PES UNIVERSITY

ELECTRONIC CITY CAMPUS

Operators, Preprocessors

Programming Using C

UE19BC151

2020 JAN – MAY

Objectives

Unit Objectives are –

- To learn the function and concept of pre-processors in any programming language
- To know the advantage of macros which helps in the execution speed of program fragment
- To identify the benefits of inline functions in programming language.

Operators

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows −

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Operators

Assume A = 60 and B = 13 in binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

Operators

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

Operators

Logical Operators

Operator	Description
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.

Bitwise Vs Logical Operators

- A Bitwise And operator is represented as '&' and a logical operator is represented as '&&'.
- Following are some basic differences between the two operators.

a) The logical and operator '&&' expects its operands to be boolean expressions (either 1 or 0) and returns a boolean value. The bitwise and operator '&' works on Integral (short, int, unsigned, char, bool, unsigned char, long) values and return Integral value.

Eg-

```
int main()
{
   int x = 3;  //...0011
   int y = 7;  //...0111
if (y > 1 && y > x) // A typical use of '&&'
    printf("y is greater than 1 AND y\n");
// A typical use of '&'
   int z = x & y;   // 0011
   printf ("z = %d", z);
   return 0;
}
```

Bitwise Vs Logical Operators

b) If an integral value is used as an operand for '&&' which is supposed to work on boolean values, following rule is used in C. A zero is considered as false and non-zero is considered as true.

For example in the following program x and y are considered as 1.

/ Example that uses non-boolean expression as operand for '&&'

```
int main()
{
  int x = 2, y = 5;
  printf("%d", x&&y);
  return 0;
}
```

Bitwise Vs Logical Operators

c) The '&&' operator doesn't evaluate second operand if first operand becomes false. Similarly '||' doesn't evaluate second operand when first operand becomes true. The bitwise '&' and '|' operators always evaluate their operands.

Eg –

```
int main()
{
  int x = 0;
    // 'Geeks in &&' is NOT printed because x is 0
  printf("%d\n", (x && printf("Geeks in && ")) );
    // 'Geeks in &' is  printed
  printf("%d\n", (x  & printf("Geeks in & ")) );
    return 0;
}
```

The same differences are there between logical OR '||' and bitwise OR '|'.

Operators

sizeof() Operator

This is a function like operator that returns the occupied size of any variable, object, constant etc, even this operator returns the size of any data type, literal etc.

eg –

```
#include <stdio.h>
int main()
{
    int x=10;
    printf("size of x    : %d\n",sizeof(x));
    printf("size of 10   : %d\n",sizeof(10));
    printf("size of int  : %d\n",sizeof(int));
    printf("size of char : %d\n",sizeof(char));
    printf("size of float: %d\n",sizeof(float));
    return 0;
}
```

Preprocessors

- 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.

- 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.

Preprocessors

Sr.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.

Preprocessors

9	**#endif** Ends preprocessor conditional.
10	**#error** Prints error message on stderr.
11	**#pragma** Issues special commands to the compiler, using a standardized method.

1) #define MAX_ARRAY_LENGTH 20

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use #define for constants to increase readability.

2) **#include <stdio.h>**
#include "myheader.h"

These directives tell the CPP to get stdio.h from System Libraries and add the text to the current source file. The next line tells CPP to get myheader.h from the local directory and add the content to the current source file.

Preprocessors

3) **#undef FILE_SIZE**
#define FILE_SIZE 42

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

4) **#ifndef MESSAGE**
#define MESSAGE "You wish!“
#endif

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

5) **#ifdef DEBUG**
/*Your debugging statements here */
#endif

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the -DDEBUG flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Predefined Macros

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

S.no	Macro & Description
1	**__DATE__** The current date as a character literal in "MMM DDYYYY" format.
2	**__TIME__** The current time as a character literal in "HH:MM:SS" format.
3	**__FILE__** This contains the current filename as a string literal.
4	**__LINE__** This contains the current line number as a decimal constant.
5	**__STDC__** Defined as 1 when the compiler complies with the ANSI standard.

Predefined Macros

Eg –

```
#include <stdio.h>
int main() {
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );
}
```

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Conditional Compilation

The C preprocessor provides a series of directives for conditional compilation:

- #if, #elif, #else, #endif
- #ifdef, #ifndef

They are used for 3 main purposes:

- including optional debug code,
- enclosing non-portable code,
- protecting header-files from multiple inclusion.

Preprocessors

Multiline Macros

Macro definitions normally extend to the end of the line. But long macros can be split over several lines by ending the line with \.

```
#define ERROR(condition, message) \
    if (condition) printf(message)
```

Inline Functions

- An inline function is a function that in expanded in line when it is invoked.
- Inline function must be defined before they are called.
- Inline keyword sends a request, not a command to the compiler

Syntax

```
Inline function-header
{
    function body
}
```

- The inline function is just a code replacement instead of the function call.
- There is a need of stack storage and other special mechanism for function call and return.
- The stack storage is used to store the return value and return address for function call and return process. And while passing the parameter the stack storage needed to store the parameter values, and from the stack area the values moved to data area.

Inline Functions

- In this example, all the declarations and definitions use inline but not extern:

```
// a declaration mentioning inline
    inline int max(int a, int b);
// a definition mentioning inline
    inline int max(int a, int b)
  { return a > b ? a : b; }
```

Inline Functions

```
#include <stdio.h>
void inline func1(int a, int b) {
    printf ("a=%d and b=%d\n", a, b);
}
int inline square(int x) {
    return x*x;
}
int main() {
    int tmp;
    func1(2,4);
    tmp = square(5);
    printf("square value =%d\n", tmp);
    return 0;
}
```

Inline Functions

Inline functions provides following advantages over macros.

- Since they are functions so type of arguments is checked by the compiler whether they are correct or not.
- There is no risk if called multiple times. But there is risk in macros which can be dangerous when the argument is an expression.
- They can include multiple lines of code without trailing backlashes.
- Inline functions have their own scope for variables and they can return a value.
- Debugging code is easy in case of Inline functions as compared to macros.

Command Line Arguments

- It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments.**
- The command line arguments are handled using **main()** function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program.

Eg –

```
#include <stdio.h>
int main( int argc, char *argv[] )  {
if( argc == 2 ) {
   printf("The argument supplied is %s\n", argv[1]);   }
else if( argc > 2 ) {
   printf("Too many arguments supplied.\n");   }
else {     printf("One argument expected.\n");
}}
```

Command Line Arguments

- It should be noted that argv[0] holds the name of the program itself and argv[1] is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then argc is set at 2.

Environmental Variables

- Most of the compilers also support a **third declaration** of main that accepts third argument. The third argument stores all **environment variables.**

Eg –

```
#include <stdio.h>
  // Most of the C compilers support a third parameter to main which
// store all envorinment variables
int main(int argc, char *argv[], char * envp[])
{
   int i;
   for (i = 0; envp[i] != NULL; i++)
      printf("\n%s", envp[i]);
   getchar();
   return 0;
}
```

Environmental Variables

ALLUSERSPROFILE=C:\ProgramData
CommonProgramFiles=C:\Program Files\Common Files
HOMEDRIVE=C:
NUMBER_OF_PROCESSORS=2
OS=Windows_NT
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 42 Stepping 7, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=2a07
ProgramData=C:\ProgramData
ProgramFiles=C:\Program Files
PUBLIC=C:\Users\Public
SESSIONNAME=Console
SystemDrive=C:
SystemRoot=C:\Windows
WATCOM=C:\watcom
windir=C:\Windows

Recursion

- Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

Eg –

```
void recursion() {
  recursion(); /* function calls itself */
}
int main() {
  recursion();
}
```

- The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursion

- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Number Factorial

```
#include <stdio.h>
unsigned long long int factorial(unsigned int i) {
  if(i <= 1) {
    return 1;
  }
  return i * factorial(i - 1);
}
int  main() {
  int i = 3;
  printf("Factorial of %d is %d\n", i, factorial(i));
  return 0;
}
```

Fibonacci Series

```
#include <stdio.h>
int fibonacci(int i) {
  if(i == 0) {
    return 0;
  }
   if(i == 1) {
    return 1;
  }
  return fibonacci(i-1) + fibonacci(i-2);}
int  main() {
  int i;
  for (i = 0; i < 5; i++) {
  printf("%d\t\n", fibonacci(i));
  }
   return 0;}
```

Recursion Vs Iteration

- Recursion is when a method in a program repeatedly calls itself whereas, iteration is when a set of instructions in a program are repeatedly executed.
- A recursive method contains set of instructions, statement calling itself, and a termination condition whereas iteration statements contain initialization, increment, condition, set of instruction within a loop and a control variable.
- A conditional statement decides the termination of recursion and control variable's value decide the termination of the iteration statement.
- If the method does not lead to the termination condition it enters to infinite recursion. On the other hand, if the control variable never leads to the termination value the iteration statement iterates infinitely.
- Infinite recursion can lead to system crash whereas, infinite iteration consumes CPU cycles.
- Recursion is always applied to method whereas, iteration is applied to set of instruction.

Recursion Vs Iteration

- Variables created during recursion are stored on stack whereas, iteration doesn't require a stack.
- Recursion causes the overhead of repeated function calling whereas, iteration does not have a function calling overhead.
- Due to function calling overhead execution of recursion is slower whereas, execution of iteration is faster.
- Recursion reduces the size of code whereas, iterations make a code longer.

Outcomes

The students will be able to –

- Declare and Define preprocessor based on the logic and kind of functions used in the application
- Apply the macros in various applications and evaluate the speed of execution of the application
- Differentiate between inline functions and normal function and compare the efficiency of the program.