**ASSIGNMENT-1**

1. **BIT WISE OPERATORS**

**Bit Wise Operators:** C supports bit wise operators for manipulation of data at bit level. They are not applied to float (or) double.

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise Exclusive OR (Ex-OR) |
| ~ | Bitwise NOT (one's complement) |
| << | left shift |
| >> | right shift |

**Bitwise AND (&):** Bitwise AND operator takes 2 bit patterns and perform AND (multiplication) operations with it.

|  |
| --- |
| **1st Bit** 1010 |
| **2nd Bit** 1100 |
| **AND** (&) 1000 |

Note: The Bitwise AND will take pair of bits from each position, and if only both the bit is 1, the result on that position will be 1. Bitwise AND is used to Turn-Off bits.

**Bitwise OR (|):** Bitwise OR operator takes 2 bit patterns and perform OR (addition) operations on each pair of corresponding bits.

|  |
| --- |
| **1st Bit** 1010 |
| **2nd Bit** 1100 |
| **OR** (|) 1110 |

Note: The Bitwise OR, will take pair of bits from each position, and if any one of the bit is 1, the result on that position will be 1. Bitwise OR is used to Turn-On bits.

**Bitwise EX-OR (^):** Bitwise EX-OR operator takes 2 bit patterns and perform EX-OR operations on each pair of corresponding bits. The Bitwise XOR will take pair of bits from each position, and if both the bits are different, the result on that position will be 1. If both bits are same, then the result on that position is 0.

|  |
| --- |
| **1st Bit** 1010 |
| **2nd Bit** 1100 |
| **EX-OR** **(^)** 0110 |

Example: C= (20 ^ 5)

Here, the binary equivalent of 20 is (10100)2 and binary equivalent of 5 is (101)2

|  |  |
| --- | --- |
| **1st Bit** | 10100 |
| **2nd Bit** | **00**101 |
| **EX-OR** | **10001** |

**Bitwise NOT (~):** One’s complement operator (Bitwise NOT) is used to convert each “1-bit to 0-bit” and “0-bit to 1-bit”, in the given binary pattern. It is a unary operator i.e. it takes only one operand.

|  |
| --- |
| **1st Bit** 1001 |
| **NOT** 0110 |

### Bitwise Left Shift (<<): The left shift operator will shift the bits towards left for the given number of times.

Example: int C=4<<3;

Step 1: Consider binary equivalent of 4 and represent in group of 8-bits (1 byte).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **0** | **0** | **0** | **0** | 1 | 0 | 0 |

**Step 2:** Now shifting the bits towards **left for 3 bit positions**, we get

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **0** | 1 | 0 | 0 | **0** | **0** | **0** |

Now the result is equivalent to decimal number 32. You can also note that, 0 is added as padding in the vacant positions (highlighted positions).

Note: In left shift operation, the result value is increases. Means left shifting 1 time is equal to multiplying the value by 2. Consider the same example: 4<<3 here left shifting value ‘4’ to ‘3’ positions, means multiplifying ‘4’ with ‘2’ for ‘3’ times. i.e., 4 X (2X2X2) = 32.

C= x \* 2y  Here x=4, y=3 So, C=32.

### Bitwise Right Shift (>>): The Right shift operator will shift the bits towards right for the given number of times.

Example: int C=16>>2;

Step 1: Consider binary equivalent of 16 and represent in group of 8-bits (1 byte) and padding the empty positions with zero.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **0** | **0** | 1 | 0 | 0 | 0 | 0 |

**Step 2:** Now shifting the bits towards **right for 2 bit positions**, we get

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **0** | **0** | **0** | **0** | 1 | 0 | 0 |

Now the result is equivalent to decimal number 4. You can also note that, 0 is added as padding in the vacant positions (highlighted positions).

Note: In Right shift operation, the result value is decreases. Means right shifting 1 time, is equal to dividing the value by 2. Consider the same example:16>>2 here right shifting value ‘16’ to ‘2’ positions, means dividing ‘16’ with ‘2’ for ‘2’ times. i.e., 16/(2X2) = 4.

C= x / 2y  Here x=16, y=2 So, C=4.

**-----------------------------------------------------------------------------------------------------------------------------**

**2.** An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.

#include <stdio.h>  
   
int main()  
{  
   int m, n, c, d, first[10][10], second[10][10], sum[10][10];  
   
   printf("Enter the number of rows and columns of matrix**\n**");  
   scanf("%d%d", &m, &n);  
   printf("Enter the elements of first matrix**\n**");  
   
   for (c = 0; c < m; c++)  
      for (d = 0; d < n; d++)  
         scanf("%d", &first[c][d]);  
   
   printf("Enter the elements of second matrix**\n**");  
   
   for (c = 0; c < m; c++)  
      for (d = 0 ; d < n; d++)  
         scanf("%d", &second[c][d]);  
     
   printf("Sum of entered matrices:-**\n**");  
     
   for (c = 0; c < m; c++) {  
      for (d = 0 ; d < n; d++) {  
         sum[c][d] = first[c][d] + second[c][d];  
         printf("%d**\t**", sum[c][d]);  
      }  
      printf("**\n**");  
   }

**---------------------------------------------------------------------------------------------------------------------------**

**3) String Manipulation functions**

|  |  |  |
| --- | --- | --- |
| **strcpy()** | strcpy(string1, string2) | Copies string2 value into string1 |
| **strncpy()** | strncpy(string1, string2, 5) | Copies first 5 characters string2 into string1 |
| **strlen()** | strlen(string1) | returns total number of characters in string1 |
| **strcat()** | strcat(string1,string2) | Appends string2 to string1 |
| **strncat()** | strncpy(string1, string2, 4) | Appends first 4 characters of string2 to string1 |
| **strcmp()** | strcmp(string1, string2) | Returns 0 if string1 and string2 are the same; less than 0 if string1<string2; greater than 0 if string1>string2 |

**-----------------------------------------------------------------------------------------------------------------------------------------**

4) **Differences between Structure and Union:**

|  |  |  |
| --- | --- | --- |
|  | Structures | Unions |
| 1 | Structure is a Collective name given to different data items (or) elements of different data types. | Union is also a Collection of different data elements of different data types. |
| 2 | The keyword ‘struct’ is used to define the structure. | The keyword ‘union’ is used to define the union. |
| 3 | In structures, each structure member occupies the memory individually. i.e., each member within a structure is assigned unique storage area. | In unions, the memory is depends on largest data type member. i.e., in unions, the total memory allocated is shared by individual members of the union. |
| 4 | In structures, all the member variables can be assigned values at a time. And accessing of members can also be done at a time. | In unions, only one member can be assigned value at a time. And also only one member can be accessed at a time. |
| 5 | In structures, altering (or) modifying the value of a member will not affect the other members in a structure. | In unions, altering the value of any of the member will affect the other member values. |
| 6 | In structures, the address of each member will start at different offset position (or) location. Because, each member can be assigned with unique storage space. | In unions, the address is same for all the members of a union. Every member begins at the same offset position. Bcos, the memory is shared by its members. |
| 7 | The usage of structure is more efficient, when all the members are actively used in the program. | The usage of union is more efficient, when all the members are not required to access at the same time in the program. |
| 8 | Example: struct emp  {  int empno;  char ename[20];  float salary;  } e1; | Example: union book  {  char name[25]; float price; int pages;  } b1; |

**-------------------------------------------------------------------------------------------------------------------------**

**Assignment-2**

**PREPROCESSOR DIRECTIVES:**

Preprocessor is a text (or) code substitution tool. It is the code that processes our source program before it is passed to the compiler. The ‘C’ Preprocessor executes before a program is compiled.

Some of the actions performed by Preprocessor are:

1. Inclusion of External files (including header files)
2. Definition of Symbolic Constants
3. Macro’s
4. Conditional compilation
5. Conditional execution

Note: 1) Preprocessor directives begin with # (Hash Tag) and they appear after comment line section in the program.

2) Preprocessor directives does not end with ‘;’ (Semi-colon).

3) Preprocessor directives are placed in the source program before the main() function.

4) Preprocessor directives allow you to define *macros*, which are brief abbreviations for longer constructs.

**1. #include Preprocessor Directive:** #include directive puts copy of a file in place of directive. It can be used in Two ways.

a) #include <filename>

*Example:* #include<stdio.h>

It is a Standard Input Output header file. It searches the file in pre-designated directories.

b) #include ”filename”

*Example:* #include ”complex.c”

It is a User defined file. It searches the file in Current directory. Normally used for including programmer (or) User defined files.

**2. #define Preprocessor Directive:** #define directive is used to represent (or) create Symbolic constants and Macros.

a) Symbolic constants are the constants represented as symbols, when program is compiled all the occurrences are replaced with constant value.

*Example:* #define PI 3.14

Program: #include<stdio.h>

#define PI 3.14

int main( )

{

printf (“%f”, PI);

}

Result: 3.140000

b) Macros:Macros are the brief abbreviations for longer constructs.

*Example:* #define Macro

Program: #include<stdio.h>

#define min(a, b) ((a)<(b)?(a):(b))

int main( )

{

printf (“The minimum value between 10 and 20 is %d\n”, min(10,20));

}

Result: 10

**3. #undef Preprocessor Directive:** #undef Preprocessor directive is used to undefine the constant (or) macro which is defined by #define Preprocessor directive.

**Syntax:**  #undef Token

/\* Demonstrate define and undefine a constant\*/

#include<stdio.h>

#define PI 3.14

#undef PI

Void main( )

{

printf(“%f”, PI);

}

Result: It gives Error as: ’PI’ undeclared identifier.

**4. #ifdef Preprocessor Directive:** #ifdef Preprocessor directive checks if macro is defined by #define (or) Not. If yes, it executes the code. Otherwise #else code is to be executed.

**Syntax of ifdef:** #ifdef MACRO

//Executable code

#endif

**Syntax of ifdef-else:** #ifdef MACRO

//True code

#else

//False code

#endif

Example 1:

#include<stdio.h>

*#define NOINPUT*

Void main( )

{ Static int a;

*#ifdef NOINPUT*

a=2;

#else

printf (“Enter the value of a:”);

scanf (“%d”, &a);

#endif

printf (“The value of a: %d\n”, a);

}

Result: The value of a: 2 //ifdef is satisfied. So, it executes True code.

Example 2:

#include<stdio.h>

Void main( )

{ Static int a;

*#ifdef NOINPUT*

a=2;

#else

printf (“Enter the value of a:”);

scanf (“%d”, &a);

#endif

printf (“The value of a: %d\n”, a);

}

Result: Enter the value of a: 5

The value of a: 5 //ifdef is not satisfied. So, it executes False code.

**5. #ifndef Preprocessor Directive:** The #ifndef Preprocessor directive checks if macro is not defined by #define. If yes, it executes the code. Otherwise #else code is to be executed.

**Syntax of ifndef:** #ifndef MACRO

//Executable code

#endif

**Syntax of ifndef - else:** #ifndef MACRO

//True code

#else

//False code

#endif

Example:

#include<stdio.h>

Void main( )

{

Static int a;

*#ifndef NOINPUT*

a=2;

#else

a=5;

#endif

printf (“The value of a: %d\n”, a);

}

Result: The value of a: 2

**6. #if & #else Preprocessor Directive:** The #if Preprocessor directive evaluates the Expression (or) Condition. If the condition is true, it executes the code. Otherwise #elseif (or) #else (or) #endif code is executed.

**Syntax of if:** #if Expression

//Executable code

#endif

**Syntax of if-else:** #if Expression

//True code

#else

//False code

#endif

**Syntax of if-elsif:** #if Expression

//True code

#elif Expression

//else if True code

#else

//False code

#endif

**-------------------------------------------------------------------------------------------------------------------------------**

**-------------------------------------------------------------------------------------------------------------------------------**

1. Recursion is a function which is called by itself (or) Recursion is a repetitive process in which a function calls itself. It's a programming technique which facilitates to write simpler, shorter, and clearer code.

For example: main ( )

{

printf (“This is an example for recursion \n”);

main( );

}

When executed this program, the output will be printed infinite times.

**Result:**

This is an example of recursion

This is an example of recursion

This is an example of recursion

…..

…..

….. //printed infinite times

**Advantages of Recursion:**

1) Recursive functions are simpler, clearer, and shorter when compared to iterative statements (while, do while, for, go to).

2) Reduce unnecessary calling of function.

3) Recursive functions focus directly and precisely on the actual problem as compared to their non-recursive counterparts.

**Disadvantages (or) Limitations of Recursion:**

1) Recursive functions are always logical and very difficult to trace. (i.e., Debugging and understanding is difficult).

2) Recursive functions are inherently inefficient as they normally takes a lot of stack space.

3) Recursion uses more processor time.

4) In Recursive functions, we must have if statement somewhere to force the function to return without the recursive call being executed; otherwise the function will never return.

**/\*Write a Recursive function to find factorial of a number \*/**

#include <stdio.h>

Void main ( )

{

int n, R;

**int factorial (int);**

printf (“Enter the value of n:”);

scanf(“%d”, & n);

R= **factorial (n);**

printf (“The Factorial value is: %d”, R);

}

**int factorial (int n)**

{

int p;

if (n==1)

**return (1);**

else

**p= n \* factorial (n-1);**

**return (p);**

}

**Result:** Enter the value of n: 6

\_----------------------------------------------------------------------------------------------------------

3. **FILE POSITIONING FUNCTIONS:** Also called Random Access Functions.

**a) FSEEK ( ):** The fseek( ) function is used to set the file pointer to the specified offset. It is used to write data into the file at desired location (or) Position.

Syntax: fseek (file stream, int offset, int whence)

Here, File stream is Pointer to file i.e., fp.

Offset is Particular position in a file.

Whence is location where Offset starts.

**SEEK-SET =** starts offset from beginning of file.

**SEEK-CUR =** starts offset from current position of the cursor.

**SEEK-END =** starts offset from end of file.

**/\* Program to Demonstrate File Positioning Function fseek( ) \*/**

#include<stdio.h>

int main()

{

FILE \*fp;

fp = fopen(“myfile.txt” , ”w+”);

fputs (“This is C programming”, fp);

fseek (fp, 9, SEEK\_SET);

fputs (“Dennis Ritchie’s ”, fp);

fclose (fp);

}

**b) REWIND ( ):** The rewind( ) function sets the file pointer at the beginning of the stream. It is useful if you must use stream many times.

Syntax:  rewind(fp);

**/\* Program to Demonstrate File Positioning Function rewind( ) \*/**

#include<stdio.h>

void main( )

{

FILE \*fp;

char c;

fp = fopen(“myfile.txt”, “r”);

While ((c = getc(fp) != EOF)

{

printf (“%c”, c);

}

rewind(fp);

While ((c = getc(fp) != EOF)

{

printf(“%c”, c);

}

fclose(fp);

}

**c) FTELL ( ):** The ftell( ) function returns the current file position of the specified stream. We can use ftell( ) function to get the total size of a file after moving file pointer at the end of file.

Syntax:  ftell (fp);

**/\* Program to Demonstrate File Positioning Function ftell( ) \*/**

#include<stdio.h>

void main( )

{

FILE \*fp;

int length;

fp = fopen(“myfile.txt” , ”r”);

fseek (fp, 0, SEEK\_END);

length = ftell(fp);

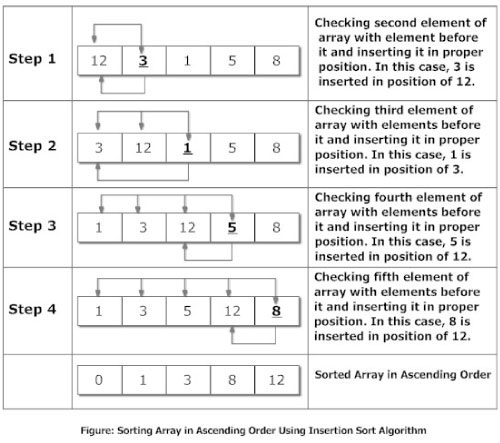
fclose (fp);

printf (“The size of file: %d”, length);

}

------------------------------------------------------------------------------------------------------------

**4.Insertion Sort** is a simple sorting algorithm that works the way we play cards in our hands.



Void main ( )

{

int a[10], i, n, key, j;

printf (“Enter the value of n:” );

scanf (“%d”, &n);

for (i=1; i<=n; i++)

{

printf (“Enter the element of array:” );

scanf (“%d”, &a[i]);

}

for (i=1; i<=n-1; i++) //number of steps

{

a[0]=-9999;

key = a[i+1];

j = i + 1;

while ( j>1 && a[j-1]>key)

{

a[j] = a[j-1];

j = j - 1;

}

a[j]=key;

}

for (i=1; i<=n; i++)

{

printf (“The sorted order is: %d \t”, a[i]);

}

}