Module 5: Structures, Pointers and Preprocessors

Introduction

Sometimes, it is necessary for the programmers to store and process large volume of dissimilar type of data elements in computer's memory and is possible by using **user defined data type** named **structure**.

Structure can be defined as a collection of data elements of dissimilar type of data elements. It is user defined data type.

Array	Structure
A collection of data elements of **similar type**	A collection of data elements of **dis-similar type**

What is structure?

A structure can be defined as a collection of data elements of dis-similar type of data elements stored in computer's memory. It is user defined data type.

Defining Structure

The structure must be defined in the *declaration or global part* of the C program before defining structure variables.

Syntax:

```
struct <tag_name>
{
type1  member1;
type2  member2;
type3  member3;
…
type-n  member-n;
};
```

Where,
struct → is a keyword
Tag_name → valid identifier or name of structure
type1, type2, type3,…, type-n → built-in data types like int, float, char, etc.
member1, member2, member3,.. → are fields or components of a structure

Example1. Structure to store and process employee information.

```
struct employee                 /* definition of structure*/
{
int eno;
char emp_name[25];
long int emp_salary;
};
struct employee e;
struct employee e1,e2,e3;
struct employee e[100];
```

Example2. Structure to store and process book information.

```
struct book                     /* structure declaration*/
{
int book_no;
char book_name[25];
int book_price;
};
struct book b;
struct book b1,b2,b3;
struct book b[100];
```

Declaration of Structure Variables

The structure variables can be defined in the *declaration or global part* of the C program before the structure variables are used in the main( ). The syntax is as follow.

Syntax:

```
struct <tag_name>   <list of variables separated by comma>;
```

Where,
struct → is a keyword
tag_name → valid identifier or name of structure
variables_list → List of variables

Example1. Structure to store and process student information.

```
struct student          /* definition of structure*/
{
int rno;
char name[25];
int marks;
char grade;
};
struct student s;                   /* To store info. of 1 student*/
struct student s1,s2,s3;            /* To store info. of 3 students*/
struct student s[50];               /* To store info. of 50 students*/
```

Example2. Structure to store and process employee information.

```
struct employee                 /* definition of structure*/
{
int eno;
char emp_name[25];
long int emp_salary;
};

struct employee e;              /* To store info. of 1 employee*/
struct employee e1,e2,e3;       /* To store info. of 3 employee*/
struct employee e[50];          /* To store info. of 50 employee*/
```

Example3. Structure to store and process product information.

```
struct product          /* definition of structure*/
{
int pno;
char pname[25];
float price, total;
int qty;
};

struct product p;               /* p is variable to store information of one product*/
struct product p1,p2,p3;        /* 3 variables of type struct product */
struct product p[100];          /* array of structure to store information of 100 products*/
```

Example 4. Structure to store and process book information.

```
struct book             /* definition of structure*/
{
int bookno;
char bookname[25];
char author[25];
float price;
};
struct book b;                  /* b is variable to store information of one book */
struct book b1,b2,b3;           /* 3 variables of type struct book */
struct book b[100];             /* array of structure to store information of 100 book*/
```

Three Ways to define structure variables

[1]
```
struct student
{
int rno;
char name[25];
int marks;
char grade;
};
struct student s1,s2,s3;                /* s1, s2, s3 are variables */
```

[2]
```
struct student
{
int rno;
char name[25];
int marks;
char grade;
}s1,s2,s3;                              /* s1, s2, s3 are variables */
```

[3]
```
struct
{
int rno;
char name[25];
int marks;
char grade;
}s1,s2,s3;                              /* s1, s2, s3 are variables */
```

[3]
```
typedef struct  student
{
int rno;
char name[25];
int marks;
char grade;
}stud;                                  /* here, stud is alternative name for struct student */

stud s1,s2,s3;                          /* s1, s2, s3 are variables */
```

Initialization of Structure Variables

The structure variables can be assigned with values at compile or rum time. The syntax is as follow.

Syntax:

struct tag_name variable={list of values separated by comma};

At Compile Time:

```
struct student
{
int rno;
char name[25];
int marks;
char grade;
};
struct student s={101,"John",600,'A'};
```

At Run Time:

```
struct student
{
int rno;
char name[25];
int marks;
};
struct student s;

printf("Enter student roll number, name and marks:");
scanf("%d %s %d", &s.rno, &s.name, &s.marks);
```

```
O/p:
Enter student roll number, name and marks:
101 Thomas 600
```

	rno	name	marks
s→	101	Thomas	600

Programming Examples on Structures

```
/* C program to read and display student information by using structure */
#include<stdio.h>
int main()
{
        struct student
        {
        int rno;
        char name[25];
        float avg;
        };
struct student s;

printf("Enter student roll no:");
scanf("%d",&s.rno);
printf("Enter student name:");
fflush(stdin);
scanf("%s",&s.name);
printf("Enter student average marks:");
scanf("%f",&s.avg);
printf("\n STUDENT DETAILS");
printf("\n Roll no=%d",s.rno);
printf("\n Name=%s",s.name);
printf("\n Average marks=%f",s.avg);
return (0);
}
```

o/p:
Enter student roll no: 101
Enter student name: Thomas
Enter student average marks: 69
STUDENT DETAILS
Roll no= 101
Name= Thomas
Average marks= 69

Write C program to read and display employee information (emp_no,name, designation & salary) by using structure variable.

```
/* C program to read and display employee information by using structure */
#include<stdio.h>
int main()
{
        struct employee
        {
        int no;
        char name[25];
        int salary;
        };
struct employee e;

printf("Enter employee no:");
scanf("%d",&e.no);
printf("Enter employee name:");
fflush(stdin);
scanf("%s",&e.name);
printf("Enter employee salary:");
scanf("%f",&e.salary);
printf("\n EMPLOYEE DETAILS");
printf("\n Employee No=%d",e.no);
printf("\n Employee Name=%s",e.name);
printf("\n Employee Salary=%d",e.salary);
return (0);
}
```

o/p:
Enter employee no: 207
Enter employee name: Bob
Enter employee salary: 30000
EMPLOYEE DETAILS
Employee No=207
Employee Name=Bob
Employee Salary=30000

/* C program to read and display student name,USN,subject and IA marks using structure*/

```
#include<stdio.h>
struct student
{
char name[25];
char usn[11];
char sub[50];
int ia;
};
int main()
{
struct student s;

printf("\nEnter Student Name:");
scanf("%s",&s.name);
printf("\nEnter Student USN:");
scanf("%s",&s.usn);
printf("\nEnter Subject Name:");
scanf("%s",&s.sub);
fflush(stdin);
printf("\nEnter student IA Marks:");
scanf("%d",&s.ia);
printf("\n STUDENT DETAILS");
printf("\n Name: %s",s.name);
printf("\n USN: %s",s.usn);
printf("\n Subject Name: %s",s.sub);
printf("\n IA Marks: %d",s.ia);
return (0);
}
```

```
o/p:
Enter student name: Jasmin
Enter student USN: 2BL15CS001
Enter subject name: Programming in C
Enter student IA marks: 20
STUDENT DETAILS
Name: Jasmin
USN: 2BL15CS001
Subject Name: Programming in C
IA Marks:20
```

Accessing Structure Members:

The programmer makes the use of dot operator(.) to access the individual members of structure variable. The syntax is as follow.

Syntax:

structure_variable . member_name;

Example:

```
struct student
{
int rno;
char name[25];
int marks;
};
struct student s={101,"John",600};
printf("\n Student Roll Number =%d",s.rno);
printf("\n Student Name =%s",s.name);
printf("\n Student Marks =%d",s.marks);
```

Nested Structure:

The C language permits the programmers to nest structures within one another i.e. structure can contain another structure member.

Example:

```
struct date
{
int dd, mm, yy;
};

struct student
{
char name[25];
struct  date dob,doa;          /* nesting of structure member to store date of birth and date of admission*/
}s;

strcpy(s.name,"John");

s.dob.dd=1;
s.dob.mm=6;
s.dob.yy=1960;

s.doa.dd=1;
s.doa.mm=8;
s.doa.yy=1978;
```

Array of Structures:

The programmers can define arrays of structures variables to store and process large volume of data i.e. a group of related data elements of dissimilar type.

Syntax:

```
struct tag_name array_name[size];
```

Example:
To store student information(roll no, name & average marks) of 'n' students.

```
struct student
{
int rno;
char name[25];
int marks;
};
struct student s[25];    /* Array of size 25 */
```

Array in Structures:

The programmers can use arrays as members of structure to store and process large volume of data.

Example:

Write a c program read and display student name, USN and 5 subjects IA marks using structure.

```
/* Example for arrays in structure */
#include<stdio.h>
struct student
{
char name[25];
char usn[11];
int ia[5];
};
void main()
{
struct student s[100];
int i;

printf("\nEnter Student Name:");
scanf("%s",&s.name);
printf("\nEnter Student USN:");
scanf("%s",&s.usn);
printf("\nEnter 5 subjects IA Marks:");
for(i=0;i<5;i++)
{
scanf("%d",&s.ia[i]);
}
printf("\n STUDENT DETAILS");
printf("\n Name: %s",s.name);
printf("\n USN: %s",s.usn);
printf("\n IA Marks:");
for(i=0;i<5;i++)
{
printf("\t %d",s.ia[i]);
}
getch();
}
```

Structures and Functions

The programmers can pass the structure variables or structure members to user defined functions to do specific task. The contents of structure variable can be passed as parameter user defined function in the following three ways.

Way1 to pass parameters to structure: The individual members of a structure can be passed as parameter to a function to do specific task.

e.g. result(s.avg);

Here, *result* is user defined function that receives s.avg as parameter to display student result as per average marks scored.

Way2 to pass parameters: The whole structure variable can be passed as an argument or parameter to user defined function to do specific task.

e.g. display(s);

Here, s is struct variable containing student's information that will be passed to display to display student's information.

Way3 to pass parameters: The address of structure variable can be passed as an argument or parameter to user defined function to do specific task. i.e. *call by reference method.*

e.g. display(&s);

Here, address of struct variable s is passed to user defined function to do specific task.

```
/* Programming Examples for Structure and Functions*/
#include<stdio.h>
struct student read(void);
void display(struct student s);
void result(float x);

struct student
{
int rno;
char name[25];
float avg;
};

void main()
{
struct student temp;

clrscr();
temp=read();
display(temp);
result(temp.avg);
getch();
}
```

```
O/p:
Enter student roll no, name and average marks:
101 Thomas 70
Roll No=101
Name=Thomas
Average Marks=70
Pass
```

```
struct student read(void)
{
struct student s;
printf("Enter student roll no, name and average marks");
scanf("%d %s %f",&s.rno,&s.name,&s.avg);
return(s);
}

void display(struct student s)
{
printf("\n Roll No=%d \n Name=%s \n Average Marks=%f",s.rno,s.name,s.avg);
}

void result(float x)
{
if (x>35)
printf("\n Pass");
else
printf("\n Fail");
}
```

typedef statement:

The typedef stands for type definition. It is one of the built-in statement available in C. It helps the programmer to give alternative name (possibly short and meaningful names) to the basic data types (int, float, char) or user defined data type like structure.

Syntax:

typedef <data_type> < alternative new name>;

where,

typedef → is a keyword

data_type→ basic or user defined data type

new_name → alternative name

Examples:

```
typedef int whole_nos;
Whole_nos n1,n2,n3;              /* variables of type int */

typedef int marks;
marks che,pcd,maths;             /* variables of type int */

typedef float real_nos;
real_nos r1,r2;                  /* variables of type float */

typedef char letter;
letter grade;                    /* variables of type char */

. typedef unsigned short int ushort;
ushort age;                      /* age is variable of type unsigned short int */
```

. typedef struct student stud;
stud s; /* s is variables of type struct student */

typedef struct employee emp;
emp e1,e2,e3; /* variables of type struct employee */

typedef struct student
{
int rno;
char name[25];
float avg;
}stud;
stud s1,s2,s3;

. typedef struct
{
int rno;
char name[25];
float avg;
}stud;
stud s1,s2,s3;

Viva-Voce

Can you answer these questions?

What is computer program?
What is stdio.h?
What is variable?
What is constant?
Can you name input and output functions available in C.
Can you name conditional statements available in C.
Can you name loop statements available in C.
What is the difference between while and do while?
What is the difference break and continue statements available n C?
What are the advantages of writing user defined functions?
What is an array?
What is structure?
What is the difference between array and structure?
What is pointer?
What is typedef?
What is dynamic memory allocation?
What are the benefits of learning computer programming languages like C?
Who developed the 'C' language?
What is structured programming language?
What is modular programming approach?
How do you develop software?
What is compiler?
Why do you compile the program?
What is the difference between syntax and logic errors?
What is meant by debugging?
What is an algorithm?
What is flowchart?
What is pseudocode?
What are C tokens?
What are data types?
What is computer?
What is your area of interest?
Why? Did you take admission for engineering?
Why? Do you write computer programs?

Good Luck!.....

Pointers

Introduction

A pointer is very powerful feature of C language. It helps the programmers to manipulate the data at low level i.e. by using addresses of computer's memories instead of identifiers. It helps the programmers to write concise and efficient computer programs.

Q. What is pointer? Discuss its declaration and initialization with syntax and examples.

"A pointer is also a variable that stores only the ADDRESS of other variable"

It helps the programmers to efficient computer programs in terms of time and space efficiency. i.e. the program that works with high speed and occupies less computer's memory.

Declaration of Pointer:

Pointers must be defined in the declaration part of the main program before they are used in executable part by using following syntax.

Syntax

data_type *pointer_name;

where,

data_type : It may be any data type like int, float, char, double, structure type etc.

pointer name: valid identifier

Examples:

```
int n=7;            /* normal or scalar variable */
int *ip;            /* integer pointer ip to store address of n variable*/
ip=&n;              /* Address of n variable will be stored in ip pointer*/

float ans=30.75;    /* float type of variable */
float *fp;          /* float pointer fp to store the address of ans variable*/
fp=&ans;            /* address of ans will be stored in fp pointer*/

char ch='a';        /* char type of variable */
char *cp;           /* character pointer cp to store the address of ch variable*/
cp=&ch;             /* address of ch will be stored in cp pointer*/
```

Initialization of Pointers:

The pointers can be assigned (initialized) with the address of other variables by using the following syntax.

Syntax:

pointer=&variable

Where,

Pointer : valid pointer

& : ampersand sign (Meaning: address of)

Variable : Well defined variable.

Examples:

```
int n=7;            /* Declaration of variable*/
int *ip;            /* Declaration of integer pointer ip */
ip=&n;              /* Initialization of pointer ip with the address of n variable */

float ans=30.75;
float *fp;
fp=&ans;

char ch='a';
char *cp;
cp=&ch;
```

Advantages of using pointers:

1. The use of pointers in programming results into concise and efficient programming.
2. Dynamic Memory Allocation (DMA) is possible by using pointers. i.e. the required number of memory blocks can be allocated neither more nor less. The process of allocating the computer's memory during program execution is called DMA.
3. Using pointers, programmers can construct Advanced Data Structures (ADTs) like linked list, queue, stack, tree, graph, set, etc. These ADTs are most commonly used during the design and development of today's real time applications.
4. The use of pointers with arrays, strings and user defined functions results into concise and efficient programming.

Disadvantages of pointers:

1. Uninitialized pointers may create errors in program.
2. Pointer bugs (errors) are difficult to remove.
3. Dynamically allocated memory must be freed explicitly.
4. Pointers updated with incorrect values may lead to memory corruption.

Declaration of Pointers

The pointers can be defined in the declaration part of the main( ) function by using dereference operator(*). The syntax is as follow.

Syntax:

data_type * ptr_variable;

where,

data_type → valid data types like int, float, char, struct type, etc.

asterisk (*) → dereference operator

ptr_ variable → valid identifier

1. Example to work with integer pointer

```
int n=7;          /* integer variable */
int *ip;/* integer pointer*/
ip=&n;/* address of 'n' will be stored in ip */
```

∴ ip=0x800

Without using pointers:

We can print address and contents of 'n' without using pointers as follows.

```
printf("Address of n = %d",&n);
printf("Contents of n=%d",n);
```

```
o/p:
 Address of n=0x800
 Contents of n= 7
```

Using pointers:

We can print the address and contents of 'n' using pointers as follows.

```
printf("Address of n = %d", ip);
printf("Contents of n=%d", *ip);      /* value
```

```
o/p:
 Address of n=0x800
 Contents of n= 7
```

2. Example to work with float pointer

```
float avg=88.88;              /* variable of type float */
float *fp;                    /* pointer of type float*/
fp=&avg;                      /* address of n will be stored in fp*/
printf("Address of variable avg = %d",fp);
printf("Contents of avg is %d = ",*fp);
```

Address Operator (&) and Dereference Operator(*)

To manipulate the data by using pointers, the C programmer makes the use of address operator (&) and dereference operator (*).

ampersand (&) : The ampersand sign is used as address operators. It helps the programmers to get the addresses of other variables.

asterisk (*) : The asterisk sign is used as deference operator. It helps the programmer to define pointer variables and also to get the value at addresses.

e.g. To print the address and contents of 'n' using pointers.

```
int n=7;
int *ip;
ip=&n;
printf("Address of n = %d",ip);
printf("Contents of n=%d",*ip);       /* value at address 0x800*/
```

Output :
Address of n=0x800
Contents of n= 7

e.g. To print the address and contents of 'n' using pointers.

```
float avg=88.88;              /* variable of type float */
float *fp;                    /* pointer of type float*/
fp=&avg;                      /* address of n will be stored in fp*/
printf("Address of variable avg = %d",fp);
printf("Contents of avg is %d = ",*fp);
```

Output:
Address of variable avg = OX2300
Contents of avg is 88.88

Pointers and Functions Arguments (Call by address)

The call by reference method makes the use of **pointers** to pass the addresses (references) of actual parameters to user defined function instead of values. In this method, changes made to formal parameters will affect the actual parameters. This method is having the capability to return multiple values back to called function, where as pass by value returns only one value.

```
/* Example for call by reference or function to swap two values */
#include<stdio.h>
void swap(int *x,int *y);
void main()
{
int x=7,y=3;    [3]x     [7]y
                8000    9000    …addresses

printf("\n Before swap or Before calling function \n");
printf("\n x=%d y=%d \n",x,y);
swap(&x,&y);
printf("\n After Swap or After calling function \n");
printf("\n x=%d y=%d \n",x,y);

}
void swap(int *x, int *y)       [8000]*x                [9000]*y
{
int temp;               temp=7
temp=*x;        temp=value at address 8000=7
*x=*y;          value at address 8000=value at address 9000
*y=temp;        value at address 9000=tmp=7
}
```

```
o/p:
Before swap
x=7    y=3
After swap
X=3     y=7
```

Pointers and Arrays

An array can be defined as a collection of logically related data elements of similar type. The programmers can make the use of pointers with arrays to write efficient programs. The programmers can increment / decrement the pointers to interact with the array elements. By default, array name will return the address of first memory allocated. (a is equals to &a[0])

```
/* Pointers and Arrays */
#include<stdio.h>
void main()
{
int a[5]={10,20,30,40,50};     a or a[0]
int *ip,i;
ip=&a[0];                  /* ip=a  i.e. Address 8000 will be stored*/

printf("The arrays elements:");
for(i=0;i<5;i++)
{
printf("\n%d",*(ip+i));              value at address 8000,8002,8004,..
}

}
```

Output:
The array elements:
10
20
30
40
50

Explanation:
8000+1=8002, 8002+1=8004, 8004+1=8006

i→	0	1	2	3	4
a→	10	20	30	40	50
Address	8000	8002	8004	8006	8008

Increment/decrement pointers:

The pointer variables can be incremented or decremented to interact with next or previous computers memory locations sequentially.

- Whenever, we increment/decrement the **integer pointer** that will be incremented / decremented by 2 because that occupies **2 bytes**.
- Whenever, we increment/decrement the **float pointer** that will be incremented / decremented by 4 because that occupies **4 bytes**.
- Whenever, we increment/decrement the **character pointer** that will be incremented / decremented by 1 because that occupies **1 byte**.

int n=5;	**float ans=4.5;**	**char ch='a';**
[5]n	**[4.5] ans**	**[a]ch**
8000	**9000**	**1000**
int *ip=&n;	**float *fp=&ans;**	**char *cp=&ch;**
2 bytes	**How many bytes…?2 bytes**	**cp?...2 bytes**

Pointers and Strings:

The programmers can use the pointers while working with string variables to write concise and efficient programs.

Examples:

```
/* Example for pointers and strings to display string on monitor by using pointer*/
#include<stdio.h>
void main()
{
char str[]="Good Morning";
char *cp;               /* character pointer*/
int i;
cp=&str[0];     8000    /* address of first array element will be stored in cp */

while (*cp!='\0')       /* value at first address not equals to NULL repeats */
{
printf("%c",*cp);       /* prints value at address i.e. G and so on..*/
cp++;                   /*pointer increments by 1 every time */
}

}
```

Output:

Good Morning

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Str	**G**	**o**	**o**	**d**		**M**	**o**	**r…..**	
Address	**8000**	**8001**	**8002**	**8003**	**8004**	**8005**	**8006**		

Arrays of Pointers: Programmers can define arrays of pointers to store and process addresses on 'n' number of variables while solving complex problems. It results into efficient program.

Example: To store addresses of three variables x, y and z, you can define one pointer array of the size 3.

```
int x=100, y=200, z=300;
int *ip[3];
ip[0]=&x, ip[1]=&y, ip[2]=&z;
```

```
[100]x      [200]y      [300]z      : variables and the values inside
9000        10000       15000       ← Addresses of varibles(assume)
```

Without using pointer (printing x,y,z)	Using Pointer (printing x,y,z)
printf("x=%d",x); output: x=100	Ip[0]=&x; printf("x=%d",*ip[0]); value at address 9000 output: x=100
printf("y=%d",y); output: x=200	Ip[1]=&y; printf("y=%d",*ip[1]); value at address 10000 output: y=200
printf("z=%d",z); output: z=300	Ip[2]=&z; printf("z=%d",*ip[2]); value at address 15000 output: z=300

Pointers to Pointers: While solving complex problems it is necessary for the programmer to store and process the addresses of pointers and is possible by defining pointer to pointer.

Examples:

```
int n=7;
int *ip;/* pointer to normal variable n*/
int **ipp;      /* it is the pointer to pointer to point ip */
ip=&n;
ipp=*ip;
```

Now, we display the contents of 'n' by using both pointer (ip) and pointer to pointer (ipp).

printf("\n n=%d",*ip); o/p: n=7

printf("\n n=%d",**ipp); o/p: n=7

Explanation:

[7]n	**Variable**
9000	**Address**
[9000]	***ip=&n**
10000	**Address**
[10000]	****ipp=&ip**
15000	**Address**

Memory Management (Allocation) in C

The programmer needs to allocate the computer's primary memory to store and process the data to solve a problem by using a computer. For this purpose, the programmer makes the use of following two techniques.

i. **Compile time (static) Memory Allocation**
ii. **Dynamic (run-time) Memory Allocation (vimp)**

Compile time memory allocation: As the name implies, the process of allocating the computer's primary memory well in advance before the execution of statements from executable part of main() function in the declaration part is called compile time memory allocation.

This technique has the following limitations.

- The allocated memory is limited in size that cannot be varied during program execution.
- An array size must be defined with fixed size. i. e. size cannot be changed during program execution.
- Programmers cannot create Abstract Data Types (ADTs) like linked lists, trees and graphs, etc.

All these limitations can be solved by using Dynamic Memory Allocation technique.

Static Memory Allocation	Dynamic Memory Allocation
We allocate the memory (we define variables or arrays, well in advance)	We allocate the memory during program execution.
Drawback of array: Fixed Size int a[100]; printf("Enter number of elements n"); scanf("%d",&n); if , n is 25, then 75 memories will be wasted. If, n is 200, then shortage of 100 memories. This problem can be solved by DMA.	Int *ip; printf("Enter number of elements n"); scanf("%d",&n);5 Ip=(int *) malloc(n*sizeof(int)) … We allocate n number of memories only, neither more nor less.

Dynamic (run-time) Memory Allocation (DMA):

Q. What is dynamic memory allocation? Explain in brief.

Ans. The process of allocating the required number of memory blocks during the program execution is called dynamic or run-time memory allocation. This technique makes the use of memory allocating functions like malloc(), calloc(), realloc() and free() with pointers.

Memory Allocation Functions: The alloc.h or stdlib.h file is having number of memory allocating functions. Some of the important are as follows.

1. malloc()
2. calloc()
3. realloc() and
4. free()

malloc() : It allocates the required number of memory blocks during the program execution and returns the address of first memory block. The other addresses of next allocated memory blocks can be obtained by addition operation with a pointer. In case of failure of memory allocation, the malloc() funcion returns NULL value.

Syntax:

pointer=(data_type *) malloc(byte size);

where,

pointer : Valid pointer
data type : Valid data type of which type memory could be allocated.
byte_size :Total number of bytes. i.e. n * sizeof(data type)

/* Programming example for dynamic memory allocation by malloc */

```
#include<stdio.h>
#include<alloc.h>
void main()
{
int n,i,*ip;
printf("Enter the value of n:  ");
scanf("%d",&n);
ip=(int *) malloc (n*sizeof(int));
printf("\n Enter %d values ",n);
        for(i=0;i<n;i++)
        {
        scanf("%d",(ip+i));
        }
printf("\n The values are ");
        for(i=0;i<n;i++)
        {
        printf("\t %d",*(ip+i));
        }
free(ip);
}
```

```
Output:
Enter the value of n: 5
Enter 5 values
10 20 30 40 50
The values are
10 20 30 40 50
```

```
[10]   [20]   [30]   [40]   [50]
8000   8002   8004   8006   8008
```

8000+0=8000, 8000+1=8002,8000+2=8004,….8008

value at address 8000, 8002, 8004, 8006,8008

calloc() : It allocates the required number of memory blocks during the program execution and returns the address of first memory block. The other addresses of next allocated memory blocks can be obtained by addition operation with a pointer. The calloc() function initializes the allocated memory blocks by zero, whereas, the malloc() stores garbage values. In case of failure of memory allocation, the calloc() funcion returns NULL value.

Syntax:

pointer=(data_type *) calloc (n, byte_size);

where,

pointer : Valid pointer

data type : Valid data type of which type memory could be allocated.

n : Number of memory blocks required

byte_size : Total number of bytes.

/* Programming example for dynamic memory allocation by calloc */

```
#include<stdio.h>
#include<alloc.h>
void main()
{
int n,i,*ip;
printf("Enter the value of n:  ");
scanf("%d",&n);
ip=(int *) calloc(n,2);
printf("\n Enter %d values ",n);
        for(i=0;i<n;i++)
        {
        scanf("%d",(ip+i));
        }
printf("\n The values are ");
        for(i=0;i<n;i++)
        {
        printf("\t %d",*(ip+i));
        }
free(ip);
}
```

Output:
Enter the value of n: 5
Enter 5 values
10 20 30 40 50
The values are
10 20 30 40 50

malloc(n*sizeof(int)) function allocates the memory and stores garbage values by default

Values:	Unknown value (garbage Value)	Unknown value (garbage Value)	Unknown value (garbage Value)	Unknown value (garbage Value)	Unknown value (garbage Value)
Address	8000	8002	8004	8006	8008

calloc(n,sizeof(int)) function allocates the memory and stores zero'es (0's) values by default

Values:	0	0	0	0	0
Address	8000	8002	8004	8006	8008

realloc(): This is another memory allocation function available in alloc.h. It helps the programmer to allocate new memory space for the previously allocated memory by malloc() or calloc().

Syntax:

pointer=realloc(pointer, new_size);

Where,

Pointer : Address of previously allocated memory
New_Size : New memory spaced needed in terms of bytes.

Example:

```
int *ip; /* integer pointer*/
int n;
printf("Enter the value of n:  ");
scanf("%d",&n);
ip=(int *) malloc(n*sizeof(int));
printf("\n %d memory blocks are allocated ",n);
…
…
printf("Enter the new value of n:  ");
scanf("%d",&n);
ip=realloc(ip, n*sizeof(int));
printf("\n %d memory blocks are newly allocated ",n);
```

Output:

```
Enter the value of n: 5
5 memory blocks are allocated
Enter the new value of n: 7
7 memory blocks are allocated
```

free(): It de allocates the dynamically allocated memory by malloc() , calloc() or realloc() functions. The programmer needs to de allocate the memory allocated by DMA (Dynamic Memory Allocation) technique explicitly.

Syntax

free (pointer);

The Preprocessors

What are preprocessors? Explain in brief.

The preprocessor is another unique feature of C language. The preprocessor can be defined as a small program that runs before the compilation of main( ) program. All the preprocessors begin with pound(#) symbol and do not end with semicolon.

The #include and #define are most commonly used preprocessor directives. These are used to include external files and to define symbolic constants.

Advantages of using preprocessors

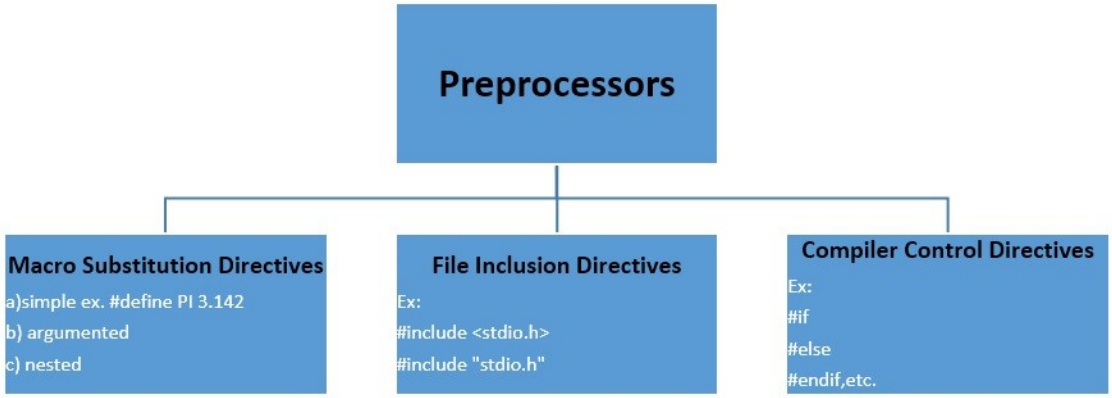
1. Written programs are easy to read
2. Written programs are easy to modify
3. Written programs are easy to portable

Types of preprocessors

1. Macro substitution directives

   (a) Simple (b) argumented (c) nested

   (b)
2. File inclusion directives
3. Compiler control directives



1. Macro substitution directives: These are used to replace an identifier (symbolic constant) with equivalent strings. These are used to define symbolic constants by using *#define* directive.

Syntax:

 # define <identifier> <equivalent string>

Examples:

define N 100

define MAX 50

define MIN 0

define INT 5.1452

Examples for macros with arguments

define SQUARE(x) ((x) * (x))
define CUBE(x) (x * x * x)

define MAX(a,b) (((a) > (b)) ? (a) : (b))

Examples for nested macros
define M 5
define N M + 1
Undefining macros:
A defined macro can be undefined by using #undef statement.
Syntax:
undef <identifier>
Example:
define N 500
undef N

2. File inclusion directives:

The file inclusion directives help the programmers to include external header files or C files in to C program by using **# include** directive. We are able to access functions and macro from the included external files.

Syntax:

 # include "filename" or # include <filename>

If, we use double quote to include external file, then the specified file will be searched from working directory.

If, we use angle brackets to include external file, then the specified file will be searched from standard directory like c:\tc\include.

Examples:

include <stdio.h>
include <conio.h>
include "p1.c"
include "p2.c"
include "D:\myself\p3.c"

3. Compiler control directives:

The compiler control directives help the programmers to define or undefined macros based on given condition. These can be used to write portable programs.(program written for one machine works on another machine)

Directives	**Explanation**
#ifdef	It will test, whether macro is defined
#endif	It is the end of if
#ifndef	It will test, whether macro is not defined
#if	the if … else sequence
#else	
#elif	the else … if sequence

Examples:

```
# include "define.h"
# ifndef TEST
# define TEST 1
# endif
```

```
# include "define.h"
# ifdef TEST
    # undef TEST
# endif
```

We can use compiler control directives to write portable program as follow

```
#if MACHINE = = HCL
        # define FILE “hcl.h”
# elif MACHINE = = WIPRO
        # define FILE “wipro.h”
# elif MACHINE = = DELL
        # define FILE “dell.h”
# endif

#include FILE
```

Tips to score more marks in VTU Examinations (Engineering)
Dear students,
You are clever and intelligent, do not lose your confidence. Yes, you can do well in exams.
Move towards examination hall with positive thoughts in such a way that, this time I do well.
Your's one positive thought helps you lot.
Be happier on examination day.
Be cool, try to feel weight less...be calm and cool in all situations...do not be over confident...
Move towards exam hall with all necessary things like Hall Ticket, 2 Dark Blank Ink Pens, Calculator.
Be independent; do not depend on others for asking pen, pencil, rubber, scale, etc.
Be well dressed i.e. formal.
Do prayer....before coming to exam; as well as before answering questions...in your peaceful soul....
Better; attempt that question first, in which you are perfect....
Very important is that, write question number correctly.
Underline important points in your answer.
Highlight the important word.
Give more examples.
Quality points are more important than quantity.
We need to have the habit of increasing quantity of quality; that comes by practice.
Wherever necessary, draw figure..
For every question; one page answer is compulsory.
For 10 marks minimum 2 to 3 pages (front and back)
Yours signature and supervisor signature on Answer booklet is compulsory...
Handwriting must be readable.....
Good luck....
All the best...
Chidanand S. Kusur
Asst. Prof., Dept. of CSE
BLDEA's CET, Vijayapur-03.
cs.kusur@gmail.com
9739762682
Dear Student,

You are hereby requested to visit the blog **cskusur.blogspot.in** and send your suggestions/feedback about class notes to e.mail id cs.kusur@gmail.com , youtube channel: Chidanand S Kusur (for C and Python Video Lectures…subscribe the channel)…Please free to ask doubts at any time…..send e mail or send whatsup message or phone call….All the best…Visit the blog: **cskusur.blogspot.in, subscribe my youtube channel Chidanand S Kusur.**

"Good Luck for Annual Examination"
Mr. C. S. Kusur Asst. Prof., Dept. of Computer Science and Engineering
B.L.D.E.A's Vachana Pitamaha Dr. P. G. Halakatti College of Engineering and Technology, Vijayapur-586 103, Karnataka, India.

…