



MODULE 5

MODULE 5

STRUCTURES

Structure is a user defined data type which allow to combine different data type.

Advantages

- ❖ Structures help to organize complex data in a more meaningful way.
- ❖ Being able to manipulate several variables as a single group makes your programs easier to manage.
- ❖ Business data processing uses the concepts of structures in almost every program. Example, student database and employee database management.

DEFINITION OF STRUCTURES

As variables are defined before they use in the program, structures are also defined and declared before they are used.

A structure definition forms a template that may be used to create structure objects. The variables that make up the structure are called members of the structure.

The structure definition associated with the structure name is referred as tagged structure. It does not create an instance of a structure and does not allocate any memory.

The general form or syntax of tagged structure definition is as follows,

```
struct tagname
{
    datatype    member1;
    datatype    member2;
    .....
    .....
    datatype    membern;
};
```

Where,

- ❖ **struct** is the keyword which tells the compiler that a structure is being defined.
- ❖ **tagname** is the name of the structure.
- ❖ member1, member2 ... are called members of the structure. The members are declared within curly braces.
- ❖ The closing brace must end with the semicolon.

Example: student details using tagged structure

```
struct student
{
    int    regno; char
    name[50]; float
    avgmarks;
};           // no memory is allocated for the structure Note
```

the following points with respect to above structure definition,

- ❖ **struct** is the keyword which tells the compiler structure is being defined.
- ❖ **student** is an identifier representing the structure name (tagname).
- ❖ name, regno and avgmarks are members of a structure and are themselves are not variables. They do not occupy any memory.

Memory is not reserved for the structure definition since no variables are associated with the structure definition. The members of the structure do not occupy any memory until they are associated with the structure variables.

The declaration of the structure variable takes of the form,

struct tagname var1, var2...;

where, struct is the keyword. tagname is the name of the structure. Structure variables are separated by comma, followed by semicolon.

We can declare structure variables anywhere in the program. For the

above example the variable declaration as follows,

```
struct student s1;           // memory is allocated for the variable
```

now a variable of type **struct student** (derived type) is created, the memory is allocated for the variable s1.

The following figure shows the memory organization for the above example.

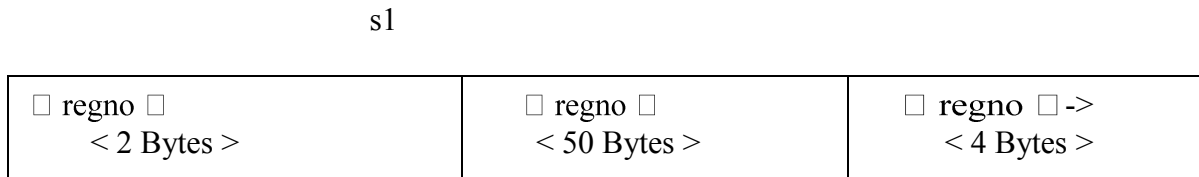


Fig. Memory map for structure variable

The number of bytes allocated for the structure variable is the sum of sizes of the individual members. In the above example the size of s1=56 bytes (2+50+4).

Note: Normally, structure definition appears at the beginning of the program file, before any variables or functions defined.

INITIALIZATION OF STRUCTURES

The rules for structure initialization are similar to the rules for array initialization. The initializes are enclosed in braces and separate by commas. They must match their corresponding types in the structure definition.

The syntax is shown below,

struct tag_name variable = {value1, value2,... valuen};

Structure initialization can be done in any of the following initialization.

Initialization along with Structure definition

Consider the structure definition for student with three fields regno, name, and avgmarks. The initialization of variable can be done as shown below,

```
struct student
{
    int    regno; char
    name[50]; float
    avgmarks;
};
```

```
struct student s1= {186, "Raj G Sami", 78.8};
```

ACCESSING STRUCTURES

We know that variables can be accessed and manipulated using expressions and operators. On the similar lines, the structure members can be accessed and manipulated. The members of a structure can be accessed by using dot(.) operator.

dot (.) operator

Structures use a dot (.) operator to refer its elements, also known as period operator.

Before dot, there must always be a structure variable. After the dot, there must always be a structure element.

The syntax to access the structure members as follows,

structurevariablename . structuremembername

For example, consider the example as shown below, struct

```
student
{
    int    regno; char
    name[50]; float
    avgmarks;
};
```

```
struct student s1= {186, "Raj G Sami", 78.8};
```

The members can be accessed using the variables as shown below, s1.regno -->

refers 186

s1.name --> refers the string "Raj G Sami"

s1.avgmarks--> refers 78.8

Array of structures

Array of structures is nothing but collection of structures. This is also called as structure array in C.

```
#include<stdio.h>
#include<string.h>

void main()
{
    struct student
    {
        int regno;
        char name[50];
        float avgmarks;
    };
    struct student s[2]; int
    i;
    clrscr();
    // 1st student's record s[0].regno=186;
    strcpy(s[0].name,"Raj G Sam");
    s[0].avgmarks = 78.8;

    // 2nd student's record s[1].regno=274;
    strcpy(s[1].name,"Varsha G Sam");
    s[1].avgmarks = 85.9;

    for(i=0; i<2; i++)
    {
        printf("    Records of STUDENT : %d \n", i+1);
        printf(" Regno: %d \n", s[i].regno);
        printf(" Name: %s \n", s[i].name);
        printf(" Average Marks: %.2f\n\n",s[i].avgmarks);
    }
    getch();
}
```

Passing struct to function

- A structure can be passed to any function from main function or from any sub function.
- Structure definition will be available within the function only.
- It won't be available to other functions unless it is passed to those functions by value or by address(reference).
- Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program.

Example program – passing structure to function in C by value:

In this program, the whole structure is passed to another function by value. It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function. This concept is very useful while writing very big programs in C.

```
#include    <stdio.h>
#include<string.h>

struct student
{
    int regno;
    char name[50];
    float avgmarks;
};

void gkstrtfun(struct student s);

void main()
{
    struct student s;
    clrscr();
    s.regno=186;
    strcpy(s.name,"Raj G Sam");
    s.avgmarks = 78.8;
    gkstrtfun(s);
    getch();
}
```

```
void gkstrtfun(struct student s)
{
    printf(" Regno: %d \n", s.regno);
    printf(" Name: %s \n", s.name);
    printf(" Average Marks: %.2f\n\n",s.avgmarks);
}
```


NESTED STRUCTURES

A structure which includes another structure is called nested structure i.e a structure can be used as a member of another structure.

The syntax for the nesting of the structure as follows, struct

```
tag_name1
{
    type1 member1;
    .....
    .....
};

struct tag_name2
{
    type1 member1;
    .....
    .....
    struct tag_name1 var;
    .....
};
```



The syntax for accessing members of a nested structure as follows,

outer_structure_variable.innerstructurevariable.membername

Example: Consider the student information regno, name, DOB and avgmarks. The DOB consists of day, month, and year. It can be defined as shown below,

```
struct data
{
    int day; int
    month; int
    year;
```



```
};

struct student
{
    int regno;
    char name [50];
    struct data dob;
    float avgmarks;
};

struct student s;
```

Note that the above structure consists of a member identified by `dob` whose type is `struct data`.

The members contained in the inner structure namely `day`, `month` and `year` can be referred to as

`s.dob.day`

`s.dob.month`

`s.dob.year`

An inner-most member in a structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most).

The memory organization of variable `s` is shown in below Figure.

s

regno	name	dob			avgmarks
2 bytes	50 bytes	day 2 bytes	month 2 bytes	year 2 bytes	4 bytes

Figure : Memory organization of nested structure

UNIONS

- ❖ A union is one of the derived data type.
- ❖ Union is a collection of variables referred under a single name.
- ❖ The syntax, declaration and use of union is similar to the structure but its functionality is different.
- ❖ The major distinction between structure and union is, in terms of storage.
- ❖ In structure each member has its own storage location.
- ❖ Whereas all the members of a union use the same location.
- ❖ Although a union may contain many members of different types, it can handle only one member at a time.

The general format or syntax of a union definition is as follows,

```
union tagname
{
    datatype1    member1;
    datatype2    member2;
    .....
    .....
};
```

Observe the following points while defining a union.

- ❖ **union** is the keyword which tells the compiler that a union is being defined.
- ❖ member1, member2, ... are called members(or fields) of the union.
- ❖ The members are declared within curly braces.
- ❖ The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.
- ❖ There should be semicolon at the end of closing braces.

A union variable can be declared same way as structure variable.

union tagname var1, var2...;

Differences between structures and unions:

Structures

1. The keyword struct is used to define a structure.
2. When a variable is associated with a structure, the compiler allocates the memory for each. The size of structure is equal to the sum of sizes of its members.
3. Each member within a structure is assigned unique storage area.
4. Altering the value of a member will not affect other members of the structure.
5. Individual members can be accessed at a time.
6. Several members of a structure can be initialized at once.

Unions

1. The keyword union is used to define a union.
2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest member.
3. Memory allocated is shared by individual members of union.
4. Altering the value any of the member will alter other member values.
5. Only one member can be accessed at a time.
6. Only the first member of a union can be initialized.

For example programs and solutions refer the following links in these links you have an option to see the solution as well as option goto editor in which you can type your program and run it and see the output online. You can type all the programs in the notes and also lab manual and execute it here.

<https://www.w3schools.in/c-tutorial/structures/>

Refer the following links for online lecture

https://www.youtube.com/watch?v=kDDd7AmXq1w&list=PLJ5C_6qdAvBFzL9su5J-FX8x80BMhkPy1&index=55

https://www.youtube.com/watch?v=PFebR3CbftE&list=PLJ5C_6qdAvBFzL9su5J-FX8x80BMhkPy1&index=56

https://www.youtube.com/watch?v=nC_egkQnbJE&list=PLJ5C_6qdAvBFzL9su5J-FX8x80BMhkPy1&index=57

POINTER

Pointer is a variable which contains the address of a variable. **or**

A variable which holds address of another variable is called a pointer variable. Example:

If x is a variable and address of x is stored in a variable p (i.e. $p = \&x$), then the variable p is called a pointer variable.

(Note: A pointer variable should contain only the addresses)

Syntax of declaring pointer variable: datatype

***identifier;**

Here datatype indicates any data type like int, char, float etc. It can be derived or user defined data type also.

Identifier is the name given to the pointer variable. It should be a valid user-defined variable name. The asterisk(*) in between type and identifier indicates that the identifier is a pointer variable.

Example: `int *p;`

`char *x;`

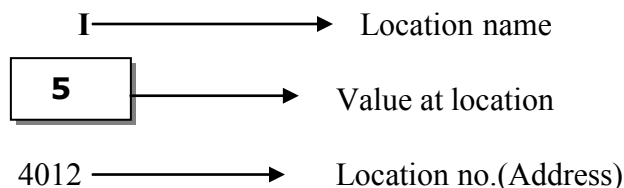
`float *z,*a,*b;`

Pointer operators or Operators used in Pointers:

Consider the declaration

`int I =5;`

We may represent the I's location in memory by the memory map :



Here the location number 4012 is not a number to be relied upon. The important point is I's address in memory is a number.

Address operator(&): It is used to get the address of the variable where it is stored in the memory.

a) Indirection operator or Dereferencing operator or Asterisk(*): It used to get the value stored in the particular address.

Accessing variables through pointers:

We use indirection operator(*) to access the values of the variables through pointers. Example:

```
int x=10, *p;  
p=&x;  
printf("%d",*p);
```

Here *p prints the value of the variable x. (i.e. 10).

NULL Pointer

- A NULL pointer is defined as a special pointer value that points to '\0'(nowhere) in the memory.
- In other words, NULL pointer does not point to any part of the memory.

For ex:

```
int *p=NULL;
```

C - Pointer arithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer –

```
ptr++
```

After the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array –

```
#include <stdio.h>
const int MAX = 3;

int main () {
    int var[] = {10, 100, 200};
    int i, *ptr;

    /* let us have array address in pointer */
    ptr = var;

    for ( i = 0; i < MAX; i++) {

        printf("Address of var[%d] = %x\n", i, ptr );
        printf("Value of var[%d] = %d\n", i, *ptr );

        /* move to the next location */
        ptr++;
    }

    return 0;
}
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below –

```
#include <stdio.h>

const int MAX = 3;

int main () {

    int var[] = {10, 100, 200};
    int i, *ptr;

    /* let us have array address in pointer */
    ptr = &var[MAX-1];

    for ( i = MAX; i > 0; i--) {

        printf("Address of var[%d] = %x\n", i-1, ptr );
        printf("Value of var[%d] = %d\n", i-1, *ptr );

        /* move to the previous location */
        ptr--;
    }

    return 0;
}
```

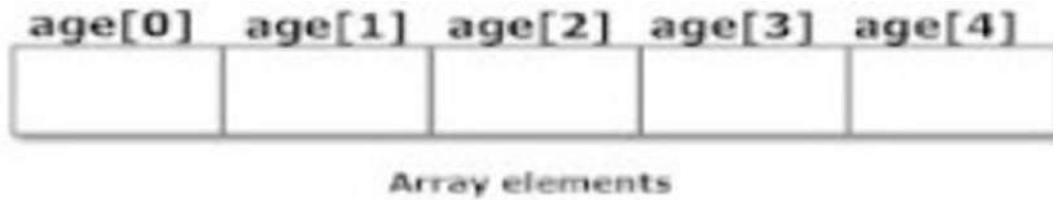
```
}
```

Pointers and Arrays

Consider an array:

```
int arr[4];
```

The above code can be pictorially represented as shown below:



- The name of the array always points to the first element of an array.
- Here, address of first element of an array is `&arr[0]`.
- Also, `arr` represents the address of the pointer where it is pointing. Hence, `&arr[0]` is equivalent to `arr`.
- Also, value inside the address `&arr[0]` and address `arr` are equal.
- Value in address `&arr[0]` is `arr[0]` and value in address `arr` is `*arr`.
- Hence, `arr[0]` is equivalent to `*arr`.

`&a[1]` is equivalent to `(a+1)` AND, `a[1]` is equivalent to `*(a+1)`. `&a[2]` is equivalent to `(a+2)` AND, `a[2]` is equivalent to `*(a+2)`. `&a[3]` is equivalent to `(a+3)` AND, `a[3]` is equivalent to `*(a+3)`

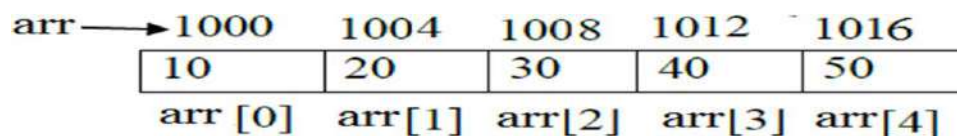
.

.

.

`&a[i]` is equivalent to `(a+i)` AND, `a[i]` is equivalent to `*(a+i)`.

You can declare an array and can use pointer to alter the data of an array.



Pointers and Functions:

There are two ways of passing parameters to the functions:

- a) Pass by value (also called **call by value**).
- b) Pass by reference (also called **call by reference** OR **call by address**).

a) **Pass by value:** Whatever the changes done inside the function is not reflected outside the function.

If the values of the formal parameters changes in the function, but the values of the actual parameters are not changed. Then it is called pass by value or call by value.

Example for Call by Value.

Program to swap two numbers

```
void main()                                void swap(int x,int y)
{
    int x=10,y=20;                          {
    printf("x=%d y=%d",x,y);                int temp;
    swap(x,y);                             temp=x;
    printf("x=%d y=%d",x,y);               x=y;
    getch();                             y=temp;
}                                           printf("x=%d y=%d",x,y);
                                           }
```

After execution of this program, the first printf() function prints the value x=10,y=20, then it calls the function swap(x,y). In the function swap(x,y), the values of x and y are interchanged. So in the swap(x,y), the printf() function prints x=20 and y=10, then the control will be transferred to main() function and the printf function prints the value x=10 and y=20. This indicates that, in call by value, whatever modifications done to the formal parameters, it will be reflected in that particular function, outside the function the value will remain same as actual parameter.

b) **Pass by reference:** Whatever the changes done inside the function is reflected inside the function

C Programming for Problem Solving (18CPS23)

as well as outside the function.

If the values of the formal parameters changes in the function, and the values of the actual parameters are also changed. Then it is called pass by reference or call by address.

Example for call by reference.

Program to swap two numbers

```
void main()                                void swap(int *x,int *y)
{
    int x=10,y=20;                          {
    printf("x=%d y=%d",x,y);                int temp;
    swap(&x,&y);                             temp=*x;
    printf("x=%d y=%d",x,y);                *x=*y;
    getch();                               *y=temp;
}                                           printf("x=%d y=%d",x,y);
}
```

After execution of this program, the first printf() function prints the value x=10,y=20, then it calls the function swap(&x, &y), In the function swap(&x, &y), the values of x and y are interchanged. So in the swap(&x, &y), the printf() function prints x=20 and y=10, then the control will transferred to main() function and the printf function prints the value x=20 and y=10. This indicates that, in call by reference, whatever modifications done to the formal parameters, it will be reflected in that function and also outside the functions the value will be changed.

difference between call by value and call by address

call by value

1. When a function is called the values of variables are passed.
2. The type of formal parameters should be same as type of actual parameters.
3. Formal parameters contains the values of actual parameters
4. Change of formal parameter in the function will not affect the actual parameters in the calling function.
5. Execution is slower since all the values copied into formal parameters.

call by address

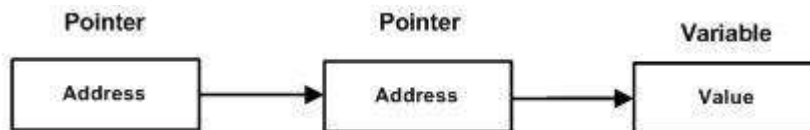
1. When a function is called the addresses of variables are passed.
2. The type of formal parameters should be same as type of actual parameters, but they have to be declared as pointers.
3. Formal parameters contains the addresses of actual parameters.
4. The actual parameters are changed, since the formal parameters indirectly manipulate the actual parameters.
5. Execution is faster since only Addresses are copied.

Write a C program to concatenate two strings using pointers

```
#include<stdio.h>
#include<conio.h> void
main()
{
    char str1[25],str2[50]; int
    i=0,j=0;
    clrscr();
    printf("Enter First String: ");
    gets(str1);
    printf("Enter Second String: ");
    gets(str2);
    while(str1[i]!='\0')
    {
        i++;
    }
    while(str2[j]!='\0')
    {
        str1[i]=str2[j]; j++;
        i++;
    }
    str1[i]='\0';
    printf("\nConcatenated String is %s",str1); getch();
}
```

Pointer to Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int –

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example –

```
#include <stdio.h>

int main () {

    int var;
    int *ptr;
    int **pptr;

    var = 3000;

    /* take the address of var */
    ptr = &var;

    /* take the address of ptr using address of operator & */
    pptr = &ptr;

    /* take the value using pptr */
    printf("Value of var = %d\n", var );
    printf("Value available at *ptr = %d\n", *ptr );
    printf("Value available at **pptr = %d\n", **pptr);

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

For example programs and solutions refer the following links in these links you have an option to see the solution as well as option goto editor in which you can type your program and run it and see the output online. You can type all the programs in the notes and also lab manual and execute it here.

<https://www.w3resource.com/c-programming-exercises/pointer/index.php>

Refer the following links for online lecture

https://www.youtube.com/watch?v=Z_0xXmOgYtY

<https://www.youtube.com/watch?v=nAGjoysNM4s>

C Preprocessor Directives

The **C Preprocessor** is not part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. All preprocessor commands begin with a pound symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column. Following section lists down all important preprocessor directives:

Directive	Description
#define	Substitutes a preprocessor macro
#include	Inserts a particular header from another file
#undef	Undefines a preprocessor macro
#ifdef	Returns true if this macro is defined
#ifndef	Returns true if this macro is not defined
#if	Tests if a compile time condition is true
#else	The alternative for #if
#elif	#else an #if in one statement
#endif	Ends preprocessor conditional
#error	Prints error message on stderr
#pragma	Issues special commands to the compiler, using a standardized method

These directives can be divided into three categories

- 1. Macro substitution**
- 2. File inclusion directives**
- 3. Compiler control directives**

1. Macro Substitution

Macro substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. The preprocessor accomplishes this task under the direction of #define statement. This statement, usually known as a macro definition (macro) takes the following general form

```
#define identifier string
```

The string may be any text, while the identifier must be valid C name. The most common forms are:

- Simple Macro substitution
- Argumented substitution
- Nested macro substitution

1.1 Simple macro substitution:

In simple macro substitution, an identifier (macro) is simply replaced by a string. This can be done by using the directive `#define`. It takes the following general form:

```
#define identifier string
```

Where Identifier must be a valid C name and string may be any text. When we use this identifier in our program then this identifier is known as MACRO and `#define` directive only replaces the MACRO by string.

Simple macro substitution is commonly used to define symbolic constants.

Examples of Symbolic constants:

```
#define PI 3.14
```

```
#define X 100
```

```
#define P printf
```

Note : It is a convention to write all macros(identifiers) in capitals to identify them as symbolic constants.

2.2 Argumented macro substitution:

As the function, macro can have arguments. The preprocessor permits us to define more complex and more useful form of replacements it takes the following form:

```
# define identifier( f1,f2,f3.....fn) string
```

Note: There is no space between identifier and left parentheses and the identifier f1, f2, f3 fn is analogous to formal arguments in a function definition. A simple example of a macro with arguments is

```
# define CUBE (x) (x*x*x)
```

CUBE(x) is macro with arguments

If the following statements appears later in the program,

```
Volume=CUBE (3);
```

The preprocessor would expand the statement to

```
volume =(3*3*3);
```

Then the variable volume value will be 27.

Undefining a macro:

A macro defined with `#define` directives can be undefined with `#undef` directive.

Syntax: `#undef identifier`

Where identifier is the name of macro It is useful when we do not want to allow the use of macros in any portion of the program.

Example program that defines a macro

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
#define PI 3.14
```

```
void main()
```

```
{
```

```
    int r=10,a;
```

```
    clrscr();
```

```
a= PI*r*r;

printf("area of circle is %d", a);

}
```

Example: program that undefines a macro.

```
#include <stdio.h>
#include<conio.h>
#define PI 3.14
void main()
{
    printf (" PI =%d", PI);

    #undef PI

    printf(" PI =%d" ,PI);

    getch();
}
```

Nested Macros

We can use one macro inside the definition of another macro . Such macros are known as nested macros. Example for nested macro is shown below

```
#define SQUARE(x)x*x
#define CUBE(x)      SQUARE(x)*x
```

File inclusion directives

An external file containing functions or macro definitions can be included as a part of program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directive `#include "filename"`

Where filename is the name of the file containing the required definitions or functions.

Alternatively this directive can take the form

```
#include <filename>
```

When the file is include with in " " the search for the file is made first in the current directory and then in the standard directories.

When the file is included with in < > the file is searched only in the standard directories.

Note: If an included file is not found, an error is reported and compilation is terminated.

Compiler control directives: These directives allow the compiler to compile selected portion of the source code based on some condition. This is known as "conditional compilation".

Six directives are available to control conditional compilation. They delimit blocks of program text that are compiled only if a specified condition is true. These directives can be nested. The program text within the blocks is arbitrary and may consist of preprocessor directives, C statements, and so on. The beginning of the block of program text is marked by one of three directives:

`#if`
`#ifdef`
`#ifndef`

Optionally, an alternative block of text can be set aside with one of two directives:

`#else`
`#elif`

The end of the block or alternative block is marked by the `#endif` directive.

If the condition checked by `#if` , `#ifdef` , or `#ifndef` is true (nonzero), then all lines between the matching `#else` (or `#elif`) and an `#endif` directive, if present, are ignored.

If the condition is false (0), then the lines between the `#if` , `#ifdef` , or `#ifndef` and an `#else` , `#elif` , or `#endif` directive are ignored.

These compiler control directives are used in different situations. They are

Directive	Purpose
<code>#ifdef</code>	Test for a macro definition
<code>#endif</code>	Specifies the end of <code>#if</code>
<code>#ifndef</code>	Tests whether a macro is not defined
<code>#if</code>	Test a compile-time condition
<code>#else</code>	Specifies alternative when <code>#if</code> fails

You have included a file containing some macro definitions. It is not known whether a certain macro has been defined in that header file. However, you want to be certain that the macro is defined.

This situation refers to the conditional definition of a macro. We want to ensure that the macro **TEST** is always defined, irrespective of whether it has been defined in the header file or not. This can be achieved as follows:

```
#include "DEFINE.H"
#ifndef TEST
#define TEST 1
#endif
```

DEFINE.H is the header that is supposed to contain the definition of **TEST** macro. The directive **#ifndef TEST** searches the definition of **TEST** in the header file and if it is not defined, then all the lines between the **#ifndef** and the corresponding **#endif** directive are executed in the program.

Example: Program to illustrate this concept.

```
#include <stdio.h>
#define MAX 100
void main()
{
    #if (MAX)
        printf("MAX is defined");
    #else
        printf ("MAX is not defined");
    #endif
}
```

Output:

MAX is defined

Preprocessors Examples

Analyze the following examples to understand various directives.

#define MAX_ARRAY_LENGTH 20 This directive tells the C Preprocessor to replace instances of **MAX_ARRAY_LENGTH** with 20.

Use *#define* for constants to increase readability.

#include <stdio.h> #include "myheader.h" These directives tell the C Preprocessor to get **stdio.h** from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

#undef FILE_SIZE #define FILE_SIZE 42 This tells the C Preprocessor to undefine existing **FILE_SIZE** and define it as 42.

#ifndef MESSAGE #define MESSAGE "You wish!"

#endif This tells the C Preprocessor to define **MESSAGE** only if **MESSAGE** isn't already defined.

#ifdef DEBUG /* Your debugging statements here */ #endif This tells the C Preprocessor to do the process the statements enclosed if **DEBUG** is defined. This is useful if you pass the *-DDEBUG* flag to gcc compiler at the time of compilation. This will define **DEBUG**, so you can turn debugging on and off on the fly during compilation.

Question Bank

1. Define a Structure. How would you declare and initialize structure variables? Give examples
2. Write a C program to read details of 10 students and print the marks of the student if his name is given as input.
3. Explain the nested structures with example.
4. Define a Pointer. How the pointers are declared and initialized. Give examples
5. Write a program in C to add two numbers using pointers
6. Explain the categories of preprocessor directives in c.