CHAPTER

STORAGE CLASSES AND PREPROCESSOR DIRECTIVES

17.1 STORAGE CLASSES

Scope, Visibility and Lifetime of Variables in Functions.

In C, not only do all the variables have data type, they also have storage classes. The following storage classes are more relevant to functions.

1. Automatic variables
2. External variables
3. Static variables
4. Register variables

The above storage classes have their own scope, visibility and life time. Scope of a variable defines up to what part of a program the variable is active. Visibility refers to the accessibility of a variable from the memory.

17.1.1 Automatic Variables

Automatic variables are created and utilized within the function. It can be destroyed automatically when it exits from the function. Hence, the name automatic. The keyword auto can be used explicitly to declare automatic variables. They are local to the particular functions only. Because of this property, automatic variables are also referred as local or internal variables. A variable declared inside a function without storage class specification is, by default an automatic variable.

Example

```
void addition()
{
auto int num;
----------
```

```
---------
}
```

The above example shows that the same variable name is used in more than one function because it can be created and destroyed within the same function without any confusion to the compiler.

EXAMPLE PROGRAM

```
#include<stdio.h>
#include<stdio.h>
#include<conio.h>
void fun1(void);
void fun2( );
void main( )
{
int m = 1000;
fun2( );
printf("\n %d", m);
}

void fun1( )
{
int m = 100;
printf("\n %d", m);
}

void fun2( )
{
auto int m = 10;
fun1( );
printf:\n %d", m);
}
```

Output

```
10
100
1000
```

17.1.2 External Variables

External variables are declared globally above all function definitions including main() function. The key word used is **extern.** These variables are active and alive throughout the program. Unlike normal variables, external variables are accessed by any function in the program. The memory allocated for

external variables are shared by all the functions utilizing it. So any modification done by any function will affect the original value. So subsequent functions can refer only that new value.

EXAMPLE PROGRAM

```
#include<stdio.h>
#include<conio.h>
int  fun1( );
int fun2( );
int fun3( );
int N;   // global declaration
void main()
{
N = 10;
printf("\n   N = %d",N);
printf("\n   N = %d", fun1( ));
printf(:\n   N = %d", fun2( ));
printf("\n   N = %d", fun3( ) );
}

int fun1( )
{
  N = N +10;     // global declaration
}
int fun2( )
{
  int N;         // local declaration
  N = 5;
  return(N);
}

int fun3()
{
  N = N+10;      // global declaration
}
```

Output

```
N = 10
N = 20
N = 5
N = 30
```

One other aspect of a global variable is that it is available from the point of declaration to the end of the program.

Example

```
main()
{
y = 5;
---------
--------
}
int y;  // global declaration
fun( )
{
y = y + 1;
}
```

As far as main is considered, the variable y is undefined. So the compiler generates an error message.

External declaration

The above issue can be solved by the storage declaration **extern**as follows:

```
 main()
{
   extern int y = 5;   // external declaration
---------
--------
}
fun( )
{
   extern int y = y + 1;    // external declaration
----------
----------
}
int y;  //  definition
```

The variable y has been declared after both the function definitions. The extern declaration of y in both the functions informs to the compiler that y is an integer type variable defined somewhere in the same program. The extern declaration does not allocate memory for its storage.

17.1.3 Static Variables

The static variable values are persisted until the end of the program. Any variable can be declared static using the keyword static. A static variable is initialized only once, when a program is compiled. It is never initialized again. Further reference take the preassigned value as it is.

```
static int x;
static float y;
```

A static variable may be internal or external depending on the place of declaration. Internal static variables are declared inside the function. The scope of internal variables extends up to the end of the program. It is similar to auto variables.

EXAMPLE PROGRAM

```
#include<stdio.h>
      #include<stdio.h>
       #include<conio.h>
       void stat( );
       main ()
     {
        int  i ;
        for ( i = 1; i < = 3 ; i++)
        stat( );
     }

void stat( )
 {
  static int x = 0;
  x = x + 1;
  printf("\n  X = %d", x);
  }
```

Output

```
X = 1
X = 2
X = 3
```

During the first call to **stat, x** is incremented to 1. Because x is static, this value persists and therefore next call to **stat** adds another 1 to **x**, now the value of x is 2.

17.1.4 Register Variables

Normally the computer will store all the variables in a physical memory. Sometimes it is possible to store the values in registers, as register access is much faster than memory access. So the frequently accessed variables are kept in registers which increases the speed of execution.

Declaration of register variable

Syntax:

register int count;

Registers are able to store only restricted size value, so most of the compilers will allow only int and char variables to be placed in the register. Registers are limited in count, therefore it is important to select required variables for this purpose. The C compiler automatically converts **register** variables into non register variables once the limit is reached.

EXAMPLE PROGRAM

```
#include<stdio.h>
int main()
{
int n1,n2;
register int sum;
printf("\nEnter the Number 1 : ");
scanf("%d",&n1);
printf("\nEnter the Number 2 : ");
scanf("%d",&n2);
sum = n1 + n2;
printf("\nSum of Numbers : %d",sum);
return(0);
}
```

Output

```
Enter the number 1:55
Enter the Number 2:40
Sum of Numbers: 95
```

PREPROCESSOR DIRECTIVE

17.2 INTRODUCTION

Preprocessing is defined as the process in which the source code is processed by a program before a C program is compiled in a compiler

Preprocessor directives are the commands used in preprocessor and they start with "#" symbol.

Below is the list of preprocessor directives in C language.

Table 17.1 *Preprocessor Directives in C language*

S. No.	Preprocessor	Syntax	Description
1.	Macro	#define	A constant value is defined by this macro and can be any of the basic data types.
2.	Header file inclusion	#include <file_ name>	The source code of the file "file_name" is included in the main program at the specified place.
3.	Conditional compilation	#ifdef, #endif, #if, #else, #ifndef	Source program includes or excludes a set of commands before compilation with respect to the condition.
4.	Other directives	#undef, #pragma	#undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program

A program in C language involves different processes.

Example program for #define, #include preprocessors in C

Program process flow	File name in each step	Description
Source code	test.c	->Preprocessor replaces #define(macro) #include(files), conditional compilation codes like #ifdef, #ifndef by their respective value and source codes in source file
↓		
Expanded source code	test.i	->Expand source code to assembly source code
↓		
Assembly source code	test.s	->Converts assembly source code to object code
↓		
Object code	test.o	-> This is a program that converts object code to executable code and also combines all object codes
↓		
Executable code	test.exe	->Executable code is loaded and executed by loader
↓		
Execution		

#define This macro defines constant value and can be any of the basic data types.

#include <file_name> The source code of the file "file_name" is included in the main C program where "#include <file_name>" is mentioned.

EXAMPLE

```
#include <stdio.h>
#include <stdio.h>
 #define height 300
#define num 3.14
#define letter 'A'
#define letter_seq "LOCHANA"
#define backslash_char '\?'

void main()
```

```
{
   printf("height    : %d \n", height );
   printf("number : %f \n", num );
   printf("letter : %c \n", letter );
   printf(" letter_sequence : %s \n", letter_seq);
   printf("backslash_char  : %c \n", backslash_char);

}
Output:
 height : 300
 number : 3.140000
 letter : A
 letter_sequence : LOCHANA
 backslash_char : ?
```

Output

```
height: 300
number: 3.140000
letter: A
letter_sequence: XYZ
backslash_char: ?
```

17.2.1 File Inclusion

A program also includes an external function containing the functions or macro definitions.

Syntax

#include"filename"
Or
#include<filename>

17.3 CONDITIONAL COMPILATION

17.3.1 Program for #ifdef, #else and #endif in C

To check whether particular macro is defined or not, "#ifdef" directive is used. If it is defined,then "If" clause statements are included in the source file. Else, the source file includes "else" clause statements for compilation and execution.

EXAMPLE

```
#include <stdio.h>
#define RAVI 200
int main()
{
   #ifdef RAVI
```

```
    printf("RAVI is defined. So, this line will be added in " \
           "this C file\n");
    #else
    printf("RAVI is not defined\n");
    #endif
    return 0;
}
```

Output

```
RAVI is defined. So this line will be added in this C file.
```

17.3.2 Program for #ifndef and #endif in C

#ifndef exactly acts as a reverse of #ifdef directive. The source file includes the "if" clause statements if a particular macro is not defined. Else, the source file includes the "if" clause statements for compilation and execution.

EXAMPLE

```
#include <stdio.h>
#define RAJ 200
int main()
{
    #ifndef VIJAY
    {
       printf("VIJAY is not defined. So, now we are going to " \
              "define here\n");
       #define VIJAY 300
    }
    #else
    printf("VIJAY is already defined in the program");
    #endif
    return 0;
}
```

Output

```
VIJAY is not defined. So now we are going to define here
```

17.3.3 Program for #if, #else and #endif in C

If the given condition is true, the source file includes the "if" clause statements. Else, the source file includes the else clause statement for compilation and execution.

EXAMPLE

```
#include <stdio.h>
#define b 100
```

```
int main()
{
   #if (b==100)
   printf("This line will be added in this C file since " \
          "b \= 100\n");
   #else
   printf("This line will be added in this C file since " \
          "b is not equal to 100\n");
   #endif
   return 0;
}
```

Output

```
This line will be added in this C file since b = 100
```

Example program for undef in C

```
This directive undefines existing macro in the program.
#include <stdio.h>
#define height 150
void main()
{
   printf("First defined value for height    : %d\n",height);
   #undef height          // undefining variable
   #define height 600     // redefining the same for new value
   printf("value of height after undef \& redefine:%d",height);
}
```

Output

```
First defined value for height: 150
Value of height after undef & redefine: 600
```

17.4 OTHER DIRECTIVES

To call a function before and after the main function, pragma is used.

EXAMPLE

```
#include <stdio.h>

void fun1( );
void fun2( );

#pragma startup fun1
#pragma exit fun2

int main( )
{
   printf ( "\n Now we are in main function" ) ;
   return 0;
}
void fun1( )
{
```

```
   printf("\nFunction1 is called before main function call");
}
void fun2( )
{
   printf ( "\nFunction2 is called just before end of " \
            "main function" ) ;"
}
```

Output

```
Function1 is called before main function call
Now we are in main function.
Function2 is called just before the end of main function
```

Table 17.2 *Pragma Commands in C language*

S. No.	Pragma Command	Description
1.	#Pragma startup <function_ name_1>	This directive executes function named "function_name_1" before
2.	#Pragma exit <function_ name_2>	This directive executes function named "function_name_2" just before termination of the program
3.	#pragma warn – rvl	If function doesn't return a value, then warnings are suppressed by this directive while compiling
4.	#pragma warn – par	If function doesn't use passed function parameters, then warnings are suppressed

17.5 PREDEFINED MACROS

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Table 17.3 *Predefined Macros in C language*

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format
__TIME__	The current time as a character literal in "HH:MM:SS" format
__FILE__	This contains the current filename as a string literal
__LINE__	This contains the current line number as a decimal constant
__STDC__	Defined as 1 when the compiler complies with the ANSI standard

EXAMPLE

```
#include <stdio.h>
main() {
```

```
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );
}
```

Output

```
File:test.c
Date:Jun 2 2016
Time:04:15:15
Line:8
ANSI:1
```

17.6 PREPROCESSOR OPERATORS

The C preprocessor offers the following operators to help create macros:

17.6.1 The Macro Continuation (\) Operator

A macro is normally confined to a single line. To continue a macro that is too long for a single line, the continuation macro(\ ) is used.

Example

```
#define  message_for(a, b)  \
printf(#a " and " #b ": CSE SJIT!\n")
```

17.6.2 The Stringize (#) Operator

The macro parameter is converted into a string constant when the stringize or number-sign operator ('#') is used within a macro definition. This operator may be used only in a macro having a specified argument or parameter list.

EXAMPLE

```
#include <stdio.h>
#define  message_for(a, b)
   printf(#a " and " #b ": Welcome!\n")
int main(void) {
   message_for(Leon,Karguvel);
   return 0;
}
```

Output

```
Leon and Karguvel: Welcome!
```

17.6.3 The Token Pasting (##) Operator

The tokenpasting operator (##) within a macro definition is used to combine two arguments. It allows two separate tokens in the macro definition to be combined into a single token.

EXAMPLE

```
#include <stdio.h>
#define tokenpaster(n) printf ("token" #n " = %d", token##n)
int main(void) {
   int token34 = 40;
   tokenpaster(34);
   return 0;
}
```

Output

```
token34 = 40
```

17.6.4 The defined() Operator

The defined operator() is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). The value is false (zero) if the symbol is not defined.

EXAMPLE

```
#include <stdio.h>
#if !defined (MESSAGE)
   #define MESSAGE "Help!"
#endif
int main(void) {
   printf(" The message is : %s\n", MESSAGE);
   return 0;
}
```

Output

```
The message is: Help!
```

17.7 PARAMETERIZED MACROS

Parameterized macros are one of the powerful functions of C which has the ability to stimulate functions.

EXAMPLE

```
#include<stdio.h>
#define MAX(x,y) ((x) > (y) ? (x) : (y))
int main(void) {
    printf("Max between 20 and 10 is %d\n", MAX(10, 20));
    return 0;
}
```

Output

```
Max between 20 and 10 is 20
```