UNIT NO.2

- Preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.
- All preprocessor directives starts with hash # symbol.
- Preprocessor directives are commands to the compiler that instruct it to perform certain actions before actual compilation of the code begins.
- These directives are lines of code preceded by the # symbol. They are processed by the preprocessor, which is a program or a part of the compiler that handles these directives.

Here are some common preprocessor directives in C:

#include: This directive is used to include the contents of another file into the current file. It is typically used to include header files that contain function prototypes, macros, and other declarations.

Example: #include <stdio.h>

#define: This directive is used to define macros, which are symbolic names representing a piece of code. Whenever the macro is encountered in the code, it is replaced by its definition before compilation.

Example: #define PI 3.14159

#ifdef, **#ifndef**, **#else**, **#endif**: These directives are used for conditional compilation. They allow parts of the code to be included or excluded based on certain conditions.

Example:

#ifdef DEBUG

#else

#endif

#error: This directive is used to generate a compiler error with a specified error message. It's typically used for compile-time checks.

Example: #ifndef MAX_SIZE

#error "MAX_SIZE must be defined"

#endif

What is File Inclusion?

File inclusion refers to the process of including external files in your C program. This is typically done to reuse code, manage large projects more effectively, and improve the organization of your code.

Types of File Inclusion

Header Files: Header files typically have a **.h** extension and contain function prototypes, constant definitions, and type declarations. They are included at the beginning of your C source files using the **#include** preprocessor directive.

Source Files: Source files have a **.c** extension and contain actual implementations of functions and other code. They are compiled separately and linked together to form the final executable.

Using Header Files

To include a header file in your C program, use the **#include** directive followed by the filename in angle brackets **< >** or double quotes **" "**:

```
#include <stdio.h>

#include "myheader.h"
```

Example: Creating and Including Header Files
Suppose you have a header file named **myheader.h**:

```
// myheader.c

#include <stdio.h>

#include "myheader.h"

void greet() {

    printf("Hello, World!\n");

}
```

Benefits of File Inclusion

- **Code Reusability**: Header files allow you to reuse functions and declarations across multiple source files.
- **Modularization**: By splitting your code into smaller, manageable files, you can improve code organization and readability.
- **Encapsulation**: Header files hide implementation details, providing only the necessary interfaces to other parts of your program.
- File inclusion is a fundamental concept in C programming, allowing you to modularize your code and improve code reuse and organization. By understanding how to use header files effectively, you can write more maintainable and scalable C programs.

Macro in c

A **macro** is a label defined in the source code that is replaced by its value by the preprocessor before compilation. Macros are initialized with the #define preprocessor command and can be undefined with the #undef command.

There are two types of macros: object-like macros and function-like macros.

1.Object-Like Macros

These macros are replaced by their value in the source code before compilation. Their primary purpose is to define constants to be used in the code.

```
#include <stdio.h>

#define PI 3.1416
```

```
int main() {
  float radius = 3;
  float area;
  area = PI * radius * radius;
  printf("Area is: %f",area);
  return 0;
}
```

```
Output:-Area is: 28.274401
```

2. **Function-Like Macros**

These macros behave like functions, in that they take arguments that are used in the replaced code. Note that in defining a function-like macro, there cannot be a space between the macro name and the opening parenthesis.

Example

AREA is defined as a function-like macro. Note that other macros can be used in defining a subsequent macro. The inner macro is replaced by its value before the outer macro is replaced.

```
#include <stdio.h>
#define PI 3.1416
#define AREA(r) r * r * PI
```

```
int main() {
  float radius = 5;
  float result;
  result = AREA(radius);
  printf("Area is: %f",result);
  return 0;
}
```

Output:- Area is: 78.540001

3. Chain Macros

Macros inside macros are termed chain macros. In chain macros first of all parent macro is expanded then the child macro is expanded.

Example:-

```
#include <stdio.h>
#define INSTAGRAM FOLLOWERS
#define FOLLOWERS 138
int main()
{    printf("PUMIS have %dK"
      " followers on Instagram",
                INSTAGRAM);
        return 0;
}
```

4. Multi-Line Macros

An object-like macro could have a multi-line. So to create a multi-line macro you have to use backslash-newline.

Example:-

```
#include <stdio.h>
#define ELE 1, \
                2, \
                3
int main()
{     int arr[] = { ELE };
      printf("Elements of Array are:\n");
      for (int i = 0; i < 3; i++) {
            printf("%d ", arr[i]);
      }
      return 0;
}
```

Conditional Compilation

- Conditional compilation **allows certain sections of code to be included or excluded during compilation*** based on the values of predefined constants or macro definitions.
- This feature is typically used to create different versions of a program for different operating systems or hardware architectures.

- **For example:** if a program needs to use different system calls on Windows and Linux, the code can use conditional compilation to include the appropriate system calls for each platform.

Syntax of Conditional Compilation

The syntax of conditional compilation in C involves the use of preprocessor directives, which are commands that are executed by the C preprocessor before the code is compiled.

The most commonly used preprocessor directives for conditional compilation are:

- **#ifdef:** tests whether a macro is defined or not, and includes the code that follows if the macro is defined.
- **#ifndef:** tests whether a macro is not defined, and includes the code that follows if the macro is not defined.
- **#elif:** specifies an alternative condition to test if the previous condition in an #ifdef or #ifndef block evaluates to false.
- **#else:** specifies the code to be included if the previous condition in an #ifdef or #ifndef block evaluates to false.
- **#endif:** marks the end of a conditional compilation block.

```
#ifdef MACRO

   code to be included if MACRO is defined

#elif ANOTHER_MACRO

   code to be included if ANOTHER_MACRO is defined

#else

   code to be included if none of the above macros are defined
```

#endif

Types of Conditional Compilation

There are two types of conditional compilation in C:

#if-#endif Directives:

- The #if-#endif directives allow you to **test a constant expression**, and include or exclude code based on the result of the test.
- The #if-#endif block is similar to an if statement in C, but it is **evaluated at compile-time rather than run-time.**

#ifdef macro

code to be included if macro is defined

#endif

#ifndef macro

code to be included if macro is not defined

#endif

#ifdef-#endif and #ifndef-#endif Directives:

- The #ifdef-#endif and #ifndef-#endif directives allow you to **test whether a particular macro is defined or not**, and include or exclude code based on the result of the test.
- The #ifdef-#endif block includes the code if the macro is defined, while the #ifndef-#endif block includes the code if the macro is not defined.

Example of Conditional Compilation

```
#include <stdio.h>

#define LINUX 1

#define WINDOWS 0

int main() {

   #ifdef LINUX

      printf("This code is for Linux\n");

   #elif WINDOWS

      printf("This code is for Windows\n");

   #else

      printf("This code is for other platforms\n");

   #endif

   return 0;

}
```

Explanation:

- The code inside the #ifdef LINUX block will be compiled if the LINUX macro is defined, while the code inside the #elif WINDOWS block will be compiled if the WINDOWS macro is defined.
- If neither macro is defined, the code inside the #else block will be compiled.

Pragma Directives: The pragma directive is used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole.

- Pragma directives are included in the C program to take effect.
- The effect of pragma will be applied from the point where it is included to the end of the compilation unit or until another pragma changes its status.
- A **#pragma** directive is an instruction to the compiler and is usually ignored during preprocessing

1. #pragma startup and #pragma exit

These directives help us to specify the functions that are needed to run before the program starts (before the control passes to the main()) and just before the program exits (just before the control returns from the main()).

Example:-

#pragma startup function_name

```
#pragma exit function_name

#include <stdio.h>

void func1();

void func2();

#pragma startup func1

#pragma exit func2

void func1() { printf("Inside func1()\n"); }

void func2() { printf("Inside func2()\n"); }

int main()

{       printf("Inside main()\n");

        return 0;

}
```

Output:-

```
Inside func1()

Inside main()

Inside func2()
```

2. #pragma warn Directive

This directive is used to hide the warning messages which are displayed during compilation. This may be useful for us hen we have a large program and we want to solve all the errors before looking on warnings then by using it we can focus on errors by hiding all warnings. we can again let the warnings be visible by making slight changes in syntax.

Syntax

#pragma warn +xxx (To show the warning)

#pragma warn -xxx (To hide the warning)

#pragma warn .xxx (To toggle between hide and show)

- **#pragma warn -rvl**: This directive hides those warning which are raised when a function which is supposed to return a value does not return a value.
- **#pragma warn -par**: This directive hides those warning which are raised when a function does not uses the parameters passed to it.
- **#pragma warn -rch**: This directive hides those warning which are raised when a code is unreachable. For example: any code written after the return statement in a function is unreachable.

Example:-

```
#include<stdio.h>

#pragma warn -rvl /* return value */

#pragma warn -par /* parameter never used */

#pragma warn -rch /*unreachable code */


int show(int x)

{     printf("hiiiclass");
}
int main()
{
        show(10);
        return 0;
}
```

```
Output: hiiiclass
```

3.#pragma GCC poison

This directive is supported by the GCC compiler and is used to remove an identifier completely from the program. If we want to block an identifier then we can use the **#pragma GCC poison** directive.

Syntax

#pragma GCC poison *identifier*

Example:-

```
#include <stdio.h>
#pragma GCC poison printf
int main()
{
        int a = 10;
        if (a == 10) {
                printf("hellllo");
        }else
        printf("bye");
  return 0;
}
```

Output:-

```
prog.c: In function 'main':
prog.c:14:9: error: attempt to use poisoned "printf"
         printf("hellllo");
         ^
prog.c:17:9: error: attempt to use poisoned "printf"
         printf("bye");
         ^
```