Procedural Programming-II

UNIT-III

Preprocessor-

Compiler converts textual form of c program into an executable.

There are four phases for a C program to become an executable.

Preprocessing

Compilation

Assembly

Linking

C preprocessor comes under section before the actual compilation process.

C preprocessor is not a part of the c compiler.

It is a text substitution tool.

All preprocessor commands begin with hash symbol (#).

Examples: (command)

#define

#include

#include directives

The #include directive causes the preprocessor to fetch the contents of some other file to be included in the present file.

This file may in turn #include some other file(s) which may in turn do the same.

Most commonly the # included files have ”.h” extension, indicating that they are header files.

In C programming there are two common formats for #include:

1) #include<myfile.h> //the angle brackets say to look in the standard system directories.

2) #include”myfile.h” //the quotation marks say to look in the current directory.

#define directives

The #define directive is used to “define” preprocessor “variables”.

The #define preprocessor directive can be used to globally replace a word with a number.

It acts as if an editor did a global search-and-replace edit of the file.

Example- #define PI 3.14

```
#include<stdio.h>
#include<conio.h>
#define PI 3.14
void main()
{
float A;
int r;
clrscr();
printf("Please enter the value of radius");
scanf("%d",&r);
A=3.14*r*r;
printf("the area of circle %f",A);
getch();
}
```

Macros using #define

We can also create macros using #define

Macros operate much like functions, but because they are expanded in place are generally faster.

Macros

Macros in C are powerful tools that allow developers to define reusable code snippets.

In C programming, a macro is a preprocessor directive defined using #define that substitutes a code snippet, expression, or value with its corresponding identifier before compilation.

Purpose:

Macros enhance code reusability, readability, and allow for code abstraction, optimization, and customization.

Parameterized macros

Macros can have parameter or arguments just like a function

Definition:

Macros are defined using the #define preprocessor directive.

The syntax is #define MACRO_NAME MACRO_VALUE.

MACRO_NAME is the identifier you choose for the macro.

MACRO_VALUE is the code snippet, expression, or value that will replace the macro name.

Syntax-

```
#define macro_name (p1, p2, p3,…..)
```

Example-

```
#include<stdio.h>
#include<conio.h>
#define sqr(x)  (x*x)
Void main()
{
int result;
clrscr();
result=sqr(5);
printf("square value:%d",result);
getch();
}
```

Nested Macros-

In C, nested macros allow you to define macros within other macros, enabling complex code generation and reusability, where the preprocessor expands inner macros before expanding the outer ones.

Example-

Example-

```
#include<stdio.h>
#include<conio.h>
#define sqr(x)  (x*x)  //expansion
#define cube(x)  (sqr(x)*x) //expantion  (macro template is a macro name)
Void main()
{
int svalue, cvalue;
clrscr();
svalue=sqr(5);
cvalue=cube(5);
printf(“square value:%d”,svalue);
printf(“cube value:%d”,cvalue);
getch();
}
```

Macros versus function-

Macros:

Preprocessing: Macros are processed by the preprocessor before compilation, meaning they are replaced by their definitions throughout the code.

Type Checking: Macros do not perform type checking.

Debugging: Debugging code with macros can be challenging because the preprocessor substitutes the macro definition directly, making it difficult to trace the code's execution.

Overloading: Macros cannot be overloaded.

Speed: Macros generally execute faster than functions because there's no function call overhead.

Code Size: Macros can lead to increased code size as their definitions are expanded inline.

Use Cases: Macros are often used for defining constants, simple code substitutions, and for performance-critical sections of code where type safety is not a primary concern.

Example: #define PI 3.14159

Functions:

Compilation:

Functions are compiled as part of the program, meaning they are executed during runtime.

Type Checking:

Functions perform type checking, ensuring that arguments passed to them are of the correct type.

Debugging:

Debugging code with functions is easier because the compiler can track the function's execution and provide more detailed information.

Overloading:

Functions can be overloaded, allowing multiple functions with the same name but different parameters.

Speed:

Functions generally execute slower than macros because of the function call overhead.

Code Size:

Functions generally do not increase code size as much as macros.

Use Cases:

Functions are used for implementing complex logic, reusable code blocks, and when type safety is crucial.

Example:

```
int add(int a, int b) { return a + b; }
```

File handling-

A file is a place on the disk where group of related data is stored

File handling in C allows programs to interact with files on the file system, enabling operations like creating, opening, reading, writing, and closing files using functions like fopen(), fprintf(), fscanf(), fputc(), fgetc(), and fclose().

Types of file-

1) Text files

2) binary files

File Modes:

Different modes are used to open files, such as:

"r": Read mode (opens for reading).

"w": Write mode (opens for writing, overwrites existing content).

"a": Append mode (opens for writing, adds to the end of the file).

"r+": Read and write mode (opens for both, allows overwriting).

"w+": Read and write mode (opens for both, overwrites existing content).

"a+": Read and write mode (opens for both, adds to the end of the file).

Key Functions- (I/O operations):

fopen(filename, mode): Opens a file and returns a file pointer or NULL if an error occurs.

fprintf(file_pointer, format, ...): Writes formatted output to a file.

fscanf(file_pointer, format, ...): Reads formatted input from a file.

fputc(character, file_pointer): Writes a single character to a file.

fgetc(file_pointer): Reads a single character from a file.

fclose(file_pointer): Closes a file.

fseek(file_pointer, offset, origin): Moves the file pointer to a specific position within the file.

ftell(file_pointer): Returns the current position of the file pointer.

rewind(file_pointer): Resets the file pointer to the beginning of the file.

Syntax-

ptr=fopen(“fileopen”,”mode”)

Example-

Ptr=fopen(“abc.txt”,”r”);