Infosys®

POWERED BY INTELLECT
DRIVEN BY VALUES

C Programming

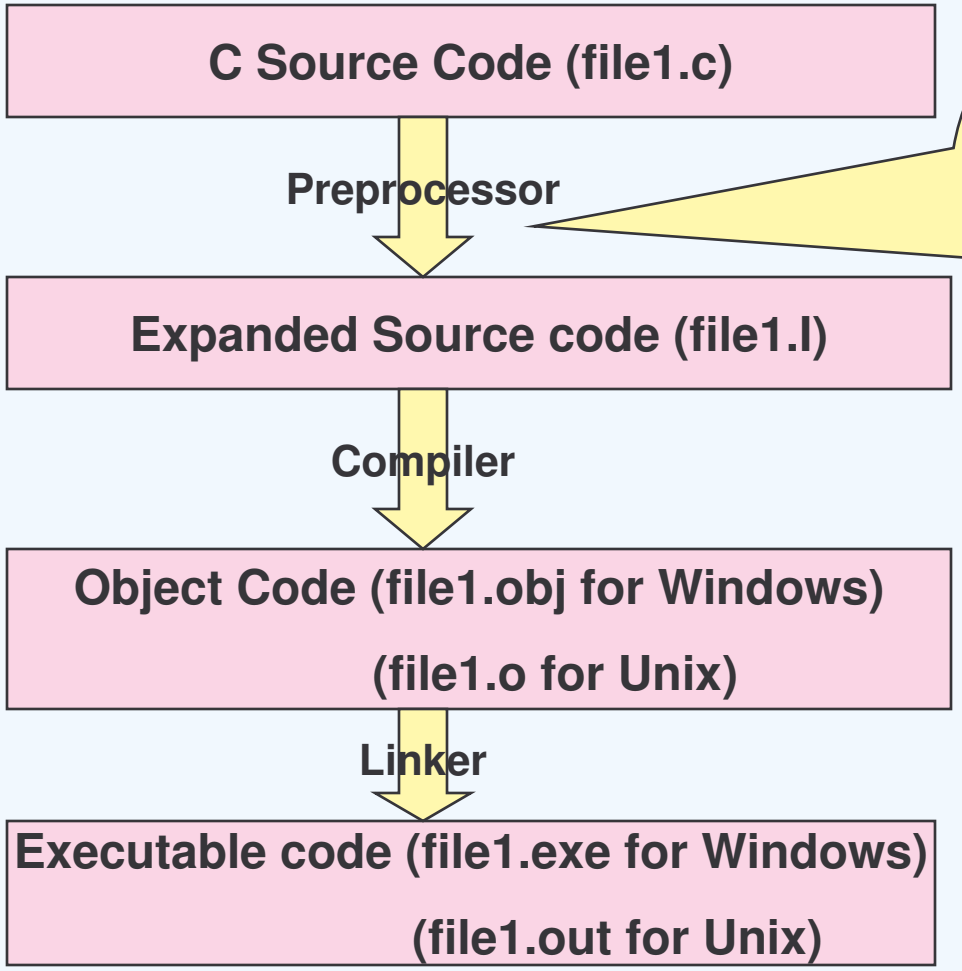
The C PreProcessor

Education and Research

Unit Plan

- Program Life Cycle.
- The C Preprocessor
- The C Preprocessor features
  - Macro expansion
  - File inclusion
  - Conditional Compilation.

Program Life Cycle.



The Preprocessor takes care of

1. Macro expansion
2. Header file inclusion
3. Conditional compilation

The C Preprocessor

- The C Preprocessor is a program that processes the source code before it is passed to the compiler
- The Processor offers several features called as preprocessor directives and each of these preprocessor directives begin with a # symbol.
- The preprocessor directives are generally placed at the beginning of the program , before main(), or before the beginning of a particular function.
- The directives provided by the C Preprocessor are
  - Directives for macro expansion.
  - Directives for File inclusion.
  - Directives for conditional compilation.

Macro Expansions

```
        /*file1.c*/
# define PI 3.1415
main()
{
   float r, area;
   printf("\nEnter the radius of the circle:- ");
   scanf("%f",&r);
   area=PI * r * r;
   printf("\n Area of the circle = %f",area);
}
```


```
        /*file1.i*/
# define PI 3.1415
main()
{
   float r, area;
   printf("\nEnter the radius of the circle:- ");
   scanf("%f",&r);
   area=3.1415 * r * r;
   printf("\n Area of the circle = %f",area);
}
```

- Instead of writing 3.1415 in the expression, we can write it in the form of a preprocessor macro PI.
- The macro PI is defined before main() using the preprocessor directive as, **# define PI 3.1415.** This statement is called as macro definition.
- **PI** is the macro template and **3.1415** is the corresponding macro expansion.
- The source file is given to the preprocessor where every occurrence of **PI** is replaced with **3.1415**.

Macro Expansions (Contd…)

- Why Macro's if the same can be achieved through variables?
  - Compiler generates efficient code for constants as compared to variables.
  - Treating a constant as a variable is principally not correct.
  - A variable may get altered somewhere in the program hence the it no longer remains constant.

Macro Expansions (Contd...)

- Advantages of Macro Expansion
  - The Programs become more readable.
  - If the macro expansion has to be changed, we need to change it only in the macro definition as the preprocessor will replace all occurrences of macro template with the corresponding macro expansion.

Macro Expansions (Contd…)

- Macros with Arguments.
  - Macros can have arguments, just as functions can.
  - Macros with arguments are used to replace small functions.

```
# define AREA(x) (3.14 * x * x)
void main()
{
   float r, area;
   printf("\nEnter the radius of the circle:-  ");
   scanf("%f",&r);
   area=AREA(r);
   printf("\n Area of the circle = %f",area);
}
```

Preprocessor

```
# define AREA(x) (3.14 * x * x)
void main()
{
   float r, area;
   printf("\nEnter the radius of the circle:-  ");
   scanf("%f",&r);
   area= 3.14 * r * r;
   printf("\n Area of the circle = %f",area);
}
```

- In the Program above wherever the preprocessor finds the phrase AREA(x) it expands it into the statement (3.14 * x * x)

Macro Expansions (Contd…)

- Precautions while writing macros with arguments.
  - Never Leave a blank space between the macro template and its argument while defining a macro.
  - The Entire macro template should be enclosed within parenthesis.

Macro Expansions (Contd...)

- Precautions while writing macros with arguments.
  - Avoid using expressions with side effects as arguments.
  - Avoid circular definitions in which the symbol being #defined appears in its own definition.

File Inclusion

- There are some functions that are frequently required in a program. For example function printf() & scanf().
- The frequently required functions prototypes can be stored in a file and that file can be included in every program.
- Such files are called as header files and have a extension as **.h**
- The prototypes of all related library functions are stored in a header file.

File Inclusion (Contd…)

- The preprocessor directive to include a header file is # include
- There are 2 ways to write # include statements
  - # include “abc.h” (The preprocessor would look for the file abc.h in the current directory and in the list of directories specified in the include path)
  - # include <abc.h> (The preprocessor would look for the file abc.h in the list of directories specified in the include path)

Conditional Compilation

- Under certain circumstances we expect the compiler to skip over a part of a source code.
- This can be done by the #ifdef and #endif preprocessor directives.
- The General form of #ifdef and #endif preprocessor directives is as follows.
  - #ifdef macroname

    statement1;

    statement2;

    #endif

    If macroname has been #defined, then the block of code between #ifdef and #endif will get compiled.

Conditional Compilation (Contd…)

- Situations when we need to go for conditional compilation
  - To “comment out” obsolete lines of code.
    - We might need to comment out some code which already has some comments in it . In that case we cannot use /* */ for commenting it as it might lead to nesting of comments.
  - To make programs portable.
    - We might need to write a program which should run on different platforms. In that case certain piece of code will be platform dependent and it should be placed between a #ifdef and #endif directives.

Summary

- Topics covered in this unit are
  - Steps involved in generating a executable from a .c file.
  - The C PreProcessor.
  - Macro Expansions.
  - Need for Macros.
  - Precautions to be taken while writing macros.
  - File Inclusion, its need, its advantages.
  - Conditional compilation, its need, its advantages.

Thank You!