**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.

**Logical operators**

**Logical Operators:** It is used to combinetwo or more relational expressions (or) conditions with an operator and depending on the resulting truth value, we take certain decisions. The Logical operators are:

|  |  |  |
| --- | --- | --- |
| Operator | Pronounce as | Meaning |
| AND && | double amberscent | satisfy both the conditions. |
| OR || | broken pipe | satisfy either /or condition. |
| NOT ! | Exclamatory | satisfy NOT condition |

Truth table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression 1 (E1)** | **Expression 2 (E2)** | **Logical AND E1 && E2** | **Logical OR**  **E1 || E2** |
| FALSE (0) | FALSE (0) | 0 | 0 |
| FALSE (0) | TRUE (1) | 0 | 1 |
| TRUE (1) | FALSE (0) | 0 | 1 |
| TRUE (1) | TRUE (1) | 1 | 1 |

|  |  |
| --- | --- |
| **Expression 1 (E1)** | **Logical NOT ! (E1)** |
| FALSE (0) | 1 |
| TRUE (1) | 0 |

Example 1 : C= (20==10) && (10>8)

Here, the First expression (20==10) is False. The second expression (10>8) is True. The result of C= (False) && (True) = 0 && 1 = 0 So, **C=0**

Example 2 : C= !(20>8) || (10!=8)

Here, the First expression !(20>8) = !(True) is False. The second expression (10!=8) is True. The result of C= (False) || (True) = 0 || 1 = 1 So, **C=1**

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

**2) Command line arguments**

he arguments passed from command line are called command line arguments. These arguments are handled by main() function.

To support command line argument, you need to change the structure of main() function as given below.

1. **int** main(**int** argc, **char** \*argv[] )

Here, **argc** counts the number of arguments. It counts the file name as the first argument.

The **argv[]** contains the total number of arguments. The first argument is the file name always.

1. #include <stdio.h>

**void** main(**int** argc, **char** \*argv[] )  {

2. printf("Program name is: %s\n", argv[0]);
4. **if**(argc < 2){
5. printf("No argument passed through command line.\n");
6. }
7. **else**{
8. printf("First argument is: %s\n", argv[1]);
9. }
10. }

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

**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**

1. **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);

}

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

1. PARAMETER PASSING TECHNIQUES:

Parameters are nothing but the values of arguments that are passed to called function from calling function.

**Actual arguments/parameters:** Actual parameters are the expressions in the calling functions. i.e., parameters present in the calling statement (function call).

**Formal arguments/parameters:** Formal parameters are the variables that are declared in the header of the function definition. These are the value parameters that copies the values passed from calling function, and are stored in the called functions memory area.

**Note:** Actual and Formal parameters must match exactly in type,order, and number. Their names however, no need to match.

Generally, the Parameters are passed into called function in two ways.

1) Call by Value

2) Call by Reference.

1) Call by Value: It is a process of passing the actual value of variables (arguments) into called function is known as Call by Value.

* Call by value technique calls the function by copying the actual parameter values of calling function into corresponding formal parameters of called function. Ex: Swap (a,b);
* Whatever changes/ modifications done on the formal arguments of called function will not affect the actual arguments of the calling function.

Example: Swapping of Two Numbers

#include <stdio.h>

void swap(int, int);

void main( )

{ int a,b;

printf(“Enter the Values of a and b:”);

scanf(“%d %d”, &a, &b);

printf(“Before Swapping: a = %d \t b = %d”, a, b);

swap(a,b);

printf(“After Swapping: a = %d \t b = %d”, a, b);

}

void Swap (int a, int b)

{ int temp;

temp = a;

a = b;

b = temp;

}

Result: Enter the Values of a and b: 30 40

Before Swapping: a = 30 b = 40

After Swapping: a = 30 b = 40

2) Call by Reference: It is a process of passing the Address of actual arguments into called function is known as Call by Reference. The calling function using pointers to pass the addresses of variables into called function. The function which is called by reference can change the value of the variable used in the call.

* Call by reference technique calls the function by copying the address of actual parameters of calling function into corresponding formal parameters of called function. Ex: Swap (&a, &b)
* Whatever changes/ modifications done on the formal arguments of called function will surely affect the actual arguments of the calling function.

Example: Swapping of Two Numbers

#include <stdio.h>

void swap(int, int);

void main( )

{ int a,b;

printf(“Enter the Values of a and b:”);

scanf(“%d %d”, &a, &b);

printf(“Before Swapping: a = %d \t b = %d”, a, b);

Swap(&a, &b);

printf(“After Swapping: a = %d \t b = %d”, a, b);

}

void Swap (int \*x, int \*y) //x, y are pointer variables holds address of a, b

{ int temp;

temp = \*x;

\*x = \*y;

\*y = temp;

}

Result: Enter the Values of a and b: 100 200

Before Swapping: a = 100 b = 200

After Swapping: a = 200 b = 100

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

1. **Dynamic memory allocation:** It is a process of allocating the memory at runtime (or) during the execution of a program.

The dynamic memory allocation uses four different memory management functions They are: a) malloc( ) Block Memory Allocation

b) calloc( ) Contiguous Memory Allocation

c) realloc( ) Reallocation of memory

d) free( ) Releases the memory

Note: All the memory management functions are found in standard library header file #include<stdlib.h>.

**a) Block Memory Allocation Malloc ( ):** The malloc function is used to allocate the memory. It allocates a Single block of memory from free pool of memory (Heap) which is requested by the user (or) number of bytes specified in its parameter. It returns a void pointer to the first byte of the allocated memory. The allocated memory is not initialized.

Syntax: ptr = (cast-type\*) malloc(byte-size);

Declaration: int \*p;

p = (int \*) malloc (n \* sizeof(int)); // ‘n’ is no. of elements in the array

Example: If n=5, then 5\*2 bytes = 10 bytes will be allocated for the pointer.

**Example: /\* Demonstrate Malloc \*/**

#include <stdio.h>

#include <stdlib.h>

Void main( )

{

int \*p, n, i;

printf (“Enter number of elements:”);

scanf (“%d”, &n);

ptr = (int \*) malloc (n \* sizeof(int)); //memory allocation

for (i=0; i<n; i++)

{

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

scanf (“%d”, (p+i));

}

for (i=0; i<n; i++)

{

printf (“%d \t”, \*(p+i)); // printing of array elements

}

free(p); // releasing of memory

}

**Result:**

Enter number of elements: 4

Enter the element: 20

Enter the element: 30

Enter the element: 40

Enter the element: 50

20 30 40 50

**b) Contiguous Memory Allocation Calloc( ):** The calloc function is also used to allocate the memory. But, it allocates multiple blocks of memory with same size and initializes with Zero. Calloc is mostly used by the derived data types such as Arrays, Structures, Unions etc.. The result is the same for both malloc and calloc.

Syntax: ptr = (cast-type\*) calloc(n, element-size);

Declaration: int \*p;

p = (float \*) calloc (25, sizeof(float));

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

## c) Re-Allocation of Memory Realloc( ): The realloc function can be highly inefficient and therefore should be used advisedly. When given a pointer to a previously allocated block of memory realloc changes the size of the block by deleting or extending the memory at the end of the block. If the memory cannot be extended because of other allocations realloc allocates completely new block, copies the existing memory allocation to the new location, and deletes the old allocation.

Syntax: ptr = (cast-type \*) realloc (ptr, newsize);

Declaration: int \*p;

p = (char \*) realloc (p, 9);

**Example: /\* Demonstrate Malloc and realloc \*/**

#include <stdio.h>

#include <stdlib.h>

Void main( )

{

char \*ptr;

**ptr = (char \*) malloc (6 \* sizeof(char));**

STRCPY (ptr, “MADRAS”);

printf (“The memory contains: %s \n”, ptr);

**ptr = (char \*) realloc (ptr, 7);**

STRCPY (ptr, “CHENNAI”);

printf(“Now the memory contains: %s \n”, ptr);

free (ptr);

}

**Result:**

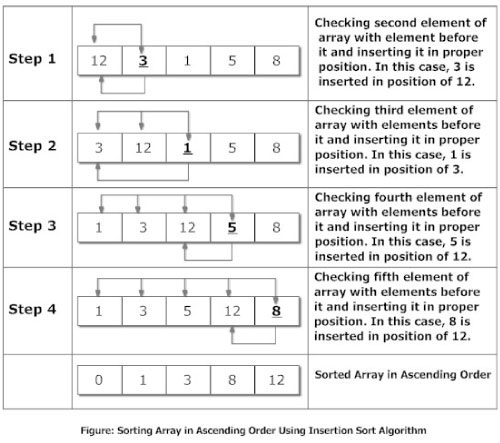
The memory contains: MADRAS

Now the memory contains: CHENNAI

**d) Free ( ):** Free function is used to release the previously allocated memory space.

Syntax: Free (ptr);

1. **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]);

}

}