**VISION INSTITUTE OF MANAGEMENT**

**C PROGRAMMING**

**BCA 1st YEAR/2nd SEM**

**UNIT-5(INTRODUCTION C PREPROCESSOR)**

**C PREPROCESSOR**

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

****

The C Preprocessor is a macro Preprocessor (allows you to define macros) that transforms your program before it is compiled. These transformations can be the inclusion of header file, macro expansions etc.

All Preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a Preprocessor directive should begin in the first column. The following section lists down all the important Preprocessor directives –

|  |  |
| --- | --- |
| **S.No.** | **Directive & Description** |
| 1 | **#define**Substitutes a Preprocessor macro. |
| 2 | **#include**Inserts a particular header from another file. |
| 3 | **#undef**Undefines a Preprocessor macro. |
| 4 | **#ifdef**Returns true if this macro is defined. |
| 5 | **#ifndef**Returns true if this macro is not defined. |
| 6 | **#if**Tests if a compile time condition is true. |
| 7 | **#else**The alternative for #if. |
| 8 | **#elif**#else and #if in one statement. |
| 9 | **#endif**Ends preprocessor conditional. |
| 10 | **#error**Prints error message. |

All preprocessing directives begin with a # symbol. **For example,**

#define PI 3.14

**Some of the common uses of C preprocessor are:**

## **Including Header Files: #include**

The #include preprocessor is used to include header files to C programs. **For example,**

#include <stdio.h>

Here, stdio.h is a header file. The #include preprocessor directive replaces the above line with the contents of stdio.h header file.

That's the reason why you need to use #include <stdio.h> before you can use functions like scanf() and printf().

You can also create your own header file containing function declaration and include it in your program using this preprocessor directive.

#include "my\_header.h"

## **Macros using #define**

A macro is a fragment of code that is given a name. You can define a macro in C using the #define preprocessor directive.

**Here's an example**.

#define c 299792458 // speed of light

Here, when we use c in our program, it is replaced with 299792458.

### Example 1: #define preprocessor

#include <stdio.h>

#define PI 3.1415

int main()

{

 float radius, area;

 printf("Enter the radius: ");

 scanf("%f", &radius);

 // Notice, the use of PI

 area = PI\*radius\*radius;

 printf("Area=%.2f",area);

 return 0;

}

## **Function like Macros**

You can also define macros that work in a similar way like a function call. This is known as function-like macros. **For example,**

#define circleArea(r) (3.1415\*(r)\*(r))

Every time the program encounters circleArea(argument), it is replaced by (3.1415\*(argument)\*(argument)).

Suppose, we passed 5 as an argument then, it expands as below:

circleArea(5) expands to (3.1415\*5\*5)

### Example 2: Using #define preprocessor

#include <stdio.h>

#define PI 3.1415

#define circleArea(r) (PI\*r\*r)

int main() {

 float radius, area;

 printf("Enter the radius: ");

 scanf("%f", &radius);

 area = circleArea(radius);

 printf("Area = %.2f", area);

 return 0;

}

## **Conditional Compilation**

In C programming, you can instruct preprocessor whether to include a block of code or not. To do so, conditional directives can be used.

It's similar to if statement with one major difference.

The if statement is tested during the execution time to check whether a block of code should be executed or not whereas, the conditionals are used to include (or skip) a block of code in your program before execution.

### Uses of Conditional

* use different code depending on the machine, operating system
* compile same source file in two different programs
* to exclude certain code from the program but to keep it as reference for future purpose

### How to use conditional?

To use conditional, #ifdef, #if, #defined, #else and #elseif directives are used.

### #ifdef Directive

#ifdef MACRO

 // conditional codes

#endif

Here, the conditional codes are included in the program only if MACRO is defined.

### #if, #elif and #else Directive

#if expression

 // conditional codes

#endif

Here, expression is an expression of integer type (can be integers, characters, arithmetic expression, macros and so on).

The conditional codes are included in the program only if the expression is evaluated to a non-zero value.

The optional #else directive can be used with #if directive.

#if expression

 conditional codes if expression is non-zero

#else

 conditional if expression is 0

#endif

You can also add nested conditional to your #if...#else using #elif

#if expression

 // conditional codes if expression is non-zero

#elif expression1

 // conditional codes if expression is non-zero

#elif expression2

 // conditional codes if expression is non-zero

#else

 // conditional if all expressions are 0

#endif

### #defined

The special operator #defined is used to test whether a certain macro is defined or not. It's often used with #if directive.

#if defined BUFFER\_SIZE && BUFFER\_SIZE >= 2048

 // codes

## **Predefined Macros**

Here are some predefined macros in C programming.

| Macro | Value |
| --- | --- |
| \_\_DATE\_\_ | A string containing the current date |
| \_\_FILE\_\_ | A string containing the file name |
| \_\_LINE\_\_ | An integer representing the current line number |
| \_\_STDC\_\_ | If follows ANSI standard C, then the value is a nonzero integer |
| \_\_TIME\_\_ | A string containing the current date. |

### Example : Get current time using \_\_TIME\_\_

The following program outputs the current time using \_\_TIME\_\_ macro.

#include <stdio.h>

int main()

{

 printf("Current time: %s",\_\_TIME\_\_);

}

**Output**

Current time: 19:54:39

# **C Preprocessors**

As the name suggests Preprocessors are programs that process our source code before compilation. There are a number of steps involved between writing a program and executing a program in C. Let us have a look at these steps before we actually start learning about Preprocessors.



You can see the intermediate steps in the above diagram. The source code written by programmers is stored in the file program.c. This file is then processed by preprocessors and an expanded source code file is generated named program. This expanded file is compiled by the compiler and an object code file is generated named program .obj. Finally, the linker links this object code file to the object code of the library functions to generate the executable file program.exe.

Preprocessor programs provide preprocessors directives which tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a ‘#’ (hash) symbol. This (‘#’) symbol at the beginning of a statement in a C/C++ program indicates that it is a pre-processor directive. We can place these preprocessor directives anywhere in our program. Examples of some preprocessor directives are: *#include*, *#define*, *#ifndef* etc.

**There are 4 main types of preprocessor directives:**

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

**File Inclusion:** This type of preprocessor directive tells the compiler to include a file in the source code program. There are two types of files which can be included by the user in the program:

1. **Header File or Standard files:** These files contains definition of pre-defined functions like printf(), scanf() etc. These files must be included for working with these functions. Different function are declared in different header files. For example standard I/O functions are in ‘iostream’ file whereas functions which perform string operations are in ‘string’ file.

**Syntax**:

#include< file\_name >

where file\_name is the name of file to be included. The ‘<‘ and ‘>’ brackets tells the compiler to look for the file in standard directory.

1. **user defined files:** When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These files can be included as:

#include"filename"

# **Bitwise Operators in C**

In arithmetic-logic unit (which is within the CPU), mathematical operations like: addition, subtraction, multiplication and division are done in bit-level. To perform bit-level operations in C programming, bitwise operators are used.

| Operators | Meaning of operators |
| --- | --- |
| & | [Bitwise AND](https://www.programiz.com/c-programming/bitwise-operators#and) |
| | | [Bitwise OR](https://www.programiz.com/c-programming/bitwise-operators#or) |
| ^ | [Bitwise XOR](https://www.programiz.com/c-programming/bitwise-operators#xor) |
| ~ | [Bitwise complement](https://www.programiz.com/c-programming/bitwise-operators#complement) |
| << | [Shift left](https://www.programiz.com/c-programming/bitwise-operators#left-shift) |
| >> | [Shift right](https://www.programiz.com/c-programming/bitwise-operators#right-shift) |

## **Bitwise AND operator &**

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.

Let us suppose the bitwise AND operation of two integers 12 and 25-

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

 00001100

& 00011001

 \_\_\_\_\_\_\_\_

 00001000 = 8 (In decimal)

### Example #1: Bitwise AND

#include <stdio.h>

int main()

{

 int a = 12, b = 25;

 printf("Output = %d", a&b);

 return 0;

}

**Output**

Output = 8

## **Bitwise OR operator |**

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. In C Programming, bitwise OR operator is denoted by |.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

 00001100

| 00011001

 \_\_\_\_\_\_\_\_

 00011101 = 29 (In decimal)

### Example #2: Bitwise OR

#include <stdio.h>

int main()

{

 int a = 12, b = 25;

 printf("Output = %d", a|b);

 return 0;

}

**Output**

Output = 29

## **Bitwise XOR (exclusive OR) operator ^**

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is denoted by ^.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25

 00001100

^ 00011001

 \_\_\_\_\_\_\_\_

 00010101 = 21 (In decimal)

### Example #3: Bitwise XOR

#include <stdio.h>

int main()

{

 int a = 12, b = 25;

 printf("Output = %d", a^b);

 return 0;

}

**Output**

Output = 21

## **Bitwise complement operator ~**

Bitwise compliment operator is an unary operator (works on only one operand). It changes 1 to 0 and 0 to 1. It is denoted by ~.

35 = 00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

 \_\_\_\_\_\_\_\_

 11011100 = 220 (In decimal)

### Twist in bitwise complement operator in C Programming

The bitwise complement of 35 (~35) is -36 instead of 220, but why?

For any integer n, bitwise complement of n will be -(n+1). To understand this, you should have the knowledge of 2's complement.

### 2's Complement

Two's complement is an operation on binary numbers. The 2's complement of a number is equal to the complement of that number plus 1. For example:

 Decimal Binary 2's complement

 0 00000000 -(11111111+1) = -00000000 = -0(decimal)

 1 00000001 -(11111110+1) = -11111111 = -256(decimal)

 12 00001100 -(11110011+1) = -11110100 = -244(decimal)

 220 11011100 -(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

### Bitwise complement of any number N is -(N+1). Here's how:

bitwise complement of N = ~N (represented in 2's complement form)

2'complement of ~N= -(~(~N)+1) = -(N+1)

### Example #4: Bitwise complement

#include <stdio.h>

int main()

{

 printf("Output = %d\n",~35);

 printf("Output = %d\n",~-12);

 return 0;

}

**Output**

Output = -36

Output = 11

## **Shift Operators in C programming**

There are two shift operators in C programming:

* Right shift operator
* Left shift operator.

### Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

212 = 11010100 (In binary)

212>>2 = 00110101 (In binary) [Right shift by two bits]

212>>7 = 00000001 (In binary)

212>>8 = 00000000

212>>0 = 11010100 (No Shift)

## **Left Shift Operator**

Left shift operator shifts all bits towards left by certain number of specified bits. It is denoted by <<.

212 = 11010100 (In binary)

212<<1 = 110101000 (In binary) [Left shift by one bit]

212<<0 =11010100 (Shift by 0)

212<<4 = 110101000000 (In binary) =3392(In decimal)

### Example #5: Shift Operators

#include <stdio.h>

int main()

{

 int num=212, i;

 for (i=0; i<=2; ++i)

 printf("Right shift by %d: %d\n", i, num>>i);

 printf("\n");

 for (i=0; i<=2; ++i)

 printf("Left shift by %d: %d\n", i, num<<i);

 return 0;

}

Right Shift by 0: 212

Right Shift by 1: 106

Right Shift by 2: 53

Left Shift by 0: 212

Left Shift by 1: 424

Left Shift by 2: 848