return_type fname (arguments)
{
    ...
    ...
}
```

Advantages of function:

- Reduce redundancy of logic
- Easy to debug
- Easy to write.

Example 1:

```
int sum(int p, int q);

void main(void)
{
    int x =1, y=2, z ;
    z = sum(x,y);
    printf("%d",z);
}

int sum(int p, int q)
{
    return p+q;
}
```

3

Example 2:

```
void one(void);
void two(void);


void main(void)
```

```
{
        one();
        two();
}

void one()
{
        printf("In One\n");
        two();
}

void two()
{
        printf("In Two\n");
}
```

```
In One
In Two
In Two
```

Example 3:

```
void swap(int p, int q);

void main(void)
{
        int x =2, y=3 ;
        printf("Before swap x = %d and y = %d\n",x,y);
        swap(x,y);
        printf(After swap x = %d and y = %d",x,y);
}

void swap(int p, int q)
{
        int temp;
        temp = p;
        p = q ;
        q = temp;
}
```

```
Before swap x = 2 and y = 3
After swap x=2 and y = 3
```

Example 4:

```
void swap(int x, int y)
{
        int temp;
        temp = x;
        x = y ;
        y = temp;
}

void main(void)
{
```

```
Before swap x = 2 and y = 3
After swap x=2 and y = 3
```

```
    int x =2, y=3 ;
    printf("Before swap x = %d and y = %d\n",x,y);
    swap(x,y);
    printf(After swap x = %d and y = %d",x,y);
}
```

Variable x and y are not same within main() and swap() because of local. declaration. If we declare variables globally then x and y variable will be recognized by both main() and swap(). In this case, output will be :

```
Before swap x = 2 and y = 3
After swap x=3 and y = 2
```

Example: 5

```
void square(int n)
{
    printf("The square of %d is %d\n", n, n*n);
}

void main(void)
{
    int index;
    for(index=1; index<=5; index++)
        square(index);
}
```

```
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

Example: 6

```
void even(int p)
{
    int x;
    for (x =1; x<=p ; x ++)
        if(x % 2 = = 0)
            printf(" %d\n", x);
}

void main(void)
{
    int n;
    printf("Please enter a number");
    scanf("%d",&n);
    even(n);
}
```

```
Please enter a number 10
2
4
6
8
10
```

```
printf("\nValue of v = %u", p);
}
```

Lecture 12

Recursion: A function is called **recursive** if a statement within the body of a function calls the same function. Sometimes called **circular definition**, recursion is thus the process of defining something in terms of itself.

Example: 1

```
int fib(int m);
void main(void)
{
int i, n;
printf("Please enter a number ");
scanf("%d",&n);
for(i =1; i <=n; i ++)
        printf("%d\n",fib(i));
}

int fib(int m)
{
if(m<=2)
        return 1;
else
        return fib(m-1) + fib(m-2);
}
```

```
Please enter a number 7
1
1
2
3
5
8
13
```

Example: 2

```
int fact(int m);
void main(void)
{
int n;
printf("Please enter a number ");
scanf("%d",&n);
printf("The factorial is %d\n",fact(n));
}

int fact(int m)
{
int f ,s =1;
for(f =1; f < =m; f ++)
        s * = f ;
return s ;
}
```

```
Please enter a number 5
The factorial is 120
```

Example: 3

```
int fact(int m);
void main(void)
{
int n;
printf("Please enter a number ");
scanf("%d",&n);
printf("The factorial is %d\n",fact(n));
}

int fact(int m)
{
if(m==1)
        return 1;
else
        return m*fact(m-1);
}
```

```
Please enter a number 5
The factorial is 120
```

Example: 4

```
int count_dn(int count);
void main(void)
{
int n;
printf("How many times ? ");
scanf("%d",&n);
count_dn(n);
}

int count_dn(int count)
{
count--;
printf("The value of count = %d\n",count);
if(count > 0)
        count_dn(count);
printf("Now count = %d\n", count);
}
```

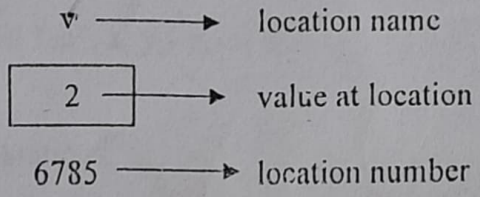
```
How many times ? 4
The value of count = 3
The value of count = 2
The value of count = 1
The value of count = 0
Now count = 0
Now count = 1
Now count = 2
Now count = 3
```

Lecture 13

Pointer: A simple variable in a program is stored in certain number of bytes at a particular memory locations or address in the machine. Pointers are used in program to access memory and manipulate address.

Consider the declaration,

```
int v = 2 ;
```



Example 1:

```
void main(void)
{
int v = 2 ;
printf("Address of v = %u",&v);
printf("\nValue of v = %d",v);
printf("\nValue of v = %d",*(&v));
}
```

```
Address of v = 6785
Value of v = 2
Value of v = 2
```

We can collect the address of a variable into another variable by saying,

```
p = & v;
```

At first we have to declare p as a variable which will store the address of an integer value.

```
int * p ;
```

Example 2:

```
void main(void)
{
int v = 2, * p ;
p = & v ;
printf("Address of v = %u",&v);
printf("\nAddress of p = %u",&p);
printf("\nValue of p = %u",p);
printf("\nValue of v = %u",v);
printf("\nValue of v = %u",*(&v));
printf("\nValue of v = %u",* p);
}
```

```
Address of v = 6785
Address of p = 3275
Value of p = 6785
Value of v = 2
Value of v = 2
Value of v = 2
```

Example 3:

```
void main(void)
{
int x = 2, y =3, * p, * q ;
p = & x ;
q = & y ;
p = q ;
printf("%d %d %d %d", x, y, * p, * q );
* p = 3;
* q = 4;
x = y;
printf("\n%d %d %d %d", x, y, * p, * q );
}
```

2333
4444

3334

Problem

Look at the following declarations,

```
int *i ;
char *ch ;
float *f ;
```

The declaration **float *f** does not mean that f is going to contain a floating point value. What it means is, f is going to contain the address of a floating point value.

Concept of pointers can be further extended. We can declare a pointer which may contains another pointer's address.

```
int x = 2, * p, ** q ;
p = & x ;
q = & p ;
```

v	p	q
2	6785	3275
6785	3275	7680

Function calls: Arguments of a function can be passed to functions in one of the two ways:

- Sending the values of the arguments (**Call by Value**)
- Sending the address of the arguments (**Call by Reference**)

Call by Value: In this method 'value' of each actual arguments in the calling function is copied into corresponding formal arguments of the called function.

Example 4:

```
void swap(int x, int y);
void main(void)
{
int a = 10, b = 20;
swap(a, b);
printf("\na = %d b = %d",a,b);
}

void swap(int x, int y)
{
int temp;
temp = x;
x = y;
y = temp;
printf("\nx = %d y = %d",x,y);
}
```

```
x = 20 y = 10
a = 10 b = 20
```

Call by Reference: In this method the addresses of actual arguments in the calling function are copied into corresponding formal arguments of the called function.

Example 5:

```
void swap(int *x, int *y);
void main(void)
{
int a = 10, b = 20;
swap(&a, &b);
printf("\na = %d b = %d",a,b);
}

void swap(int *x, int *y)
{
int temp;
temp = *x;
*x = *y;
*y = temp;
}
```

```
a = 20 b = 10
```

Note that this program manages to exchange the values of **a** and **b** using their addresses stored in **x** and **y**.

Example 6:

```
void input(int *p, int *q);
int add(int x, int y);
void display(int value);
```

```
void main(void)
{
int x,y,sum;
input (&x, &y);
sum = add(x, y);
display(sum);
}

void input(int *p, int *q)
{
scanf("%d%d",p,q);
}

int add(int x, int y)
{
return x+y;
}

void display(int value)
{
printf("The sum = %d", value);
}
```

```
4
5
The sum = 9
```

Using call by reference intelligently we can make a function return more than one value at a time.

Example 7:

```
void areaperi(int r, float *a, float *p);
void main(void)
{
int radius;
float area, perimeter;
printf("Enter radius of a circle");
scanf("%d",&radius);
areaperi(radius, &area, &perimeter);
printf("Area = %f ",area);
printf("\nPerimeter = %f ", perimeter);
}

void areaperi(int r, float *a, float *p)
{
*a = 3.14 * r * r ;
*p = 2 * 3.14 * r ;
}
```

```
Enter radius of a circle 5
Area = 78.500000
Perimeter = 31.400000
```

Lecture 14

Array: Array is a list of variables, which are all of the same type and are referenced through a common name. An individual variable in the array is called an array element.

Advantages of array:

- Reduce ordinary variable
- Allocates memory sequentially

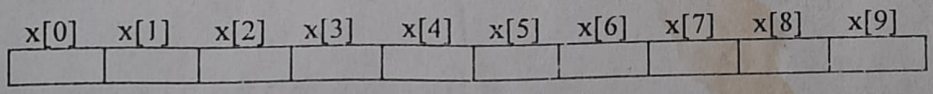
Disadvantages of array:

- Insertion difficult
- Deletion difficult
- Wastage of memory

Array declaration: We can declare an array in two ways.

- Static array
- Dynamic array

```
int x[10] ;
```



This is called static declaration of an array.

```
int *p, n;
scanf("%d",&n);
p = (int * ) malloc (n * sizeof(int));
```

This is called dynamic declaration of an array. By using (void*)malloc(int size) we can declare dynamic array.

One dimensional arrays: Say, int a[5];

Example 1:

```
void main(void)
{
        int a[5];
        for(i = 0; i< 5; i ++)
                scanf("%d",&a[i]);
        for(i = 0; i< 5; i ++)
                printf("%d",a[i]);
}
```

```
1
2
3
4
5
12345
```

Example 2:

```
void main(void)
{
        char *p, n, i ;
        scanf("%d",&n);
        p = (char *) malloc (n);
        for ( i = 0 ; i < n ; i ++)
                scanf("%d", ( p + i ));
        for ( i = 0 ; i < n ; i ++)
                printf("%d", *( p + i ));
}
```

```
5
1
2
3
4
5
12345
```

Example 3:

```
void main(void)
{
        int days[31], n, avg = 0, i ;
        printf(" How many days ? \n");
        scanf("%d",&n);
        for ( i = 0 ; i < n ; i ++){
                scanf("%d", & days[i] );
                avg = avg + days[i] ;
        }
        printf(" Average temperature = %f", (float)avg/n);
}
```

Consider the declaration, int (*any)(int, int).
This declaration indicates that any is a pointer of function which can take two integer type values as argument and also return an integer type value.

Example 4:

```
int sum(int a, int b);
void main(void)
{
        int a, b;
        int (*sab)(int, int);
        sab = sum;
        scanf("%d%d", &a, &b);
        printf("\n%d",sab(a,b));
}

int sum (int a, int b)
{
        return a+b;
}
```

```
5
4
9
```

Example 5:

```
void main(void)
{
        int a[6], sqr_a[6], *p, *q ;
        p = a ;
        q = sqr_a;
        for( i = 0 ; i < 5 ; i ++){
                *p = i + 1 ;
                p ++ ;
                *q = (i + 1)*(i + 1);
                q ++ ;
        }
        for( i = 0 ; i < 5 ; i ++)
                printf("%d/t%d\n"a[i],sqr_a[i]);
}
```

```
1    1
2    4
3    9
4    16
5    25
```

Passing array to a function: We can pass the whole array to a function by just sending the first element of that array.

Example 6:

```
void input(int *);
void output(int []);

void main()
{
        int  a[10];
        input  (a);
        output(a);
}

void input(int *a)
{
        int i;
        for( i = 0 ; i < 10 ; i ++){
                scanf("%d", (a+i));
}

void output(int a[10])
{
        int i;
        for( i = 0 ; i < 10 ; i ++){
                printf("%d", a[i] );
}
```

```
1
2
3
4
5
6
7
8
9
10
12345678910
```

Lecture – 15

Shorting: Sometimes we need to short out a list of elements by ascending order or descending order. For this purpose we can apply various shorting algorithms.

1. Insertion sort
2. Bubble sort
3. Merge sort
4. Quick sort
5. Radix sort
6. Heap sort
7. Selection sort

Insertion sort:

```
void main()
{
        int a[50], i, j, n, t;
        scanf("%d",&n);

        for( i = 0; i < n; i ++)
                scanf("%d", & a[i]);

        for( j = 1; j < n; j ++){
                i = j-1;
                t = a[ j ];
                while( (i >= 0)&&(a[i] > t)){
                                a[i+1] = a[i];
                                i -- ;
                }
                a[i+1] = t ;
        }

        for( i = 0; i < n; i ++)
                printf("%d", a[i]);
}
```

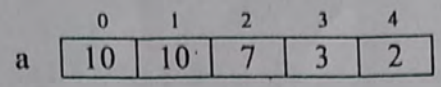
→ Shorting logic

Example: Suppose that we have a list of 5 elements. We will use the insertion sort algorithm to sort the given list in ascending order.

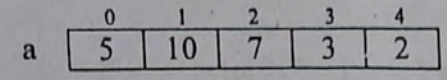
	0	1	2	3	4
a	10	5	7	3	2

Pass one: j = 1, i = 0, t = a[i] = 5

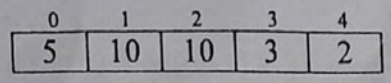
(0 >= 0) && (a[0] > 5) ⟶ TRUE



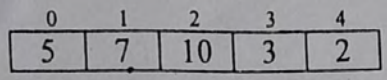
i = -1, t = 5
(-1 >= 0) && (a[-1] > 5) ⟶ FALSE



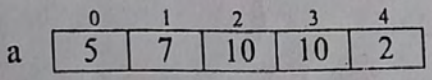
Pass two: j = 2, i = 1, t = a[2] = 7
(1 >= 0) && (a[1] > 7) ⟶ TRUE



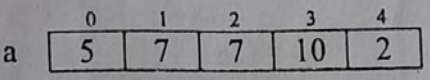
i = 0, t = 7
(0 >= 0) && (a[0] > 7) ⟶ FALSE



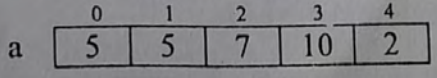
Pass three: j = 3, i = 2, t = a[3] = 3
(2 >= 0) && (a[2] > 3) ⟶ TRUE



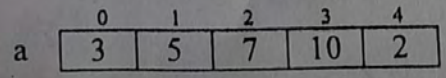
i = 1, t = 3
(1 >= 0) && (a[1] > 3) ⟶ TRUE



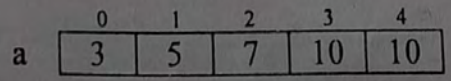
i = 0, t = 3
(0 >= 0) && (a[0] > 3) ⟶ TRUE



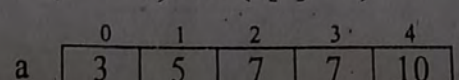
i = -1, t = 3
(-1 >= 0) && (a[-1] > 3) ⟶ FALSE



Pass four: j = 4, i = 3, t = a[4] = 2
(3 >= 0) && (a[3] > 2) ⟶ TRUE



i = 2, t = 2
(2 >= 0) && (a[2] > 2) ⟶ TRUE



i = 1, t = 2
(1 >= 0) && (a[1] > 2) ⟶ TRUE

	0	1	2	3	4
a	3	5	5	7	10

i = 0, t = 2
(0 >= 0) && (a[0] > 2) ⟶ TRUE

	0	1	2	3	4
a	3	3	5	7	10

i = -1, t = 2
(-1 >= 0) && (a[-1] > 2) ⟶ FALSE

	0	1	2	3	4
a	2	3	5	7	10

Bubble sort:

```
void main()
{
        int a[50], i, j, n, temp;
        scanf("%d",&n);

        for( i = 0; i < n; i ++)
                scanf("%d", & a[i]);

        for( i = 0; i < n-1; i ++)
                for( j = i+1; j < n; j ++)
                        if( a[i] > a[j]){
                                temp = a[ i ];
                                a[i] = a[ j ];
                                a[ j ] = temp;
                        }

        for( i = 0; i < n; i ++)
                printf("%d", a[i]);
}
```

→ Shorting logic

Example: Suppose that we have a list of 5 elements. We will use the bubble sort algorithm to sort the given list in ascending order.

	0	1	2	3	4
a	10	5	7	3	2

Pass one: i = 0, j = 1
(a[0] > a[1]) ⟶ TRUE

	0	1	2	3	4
a	5	10	7	3	2

j = 2
(a[0] > a[2]) ⟶ FALSE

	0	1	2	3	4
a	5	10	7	3	2

j = 3
(a[0] > a[3]) ⟶ TRUE

	0	1	2	3	4
a	3	10	7	5	2

j = 4
(a[0] > a[4]) ⟶ TRUE

	0	1	2	3	4
a	2	10	7	5	3

Pass two: i = 1, j = 2
(a[1] > a[2]) ⟶ TRUE

	0	1	2	3	4
a	2	7	10	5	3

j = 3
(a[1] > a[3]) ⟶ TRUE

	0	1	2	3	4
a	2	5	10	7	3

j = 4
(a[1] > a[4]) ⟶ TRUE

	0	1	2	3	4
a	2	3	10	7	5

Pass three: i = 2, j = 3
(a[2] > a[3]) ⟶ TRUE

	0	1	2	3	4
a	2	3	7	10	5

j = 4
(a[2] > a[4]) ⟶ TRUE

	0	1	2	3	4
a	2	3	5	10	7

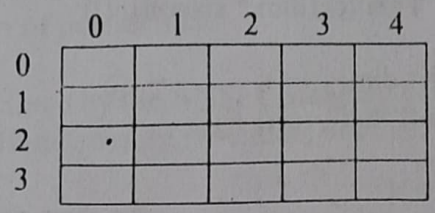
Pass four: i = 3, j = 4
(a[3] > a[4]) ⟶ TRUE

	0	1	2	3	4
a	2	3	5	7	10

CSE UNIQUE – I

Lecture – 16

Two dimensional arrays: Say, int a[4][5];



Example 1:

```
void main(void)
{
        int a[3][4], i, j ;
        for( i = 0; i < 3 ; i ++)
                for( j = 0 ; j < 4 ; j ++)
                        scanf("%d", &a[i][j]
                                );
        for( i = 0; i < 3 ; i ++){
                for( j = 0 ; j < 4 ; j ++)
                        printf("%d", a[i][j] );
                printf("\n");
        }
}
```

Example 2:

```
void main(void)
{
        int i, j, row, col, *p;
        scanf("%d%d",&row,&col);
        p = (int *) malloc ( row * col * sizeof(int));
        for( i = 0; i < row ; i ++)
                for( j = 0 ; j < col ; j ++)
                        scanf("%d", (p + i * col + j ) );
        for( i = 0; i < row ; i ++){
                for( j = 0 ; j < col ; j ++)
                        printf("%4d", *(p + i * col + j ) );
                printf("\n");
        }
}
```

Example 3:

```
void main(void)
{
    int *a[3], dim, i, j, b[3];
    for( i = 0; i < 3; i ++){
        scanf("%d",&dim);
        a[i] = (int *) malloc(dim * sizeof(int));
        b[i] = dim;
        for( j = 0; j < dim; j ++)
            scanf("%d", (a[i] + j));
    }
    for( i = 0; i < 3; i ++){
        for( j = 0; j < b[i]; j ++)
            printf("%4d",*(a[i] + j));
        printf("\n");
    }
}
```

Three dimentional array: Say, int a[3][4][5];

Example 4:

```
void main(void)
{
    char ch[2][3][10], i, j, k ;
    for( i = 0; i < 2 ; i ++)
        for( j = 0 ; j < 3 ; j ++)
            for( k = 0; k < 10; k ++)
                scanf("%c", &ch[i][j][k] );
    for( i = 0; i < 2 ; i ++){
        for( j = 0 ; j < 3 ; j ++){
            for( k = 0; k < 10; k ++)
                printf("%c", ch[i][j][k] );
            printf("\n");
        }
        printf("\n");
    }
}
```

Array initialization:

```
int a[2] = {1, 2};
char b[4] = {'a','b','c','\0'};
char a[] = "abc"; char
Char *p = "hellow"; char
Char c[10] = "hellow";
int x[2][2] = {{1, 2}, {3}};
```

Lecture – 17

String: A **string** is defined as a **null** terminated sequence of characters. A string constant is also terminated by the compiler assigns **null** automaticaily.

Header file used for string operation → **string.h**

Example 1:

```
void main(void)
{
        char n[100];
        gets(n);
        puts(n);
}
```

```
Hellow all
Hellow all
```

Example 2:

```
void main(void)
{
        char *ch;
        ch = (char *) malloc (100);
        gets(ch);
        printf("%d",strlen(ch));
}
```

```
Hellow all
10
```

Example 3:

```
void main(void)
{
        char src[]="Hellow",dest[];
        strcpy(dest,src);     //dest = src
        puts(dest);
}
```

```
Hellow
```

Example 4:

```
void main(void)
{
        char dest[]="Hellow",src[]="World";
        strcat(dest,src);     // dest = dest + src
        printf("%s",dest);
}
```

```
Hellow World
```

gets()
puts()
strlen()
strcpy()
strcat()
strcmp
strrev

$(0 >= 0)\ \&\&\ (a[0] > 5) \longrightarrow TRUE$

Example 5:

```
void main(void)
{
        char str1[], str2[];
        int val;
        gets(str1);
        gets(str2);
        val = strcmp(str1,str2);
        if( val = = 0 )
                printf("String1 and String2 are same");
        elseif( val > 0 )
                printf("String1 is greater than String2");
        elseif( val < 0 )
                printf("String1 is smaller than String2");
}
```

```
bcd
abc
String1 is greater than String2
```

Example 6:

```
void main(void)
{
        char str[20],rev[];
        gets(str);
        rev = strrev(str);
        puts(rev);
}
```

```
Ahsanullah University
ytisrevinU hallunashA
```

Example 7:

```
void main(void)
{
        char *str = "Hellow", str1[10];
        strncpy(str1, str, 4);
        str1[4] = '\0';
        puts(str1);
}
```

```
Hell
```

Lecture – 18

Preprocessor directory: The source code for C program can include various instructions to the compiler. A preprocessor directive performs macro substitution conditional compilation and inclusion of named files.

The C preprocessor as defined by the ANSI C standard contains the following directives : #if, #elif, #else, #ifdef, #ifndef, #endif, #include, #define, #undef, #line, #error, #pragma.

All the preprocessors begin with a #.

#define: The #define directive is used to define an identifier and a character sequence that is encountered in the source file. The identifier is called a macro name, and the replacement process is called macro substitution. The general form of the directive is:

#define macroname replacement_text

Example 1:

```
#define MAX 100
#define one 1
#define two one+one
#define three two+one
#define str "hellow"
```

The #define has another powerful feature. The macro name can have arguments.

Example 2:

```
#define  MIN(a,b)  ((a)<(b))?a : b

void main(void)
{
        int x = 10, y = 20 ;
        printf("Minimum %d\n", MIN(x,y));
}
```

Minimum 10

In compile time, **printf("Maximum %d\n", MIN(x,y));** will be replaced by **printf("Minimum %d\n", ((x)<(y))?x : y);**

Example 3:

```
#define  EVEN(a)  a%2 = = 0 ? 1 : 0

void main(void)
{
        if(EVEN(9+1))
                printf("This is even\n");
        else
                printf("This is odd\n");
}
```

```
This is odd
```

Conditional compilation: Syntax of conditional compilation is given below:

```
#if expression
        statement
#endif
```

Example 4:

```
#define MAX 12
#define SERIAL 5

void main(void)
{
        int flag = 0 ;
        #if MAX>10
                flag = 1;
                #if SERIAL
                        int port = 198;
                #else
                        int port = 200;
                #endif
        #else
                char a[100]="hellow";
        #endif
        if(flag)
                printf("%d",port);
        else
                printf("%s",a);
}
```

```
198
```

You need to often manipulate strings according to the need of a problem. Most, if not all, of the time string manipulation can be done manually but, this makes programming complex and large. To solve this, C supports a large number of string handling functions in the standard library **"string.h"**.

Few commonly used string handling functions are discussed below:

Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

Strings handling functions are defined under "string.h" header file.

#include <string.h>

Note: You have to include the code below to run string handling functions.

gets () and puts()

Functions gets() and puts() are two string functions to take string input from the user

```
1.  #include<stdio.h>
2.
3.  int main()
4.  {
5.    char name[30];
6.    printf("Enter name: ");
7.    gets(name);    //Function to read string from user.
8.    printf("Name: ");
9.    puts(name);    //Function to display string.
10.   return 0;
11. }
```

Note: Though, gets() and puts() function handle strings, both these functions are defined in "stdio.h" header file.

Lecture – 19

Scope rules of identifiers: The scope of an identifier is the part of the program within which the identifier can be used.

Every variable in C has an attribute type and a storage class.

Storage class: There are 4 storage classes in C.

- Automatic (auto)
- Register (register)
- Static (static)
- External (extern)

Attributes type: There are 2 attribute types in C.

- Constant (const)
- Volatile (volatile)

Variables of type **constant** may not be changed during execution by your program.

Volatile is used to tell the compiler that the variable can be changed in ways not explicitly specified by the program.

Example:

```
volatile x = 2;
const y = 3;
const volatile z = 4;
```

Features of **automatic** storage class are given below:

- Storage: Memory.
- Default initial value: Garbage value.
- Scope: Local to the block in which the variable is defined.
- Life: Till the control remains within the block in which the variable is defined.

Example 1:

```
void main(void)
{
        auto int x, y;
        printf("%d\n%d",x,y);
}
```

Example 5:

```
#define TED 10

void main(void)
{
        #ifdef TED
                printf("TED is defined\n");
        #else
                printf("TED is not defined\n");
        #endif
}
```

TED is defined

#pragma: #pragma is another preprocessor directive through which we can change compiler option. Actually it's an instruction to the compiler to perform some particular action at compile time.

Example 6:

```
void start(void);
#pragma startup start 65

void main(void)
{
        printf("In main\n");
}

void start(void)
{
        printf("In start\n");
}
```

In start
In main

Example 2:

```
void main(void)
{
        auto int i = 1 ;
        {
                auto int i = 2;
                {
                        auto int i = 3 ;
                        printf("%d",i);
                }
                printf("%d",i);
        }
        printf("%d",i);
}
```

321

Example: 3

```
void main(void)
{
        printf("hellow\n");
        {
                int k=5;
                printf("%d",k);
        }
        printf("%d",k);
}
```

This program will produce an error because **k** is unknown outside the block { }.

Features of **register** storage class are given below:

- Storage: CPU register.
- Default initial value: Garbage value.
- Scope: Local to the block in which the variable is defined.
- Life: Till the control remains within the block in which the variable is defined.

Example 4:

```
void main(void)
{
        register int i;
        for(i = 0; i <= 5; i ++)
                printf("%d",i);
}
```

012345

Features of **static** storage class are given below:

- Storage: Memory.
- Default initial value: Zero.
- Scope: Local to the block in which the variable is defined.
- Life: Value of the variable persists between different function calls.

Example 5:

```
void call(void);
void main(void)
{
        call();
        call();
        call();
}

void call(void)
{
        auto int k = 1;
        printf("%d\n",k);
        k ++;
}
```

```
1
1
1
```

Example 6:

```
void call(void);
void main(void)
{
        call();
        call();
        call();
}

void call(void)
{
        static int k = 1;
        printf("%d\n",k);
        k ++;
}
```

```
1
2
3
```

Features of **external** storage class are given below:

- Storage: Memory.
- Default initial value: Zero.
- Scope: Global.
- Life: As long as the program's execution doesn't come to an end.

Example 6:

```
int f(void);
int a = 1, b = 2, c = 3;
void main(void)
{
        printf("%d\n"f());
        printf("%d%d%d",a,b,c);
}
```

```
10
123
```



```
int f(void)
{
        int b = 4, c = 5;
        return a+b+c;
}
```

In the following example we will build a project to understand **external** storage class. Under this project we will create four files named **A.C**, **B.C**, **C.C** and **D.C**

A.C

```
int a,b;
void main()
{
        input();
        display();
}
```

```
Please enter two numbers
3
4
The sum is 7
```

B.C

```
#include<stdio.h>
extern int a,b;
void input()
{
        printf("Please enter two numbers\n");
        scanf("%d%d", &a, &b);
}
```

C.C

```
#include<stdio.h>
extern int a,b;
void display()
{
        printf("The sum is %d", add());
}
```

D.C

```
extern int a,b;
void add()
{
        return a+b;
}
```

Lecture – 20

Structures: A structure is a collection of one or more variables, possibly different types, grouped together under a single name for convenient handling. Structures help to organize complicated data particularly in large programs because they permit a group of related variables to be treated as a unit instead of as separate entities.

General format of structure:

```
struct tag_name{
        type element 1;
        type element 2;
                .
                .
                .
        type element n;
} variables;
```

The keyword **struct** introduces a structure declaration, which is a list of declaration enclosed in the braces. An optional name called a structure tag name may follow the word **struct**. The variables named in the structure are called members. A structure member or tag and an ordinary variable can have the same name without conflict, since they can be always distinguished by context. A structure declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic types. That is

```
struct { . . . }a, b, c, … , n;
```

Example 1:

```
void main()
{
        struct book
        {
                char name[20];
                float price;
                int pages;
        }b = {"Teach yourself C", 45.35, 640};
        printf("%s\t%f\t%d",b .name, b .price, b .pages);
}
```

One structure can be nested with another structure.

Example 2:

```
void main()
{
        struct address
        {
                char phone[15];
                char city[25];
                int pin;
        };
        struct emp
        {
                char name[25];
                struct address a;
        };
        struct emp e = {"Karim", "0171530476", "Dhaka",10};
        printf(" name = %s\n phone = %s", e .name, e .a .phone);
        printf("\n city = %s \n pin = %d", e .a .city, e .a .pin);
}
```

typedef: Syntex of using **typedef** is given below:

```
typedef  data_type  variable;
```

Example 3:

```
void main()
{
        typedef int x;
        x y;
        y = 5;
        printf("%d",y);
}
```

Example 4:

```
void main()
{
        struct stu...
        {
                char name[100];
                int roll;
                char address[100];
                float gpa;
        }
        typedef struct student std;
        std std1,std2 = {"Rahim",25, "South Kamalapur, Dhaka", 3.78};
        std1 = std2;
        printf("%s\t%d\t%s\t%f ", std1 .name, std1 .roll, std1 .address, std1 .gpa);
}
```

Example 5:

```
void main()
{
        int i;
        struct point
        {
                int x;
                int y;
        }p[5];
        for(i = 0; i < 5; i ++)
                scanf("%d%d", & p[i] . x, & p[i] . y);
        for(i = 0; i < 5; i ++)
                printf("%d\t%d\n", p[i] . x, p[i] . y);
}
```

Example 6:

```
void main()
{
        struct point * p;
        struct point
        {
                int x;
                int y;
        };
        p = (struct point *) malloc (sizeof(struct point));
        p → x = 5 ;
        p → y = 7;
        printf("%d\t%d\n", p → x, p → y);
}
```

JANUARY 20, 2020 / #C PROGRAMMING

Ternary Operator in C Explained

Programmers use the **ternary operator** for decision making in place of longer **if** and **else** conditional statements.

The ternary operator take three arguments:

1. The first is a comparison argument
2. The second is the result upon a true comparison
3. The third is the result upon a false comparison

It helps to think of the ternary operator as a shorthand way or writing an if-else statement. Here's a simple decision-making example using **if** and **else**:

```
int a = 10, b = 20, c;

if (a < b) {
    c = a;
}
else {
    c = b;
```

This example takes more than 10 lines, but that isn't necessary. You can write the above program in just 3 lines of code using a ternary operator.

Syntax

```
condition ? value_if_true : value_if_false
```

The statement evaluates to `value_if_true` if `condition` is met, and `value_if_false` otherwise.

Here's the above example rewritten to use the ternary operator:

```
int a = 10, b = 20, c;

c = (a < b) ? a : b;

printf("%d", c);
```

Output of the example above should be:

```
10
```

`c` is set equal to `a`, because the condition `a < b` was true.

Remember that the arguments `value_if_true` and `value_if_false`

Ternary operators can be nested just like if-else statements. Consider the following code:

```
int a = 1, b = 2, ans;
if (a == 1) {
    if (b == 2) {
        ans = 3;
    } else {
        ans = 5;
    }
} else {
    ans = 0;
}
printf ("%d\n", ans);
```

Here's the code above rewritten using a nested ternary operator:

```
int a = 1, b = 2, ans;
ans = (a == 1 ? (b == 2 ? 3 : 5) : 0);
printf ("%d\n", ans);
```

The output of both sets of code above should be:

```
3
```