"Advanced C Programming"

Preprocessor

Introduction

Definition and Features:

Pre-processing is conceptually a separate first step in the compilation. It performs the following actions before the compilation process:

- Inclusion of other files
- Definition of symbolic constants and macros
- Conditional compilation of program code
- Conditional execution of preprocessor directives

Operators: Two operators are available in standard C

- **#** - Used to replace the text token to a string surrounded by quotes
- **##** - Used to **concatenate** two tokens (Token Pasting)

File Inclusion (#include)

The #include Preprocessor Directive

- Used to include the copy of a specified file in place of the directive

Syntax:

Type 1: #include <filename>

- **It searches standard library** for the file
- Use for standard library files

Type 2: #include "filename"

- First, **It searches the current directory** for the file, if not found, then it searches the standard library
- Used for user-defined files

File Inclusion (#include)…

The #include Preprocessor Directive

- **Uses**
  - Loading header files
    - **#include<stdio.h>**
  - Programs with **multiple source files** to be compiled together
  - **Header file** - has common declarations and definitions (Structures, Unions, function prototypes)

Symbolic Constants (#define)

The #define Preprocessor Directive

Preprocessor directive used to create **symbolic constants** and **macros**. When program compiled, all occurrences of symbolic constant replaced with replacement text. (Also called as **Object-Like Macros**)

Syntax:

#define identifier replacement-text

Example:

#define PI 3.14159

- Every thing written right side of the identifier will be replaced.
- **Cannot redefine** symbolic constants with more **`#define`** statements

Macros (#define)

Macros: The #define Preprocessor Directive

- A macro is **a fragment of code** which has been **given a name**. Whenever the name is used, it is replaced by the contents of the macro. **(GNU C Definition)**
- By convention, macro names are written in uppercase to increase the readability. People can differentiate between a normal variable and a symbolic constant.

Types of Macros:

a) Object like macros (Symbolic Constants)

b) Function like macros

a) Object like Macros

Object like macros – Macros without arguments are treated like Symbolic Constants and called as object like macros

Examples:

```
#define BUFFER_SIZE 1024
foo = (char *) malloc (BUFFER_SIZE);
```

Macro on multiple lines: (Use \ as line continuation)

Symbolic Constants (#define)

The #define Preprocessor Directive

```
#include<stdio.h>
#define UPPER 25

void main( )
{
        int i ;
        for ( i = 1 ; i <= UPPER ; i++ )
        printf( "%d\n",i);
}
```

```
#include <stdio.h>
#define PI 3.1415
int main(){
    int radius;
    float area;
    printf("Enter the radius: ");
    scanf("%d",&radius);
    area=PI*radius*radius;
    printf("Area=%.2f",area);
    return 0;
}
```

b) Function like Macros

Function like macros – To define a function like macro, use the same directive "#define", but put a pair of parentheses immediately after the macro name.

Examples:

Before preprocessing

```
#define PI 3.14159
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
area = CIRCLE_AREA(4);
```

After preprocessing

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

A function like macro is only expanded if its name appears with a pair of parentheses after it.

b) Function like Macros…

- Without parentheses – Incorrect expansion

Before preprocessing

```
#define CIRCLE_AREA( x )  PI * ( x ) * ( x )
area = CIRCLE_AREA( c + 2 );
```

After preprocessing

```
area = 3.14159 * c + 2 * c + 2;
```

- Multiple arguments

Before preprocessing

```
#define RECTANGLE_AREA( x, y ) ( ( x ) * ( y ) )
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

After preprocessing

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

Macros

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
   printf("Max between 10 and 20 is %d\n", MAX(10, 20));
   return 0;
}
```

Macros

```
#define MUL(X,Y) X*Y
void main()
{
printf("value %d",MUL(3,4));
printf("value %d",MUL(3+2,4+2));
}
```

Macros - Exercise

- **Write a program to compute the area of a rectangle using macros**

Function .vs. Macro

No	Macro	Function
1	Macro is **Preprocessed**	Function is **Compiled**
2	**No Type Checking**	**Type Checking** is Done
3	**Code** Length **Increases**	**Code** Length remains **Same**
4	Use of macro can lead to **side effect**	No **side Effect**
-		
5	Speed of Execution is **Faster**	Speed of Execution is **Slower**
6	Before Compilation macro name is replaced by macro value	During function call , Transfer of Control takes place
7	Useful where small code appears many time	Useful where large code appears many time
8	Generally Macros do not extend beyond one line	Function can be of any number of lines
9	Macro does not Check **Compile Errors**	Function Checks **Compile Errors**

Macros - #undef

- #undef
  - Undefines a symbolic constant or macro, which can later be redefined

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
   printf("Max between 10 and 9 is %d\n", MAX(10, 20));
   return 0;
}
```

Conditional Compilation

We can, if we want, have the compiler skip over part of a source code by inserting the preprocessing commands **#ifdef and #endif,**which have the general form:

```
#ifdef macroname
    statement 1 ;
    statement 2 ;
    statement 3 ;
#endif
```

If macroname has been #defined, the block of code will be processed as usual; otherwise not.

#ifdef Directives

```
#include <stdio.h>
#define RAJU 100
int main()
{
   #ifdef RAJU
   printf("RAJU is defined. So, this line will be added in " \
         "this C file\n");
   #else
   printf("RAJU is not defined\n");
   #endif
   return 0;
}
```

#ifndef Directives

- #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.

#ifndef Directives

```
#include <stdio.h>
#define RAJU 100
int main()
{
   #ifndef SELVA
   {
      printf("SELVA is not defined. So, now we are going to define
     here\n");
      #define SELVA 300
   }
   #else
   printf("SELVA is already defined in the program");

   #endif
   return 0;

}
```

#undef Directives

```
#include <stdio.h>
#define height 100
void main()
{
   printf("First defined value for height    :
    %d\n",height);
   #undef height          // undefining variable
   #define height 600     // redefining the same for new
    value
   printf("value of height after undef
    redefine:%d",height);
}
```

Conditional Compilation

- **Conditional compilation**
  - Control preprocessor directives and compilation
  - **Note:** Cast expressions, **sizeof,** enumeration constants cannot be evaluated
- Structure is similar to **"if"** control structure

```
#if !defined( NULL )
        #define NULL 0
#endif
```

  - Determines if symbolic constant **NULL** defined
    - If **NULL** is defined, **defined(NULL)** evaluates to **1**
    - If **NULL** not defined, defines **NULL** as **0**
  - Every **#if** ends with **#endif**
  - **#ifdef** is the short form for **#if defined(***name***)**
  - **#ifndef** is the short from for **#if !defined(***name***)**

Conditional Compilation…

- **Other statements**

  #elif - equivalent of else if in an if structure

  #else - equivalent of else in an if structure

- **"Comment out" code**
  - Cannot use /* … */
  - **Use**

```
#if 0
    code commented out
#endif
```

  To enable the code, change 0 to 1

Conditional Compilation…

- **Debugging**

```
#define DEBUG 1
#ifdef DEBUG
   printf("Variable x = %d", x);
#endif
```

  - Defining DEBUG enables code
  - After code corrected, remove #define statement
  - Debugging statements are now ignored

Operator - Concatenation

- **## - It concatenates two tokens**

Example:

```
#define TOKENCONCAT( x, y )  x ## y

TOKENCONCAT( O, K )   becomes OK
```

```
#include<stdio.h>
#define CAT(a, b) a##b
void main() {
   printf("%d", CAT(10, 20));
}
```

Thank You