**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.

**Relational Operators:** Used to compare two different quantities and depending on their relation, we take certain decisions. The Relational operators are:

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| < | is less than |
| > | is greater than |
| <= | is less than or equal to |
| >= | is greater than or equal to |
| == | is equal to |
| != | is not equal to |

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

1. **MULTIWAY SELECTION STATEMENTS**

**Else if :** It is also called Ladder- if. It contains multiple ‘if’ statements within another ‘if’ statement to check different conditions.The general form of if -else’statement is:

Syntax 1: *if (condition 1)*

*{*

*execute statement(s);*

*} // end of true block of condition 1*

*else if (condition 2)*

*{*

*execute statement(s);*

*} // end of true block of condition 2*

*else*

*{*

*execute statement(s);*

*} // end of false block of condition 2*

**Program to check whether the given number is positive, Negative or Zero\*/**

Void main( )

{

int n ;

printf ( "Enter any number " ) ;

scanf ( "%d ", &n) ;

if ( n > 0 )

{

printf ( "\n The number %d is Positive" , n) ;

}

else if ( n < 0 )

{

printf ( "\n The number %d is Negative" , n) ;

}

else

{

printf (“\n The number %d is Zero" , n) ;

}

}

**) Switch – Case statement:** Switch case statements are a substitute for long ‘if’ statements that compares a variable with several values.

The switch statement is a multi-way branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

Important Points about Switch Case Statements**:**

1) The expression provided in the switch should result in aconstant value. i.e, either integer (or) character constant. Otherwise it would not be valid.  
2) Duplicate Case values are not allowed in switch block.

3) The default statement is optional. Even if the switch case statements do not have a default statement, it would run without any problem. Default statement executes, if any of the case is not satisfied with Switch statement.

4) The break statement is used inside the switch to terminate each executable statement written in particular case. When a break statement is executed, it breaks the switch and comes out. And the flow of control jumps to the next line following the switch block.

5) The break statement is optional. If omitted, execution will continue on into the next case. The flow of control will fall through to subsequent cases until a break is reached.

6) Nesting of switch statements are allowed, which means you can have switch statements inside another switch. However nested switch statements should be avoided as it makes program more complex and less readable.

7) The switch statement allows only integer (or) character constants. It should not allow float or any other constant in to it.

**Syntax:** Switch (expression)

{

Case value1: Executable statement(s) block;

break;

Case value2: Executable statement(s) block;

break;

Case value3: Executable statement(s) block;

break;

…. ….. …..

default: default statement block;

}

**/\*Program to demonstrate Switch- Case statement\*/**

#include <stdio.h>

main( )

{

   int x = 2;

   switch (x)

   {

       case 1: printf("Choice is 1");

                break;

       case 2: printf("Choice is 2");

                 break;

       case 3: printf("Choice is 3");

                break;

       default: printf(" entered Choice other than 1, 2 and 3");

     } //end of switch block

   return 0;

} // end of main

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

**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) Structure is a collection of dissimilar (heterogeneous) data elements (or) Structure is a collective name given to different kind of data items (variables) associated with different data types.

Structures provide the ability to create user defined data types and also to represent real

world data. i.e., Structures can be used to store the real world data like employee, student, person etc...

The syntax (or) declaration of structure:

struct <structure\_tag\_name>

{

data-type member\_name1;

data-type member\_name2;

.

data-type member\_name n;

} <structure\_variable1>, <structure\_variable2>,...;

Accessing of structure elements:

The members of a structure can be accessed by using the ‘.’ (dot) known as the dot operator (or) member operator.

The general form of the statement for accessing a member of a structure is:

<structure\_variable> . <member\_name>;

Example: struct sample

{

int x, y;

float t;

char u;

} sam2 = { 7, 3, 0.0, ‘A’ };



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

**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

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

2) **) FWRITE( ):** This function is used toWriting to a binary file on the disk.

The fwrite ( ) function takes Four arguments i) address of data to be written in disk ii) size of data to be written iii) number of such type of data iv) pointer to the file where you want to write.

Syntax: fwrite (address\_data, size\_of\_variable, number\_of\_elements, pointer\_to\_file);

Example: #include<stdio.h>

int main( )

{

char a[10] = {‘1’,’2’,’3’,’4’,’5’,’6’,’7’,’8’,’9’,’a’};

FILE \*fp;

fp = fopen (“test1.txt” , ”wb”);

fwrite (a, 1, strlen(a), fp);

fclose(fp);

return 0;

}

**c) FREAD( ):** This function is used toReading from a binary file.

The fread( ) function takes Four arguments i) address of data to be read from disk ii) size of data to be read iii) number of such type of data iv) pointer to the file where you want to read.

Syntax: fread (address\_data, size\_of\_variable, number\_of\_elements, pointer\_to\_file);

Example: #include<stdio.h>

int main( )

{

FILE \*fp;

char a[11];

fp = fopen(“test1.txt” , ”rb”);

a[10]=0;

fread (a, 1, 10, fp);

fclose(fp);

printf(“The first 10 characters of the file are:\n %s\n”, a);

return 0;

}

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

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);**

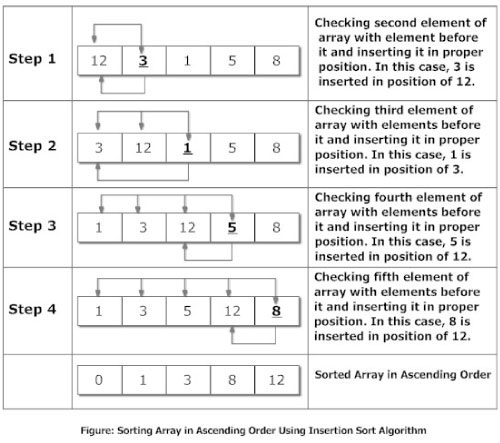
**return (p);**

}

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

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

**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]);

}

}