**17. Storage classes & Preprocessor directives**

**17.1 STORAGE CLASSES:**

Scope, visibility and life time of a 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 its own scope, visibility and life time. Scope of a variable defines up to what part a program the variable is active. Visibility refers to the accessibility of a variable from the memory.

* + 1. **Automatic variables**

Automatic variables created and utilized within the function. It can be destroyed automatically when it exit from the function. Hence the name automatic. The keyword auto can be used explicitly to declare automatic variables. They are local to the particular function 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 main()

{

int num;

---------

--------

}

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<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:**

The key word used is **extern.**

External variables are declared globally, above all function definitions including main() function. 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 which utilizing it. So any modification done by any function will affect the original value. So subsequent functions can reference 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 definition. The extern declaration of y in both the function 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 variables values are persist 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 pre assigned 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 variable 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<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. Since register access is much faster than memory access. So keep the frequently accessed variables 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,

X.

**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**

Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.

Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol.

Below is the list of preprocessor directives in C language.

|  |  |  |  |
| --- | --- | --- | --- |
| s.no | Preprocessor | Syntax | Description |
| 1 | Macro | #define | This macro defines constant value 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 | Set of commands are included or excluded in source program 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 into different processes.

Example program for #define, #include preprocessors in C:

***Program process flow*** ***File name in Description***

***each steps***

Source code

test.c

->Preprocessor replaces #define(macro) #include(files),conditional compilation codes like #ifdef, #ifndef by their Respective value & 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

Execution

->Executable code is loaded and executed by loader

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

#include<filename> - The source code of the file "filename" is included in the main program.

#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>

 #define height 100

#define number 3.14

#define letter 'A'

#define letter\_sequence "ABC"

#define backslash\_char '\?'

void main()

{

   printf("value of height    : %d \n", height );

   printf("value of number : %f \n", number );

   printf("value of letter : %c \n", letter );

   printf("value of letter\_sequence : %s \n", letter\_sequence);

   printf("value of backslash\_char  : %c \n", backslash\_char);

}

Output:

value of height : 100

value of number : 3.140000

value of letter : A

value of letter\_sequence : ABC

value of backslash\_char : ?

**17.2 .1 File inclusion:**

An external file containing functions or Marco definitions can be included as a part of a program.

Syntax:

#include”filename”

Or

#include<filename>

Example program for conditional compilation directives:

**17.3 CONDITIONAL COMPILATION**

**17.3.1.   program for #ifdef, #else and #endif in C:**

“#ifdef” directive checks whether particular macro is defined or not. If it is defined, “If” clause statements are included in source file.

Otherwise, “else” clause statements are included in source file for compilation and execution.

Example:

#include <stdio.h>

#define RAVI 100

 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 reverse as #ifdef directive. If particular macro is not defined, “If” clause statements are included in source file.

Otherwise, else clause statements are included in source file for compilation and execution.

Example:

#include <stdio.h>

#define RAJ 100

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” clause statement is included in source file if given condition is true.

Otherwise, else clause statement is included in source file for compilation and execution.

Example:

#include <stdio.h>

#define a 100

int main()

{

   #if (a==100)

   printf("This line will be added in this C file since " \

          "a \= 100\n");

   #else

   printf("This line will be added in this C file since " \

          "a is not equal to 100\n");

   #endif

   return 0;

}

Output:

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

Example program for undef in C:

This directive undefines existing macro in the program.

Example:

#include <stdio.h>

#define height 120

void main()

{

   printf("First defined value for height    : %d\n",height);

   #undef height          // undefining variable

   #define height 500     // redefining the same for new value

   printf("value of height after undef \& redefine:%d",height);

}

Output:

First defined value for height : 120

value of height after undef & redefine : 500

**17.4 OTHER DIRECTIVES**

Pragma is used to call a function before and after main function in a C program.

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 end of main function

|  |  |  |
| --- | --- | --- |
| 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 parameter , 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.

|  |  |
| --- | --- |
| 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. The macro continuation operator (\) is used to continue a macro that is too long for a single line

Example:

#define message\_for(a, b) \

printf(#a " and " #b ": CSE SJIT!\n")

**17.6.2. The Stringize (#) Operator:**

The stringize or number-sign operator ( '#' ), when used within a macro definition, converts a macro parameter into a string constant. 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(Ram,Vijay);

return 0;

}

Output:

Ram and Vijay: Welcome!

**17.6.3 The Token Pasting (##) Operator:**

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined 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 preprocessor 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). If the symbol is not defined, the value is false (zero).

Example:

#include <stdio.h>

#if !defined (MESSAGE)

#define MESSAGE “Help!"

#endif

int main(void) {

printf("Here is the message: %s\n", MESSAGE);

return 0;

}

Output:

Here is the message: Help!

**17.7 Parameterized Macros:**

One of the powerful functions of the C is the ability to simulate functions using parameterized macros.

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