

PROBLEM SOLVING THROUGH PROGRAMMING (18ESCS01)

UNIT-5: STRUCTURES & UNIONS

STRUCTURES & UNIONS

- STRUCTURE DEFINITION & EXAMPLE
- STRUCTURE DECLARATION & INITIALIZATION
- STRUCTURE WITHIN A STRUCTURE
- UNIONS
- PROGRAMS ON STRUCTURES AND UNIONS
- STORAGE CLASSES
- PRE-PROCESSOR DIRECTIVES

DEFINITION OF A STRUCTURE

- Structure is a collection of variables under a single name, and a single name containing multiple elements / members of different data type.
- A structure type is usually defined at the beginning of a program main() statement.
- Syntax & example of a structure:

Syntax of structure	Example of a structure
<pre>struct structure_name/tag_name { data_type variable_name; data_type variable_name; };</pre>	<pre>struct student { int id; char name[20]; float percentage; };</pre>

- We can define a structure called “student” with ³3 elements id, name & percentage.

DECLARATION OF A STRUCTURE

- Declaration of a structure is as follows:

Declaration of structure	Example
<pre>struct structure_name/tag_name { data_type variable_name; data_type variable_name; }; struct member1, member2;</pre>	<pre>struct student { int id; char name[20]; float percentage; }; struct s1, s2;</pre>

- Note: When you first define a structure in a file, the statement simply tells the 'C' compiler that a structure exists, but causes no memory allocation. Only when a structure variable is declared, memory allocation takes place

INITIALIZATION OF A STRUCTURE

- Structure is a user-defined data type.
- Declaration & initialization of a structure is as follows:

Declaration of structure	Example
<pre>struct structure_name/tag_name { data_type variable_name; data_type variable_name; }; struct member1, member2;</pre>	<pre>struct student { int id; char name[20]; float percentage; }; struct s1, s2;</pre>
Initialization of structure	Example
<pre>struct tag_name member1={.....}; struct tag_name member2={.....};</pre>	<pre>struct student s1={10, "ABC", 80.5}; struct student s2={20, "DEF", 75.5};</pre>

ACCESSING THE MEMBERS OF STRUCTURE

- Accessing can be done using dot '.' operator.

Accessing the member of structure	
<pre>student s1={10, "ABC", 80.5};</pre>	<pre>s2.id = 20 s2.name = "DEF" s2.percentage=75.5</pre>
<pre>printf("student id=%d student name=%s student percentage= %d", s1.id, s1.name, s1.percentage);</pre>	<pre>printf("s2.id=%d s2.name=%s s2.percentage= %d", s2.id, s2.name, s2.percentage);</pre>

4 DIFFERENT WAYS OF DECLARING & INITIALIZE OF STRUCTURE

Way 1: Declare & Initialize

```
struct student  
{  
  int id;  
  char name[20];  
  float marks;  
} s1 = {10, "ABC", 75.5};
```

- **Note:** In the above code example, we have seen that structure is declared and as soon as after declaration we have initialized the structure variable.

4 DIFFERENT WAYS OF DECLARING & INITIALIZE OF STRUCTURE

Way 2: Declare & Initialize Multiple Variables

```
struct student
{
int id;
char name[20];
float marks;
};
s1 = {10, "ABC", 75.5};
s2 = {20, "DEF", 80.8};
```

- **Note:** In this example, we have declared two structure variables in above code. After declaration of variable we have initialized two variable.

4 DIFFERENT WAYS OF DECLARING & INITIALIZE OF STRUCTURE

Way 3: Initializing single member

```
struct student
{
int mark1;
int mark2;
int mark3;
} s1 = {67};
```

- **Note:** In the above code example, there are three members of structure, only one is initialized , Then remaining two members are initialized with Zero.
- If there are variables of other data type then their initial values will be:

Data type	Default value if not initialized
integer	0
float	0.00
char	NULL

4 DIFFERENT WAYS OF DECLARING & INITIALIZE OF STRUCTURE

Way 4: Initialising inside main

```
struct student
{
int mark1;
int mark2;
int mark3;
};

void main( )
{
struct student s1 = {45, 54, 55};
.....
.....
};
```

- **Note:** We need to initialize structure variable to allocate some memory to the structure.

IMPORTANT POINTS IN STRUCTURES

1. **Struct keyword is used to declare structure.**
2. **Members of structure are enclosed within opening and closing braces.**
3. **Declaration of Structure reserves no space.**
4. **It is nothing but the “ Template / Map / Shape ” of the structure .**
5. **Memory is created, very first time when the variable is created /Instance is created.**

EXAMPLE PROGRAM ON STRUCTURES

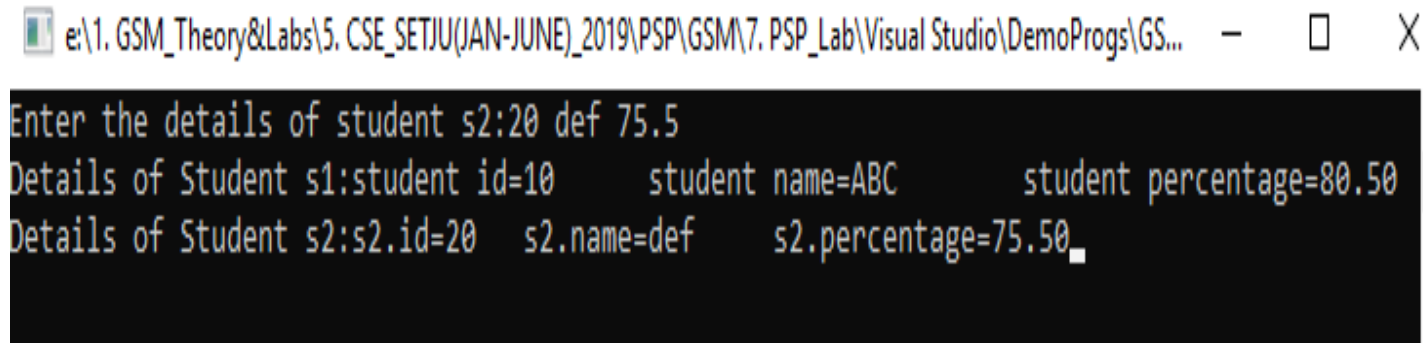
```
#include<stdio.h>   #include<conio.h>

struct student
{
    int id;
    char name[20];
    float percentage;
};

void main( )
{
    struct student s1 = {10, "ABC", 80.5};
    struct student s2;
    printf("Enter the details of student s2:");
    scanf("%d %s %f", &s2.id, s2.name, &s2.percentage);

    printf("Details of Student s1:");
    printf("student id=%d \t student name=%s \t student percentage=%0.2f\n", s1.id, s1.name, s1.percentage);

    printf("Details of Student s2:");
    printf("s2.id=%d \t s2.name=%s \t s2.percentage=%0.2f", s2.id, s2.name, s2.percentage);
    getch( );
}
```



STRUCTURE WITHIN A STRUCTURES/NESTED STRUCTURES

- When a structure contains another structure, it is called **nested structure**.
- **For example:** we have two structures named Address and Employee. To make Address nested to Employee, we have to define Address structure before and outside Employee structure and create an object of Address structure inside Employee structure.

SYNTAX FOR NESTED STRUCTURES

- Syntax for nested structures is as follows:

Syntax of a nested structures	Example
<pre>struct structure1 { }; struct structure2 { struct structure1 obj; };</pre>	<pre>struct Address { char HouseNo[25]; char City[25]; char PinCode[25]; }; struct Employee { int Id; char Name[25]; float Salary; struct Address Add; };</pre>

EXAMPLE FOR NESTED STRUCTURES

```
#include<stdio.h>

struct Address
{ char HouseNo[25];
  char City[25];
  char PinCode[25];
};

struct Employee
{ int Id;
  char Name[25];
  float Salary;
  struct Address Add;
};

void main( )
{ int i;
  struct Employee E;

  printf("\n \t Enter Employee Id : ");
  scanf("%d", &E.Id);

  printf("\n \t Enter Employee Name : ");
  scanf("%s", &E.Name);

  printf("\n \t Enter Employee Salary : ");
  scanf("%f", &E.Salary);

  printf("\n \t Enter Employee House No : ");
  scanf("%s", &E.Add.HouseNo);

  printf("\n \t Enter Employee City : ");
  scanf("%s", &E.Add.City);

  printf("\n \t Enter Employee House No : ");
  scanf("%s", &E.Add.PinCode);

  printf("\n Details of Employees");
  printf("\n \t Employee Id : %d", E.Id);
  printf("\n \t Employee Name : %s", E.Name);
  printf("\n \t Employee Salary : %f", E.Salary);
  printf("\n \t Employee House No : %s", E.Add.HouseNo);
  printf("\n \t Employee City : %s", E.Add.City);
  printf("\n \t Employee House No : %s", E.Add.PinCode);
}
```

UNIONS

- Unions are user-defined data type.
- Syntax is as follows:

Syntax of a union	Example
<pre>union union_name/tag_name { data_type variable_name; data_type variable_name; };</pre>	<pre>union student { int id; char name[20]; float percentage; };</pre>

- All elements cant be processed at a time.
- Stored the elements individually and shares the same memory location.

EXAMPLE PROGRAM ON UNIONS

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

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

```
union uni
```

```
{
```

```
    short a;
```

```
    short b;
```

```
};
```

```
void main()
```

```
{
```

```
    union uni var;
```

```
    var.a = 10;
```

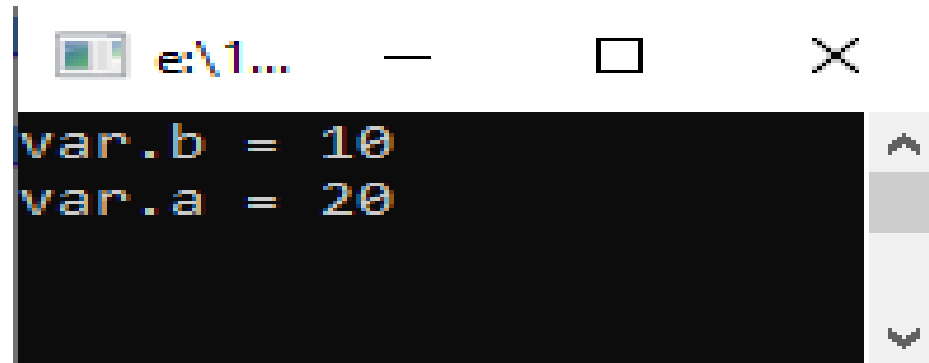
```
    printf("var.b = %d \n", var.b);
```

```
    var.b = 20;
```

```
    printf("var.a = %d", var.a);
```

```
    getch( );
```

```
}
```



```
e:\1...  
var.b = 10  
var.a = 20
```

DIFFERENCE BETWEEN STRUCTURE & UNION

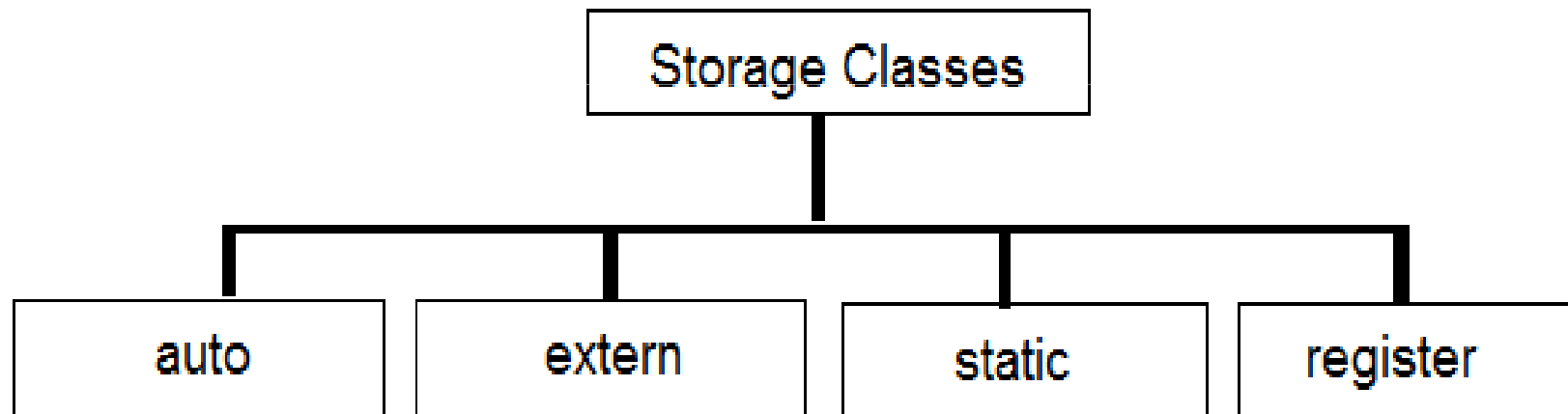
	STRUCTURE	UNION
Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

STORAGE CLASSES

- Storage class in 'C' decides the part of storage to allocate memory for a variable.
- Each variable has a storage class which decides the following parameters:
 - **Scope:** where the value of the variable would be available inside a program.
 - **lifetime** of that variable: for how long will that variable exist.
 - **default initial value:** if we do not explicitly initialize that variable, what will be its default initial value.
- The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'.

STORAGE CLASSES - TYPES

- Four storage classes are most used in C programming:
 1. Automatic variables - auto
 2. External variables - extern
 3. Static variables - static
 4. Register variables – register



- Syntax of storage classes:

Syntax of structure	Example
storage_class data_type var_name;	auto int a;

STORAGE CLASSES - TYPES

Storage Class	Declaration Location	Scope (Visibility)	Lifetime (Alive)
auto	Inside a function/block	Within the function/block	Until the function/block completes
register	Inside a function/block	Within the function/block	Until the function/block completes
extern	Outside all functions	Entire file plus other files where the variable is declared as extern	Until the program terminates
static (local)	Inside a function/block	Within the function/block	Until the program terminates
static (global)	Outside all functions	Entire file in which it is declared	Until the program terminates

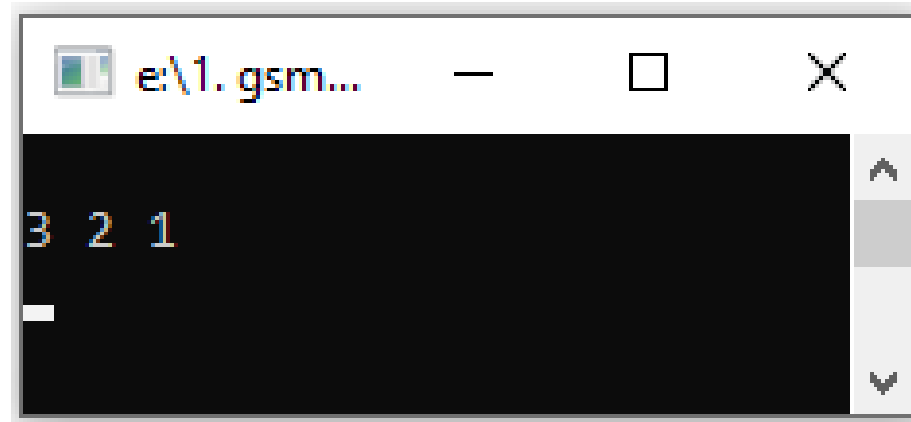
STORAGE CLASSES - AUTO

- A variable defined within a function or block with auto specifier belongs to automatic storage class.
- All variables defined within a function or block by default belong to automatic storage class if no storage class is mentioned.
- Variables having automatic storage class are local to the block which they are defined in, and get destroyed on exit from the block.

STORAGE CLASSES – AUTO EXAMPLE

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

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



```
e:\1. gsm...
3 2 1
_
```

In this example program, we have three definitions for variable 'i'. Here, we may be thinking if there could be more than one variable with the same name. Yes, there could be if these variables are defined in different blocks. So, there will be no error here and the program will compile and execute successfully.

STORAGE CLASSES - EXTERN

- The extern specifier gives the declared variable external storage class.
- When extern specifier is used with a variable declaration then no storage is allocated to that variable.
- The principal use of extern is to specify that a variable is declared with *external linkage* elsewhere in the program.

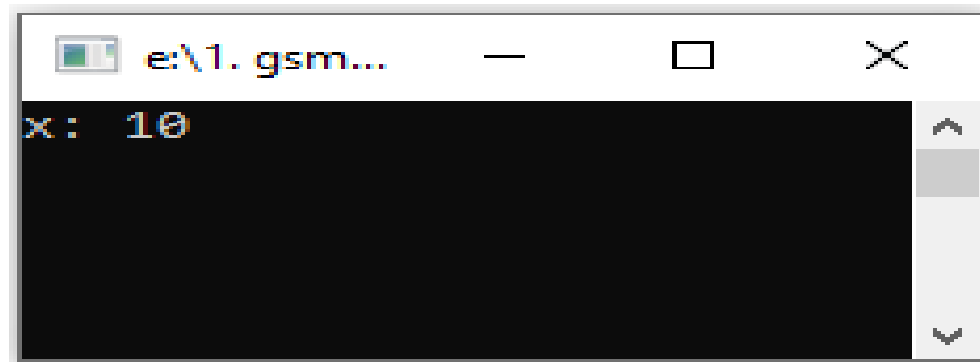
STORAGE CLASSES – EXTERN EXAMPLE

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

extern int x;

int main()
{
    printf("x: %d\n", x);
    getch();
}

int x = 10;
```

A screenshot of a Windows command prompt window. The title bar shows the file path 'e:\1. gsm...'. The command prompt displays the output 'x: 10' in a monospaced font. The window has standard Windows controls (minimize, maximize, close) in the title bar.

If you change the statement `extern int x;` to `extern int x = 50;` you will again get an error *"Redefinition of 'x'"* because with `extern` specifier the variable cannot be initialized, if it is defined elsewhere.

STORAGE CLASSES - STATIC

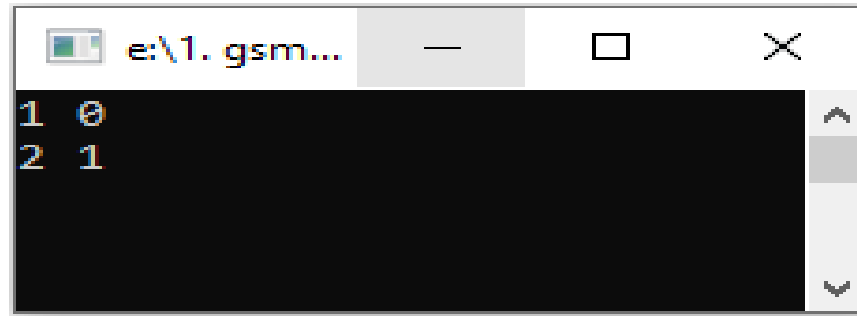
- The static specifier gives the declared variable static storage class.
- Static variables can be used within function or file.
- Unlike global variables, static variables are not visible outside their function or file, but they maintain their values between calls.
- The static specifier has different effects upon local and global variables.

STORAGE CLASSES – STATIC EXAMPLE

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

int staticDemo( )
{
    static int i;
    {
        static int i = 1;
        printf("%d ", i);
        i++;
    }
    printf("%d\n", i);
    i++;
    getch( );
}

int main( )
{
    staticDemo( );
    staticDemo( );
}
```



```
e:\1. gsm...
1 0
2 1
```

For example, the following program code defines static variable `i` at two places in two blocks inside function `staticDemo()`.

Function `staticDemo()` is called twice within from main function. During second call static variables retain their old values and they are not initialized again in second call of `staticDemo()`.

STORAGE CLASSES - REGISTER

- The register specifier declares a variable of register storage class.
- Variables belonging to register storage class are local to the block which they are defined in, and get destroyed on exit from the block.
- A register declaration is equivalent to an auto declaration, but hints that the declared variable will be accessed frequently; therefore they are placed in CPU registers, not in memory.

STORAGE CLASSES – REGISTER EXAMPLE

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

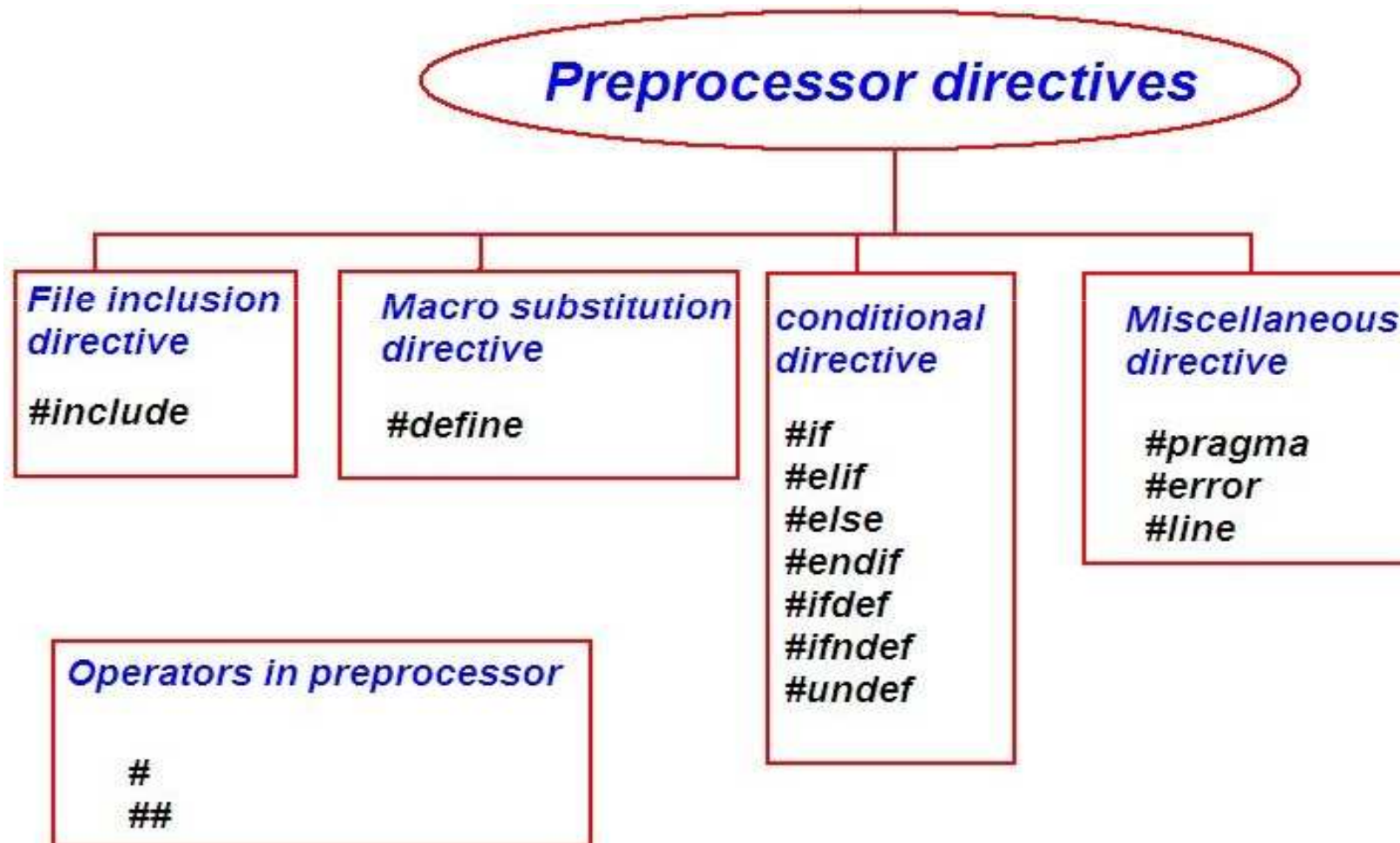
int main( )
{
    register int i = 10;
    int *p = &i;    //error: address of register variable requested

    printf("Value of i: %d", *p);
    printf("\n Address of i: %u", p);
    getch( );
}
```

The following piece of code is trying to get the address of variable i into pointer variable p but it won't succeed because i is declared register; therefore following piece of code won't compile and exit with error *"error: address of register variable requested"*.

PRE-PROCESSOR DIRECTIVES

- The 'C' Pre-processor executes before a program.
- Pre-processor directives begin with '#' symbol.



PRE-PROCESSOR DIRECTIVES

Directive	Summary of Meaning
#define identifier	Defines a compilation symbol.
#undef identifier	Undefines a compilation symbol.
#if expression	If the expression is true, the compiler compiles the following section.
#elif expression	If the expression is true, the compiler compiles the following section.
#else	If the previous #if or #elif expression is false, the compiler compiles the following section.
#endif	Marks the end of an #if construct.
#region name	Marks the beginning of a region of code; has no compilation effect.
#endregion name	Marks the end of a region of code; has no compilation effect.
#warning message	Displays a compile-time warning message.
#error message	Displays a compile-time error message.
#line indicator	Changes the line numbers displayed in compiler messages.
#pragma text	Specifies information about the program context.