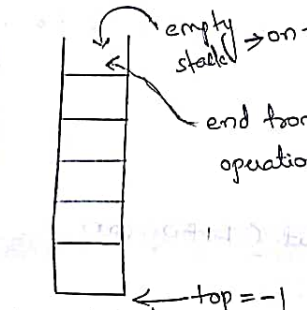
Operations on stack:

① push operation – Insert some data into stack → top incremented by 1 (top++)

② pop operation – Deleting an element from stack → Top decremented by 1 (top--)

③ overflow – if we try to insert some value, when the stack is full, this operation is executed.

④ underflow – if we try to pop some elements, when the stack is empty, this operation is executed.



initial stack.

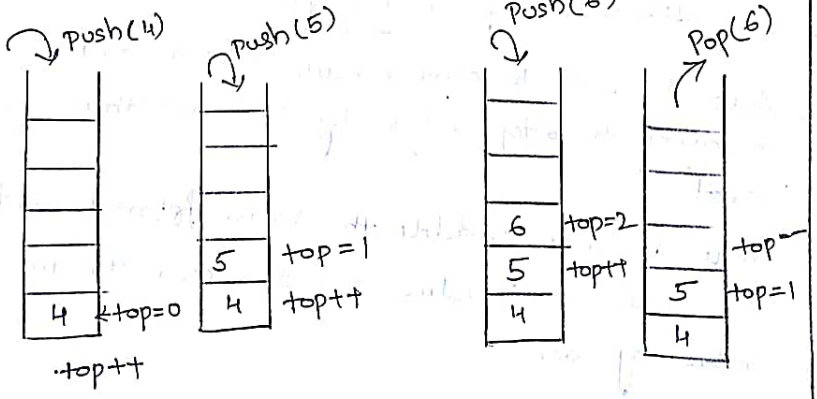


fig: → when the stack size = 6

Scanned by CamScanner

Applications of stack:

→ Handling dynamic Memory Allocations.

→ stack is used to evaluate the expressions.

→ stack is used to convert infix to postfix / prefix form.

→ During a function call the return address and agruments are pushed onto a stack. and return they are popped off.

Implementation of stacks:

→ two types:

① implementation of stack using Arrays.

② implementation of stak using pointers.

① implementation of stack using Arrays:

→ create a empty stack:-

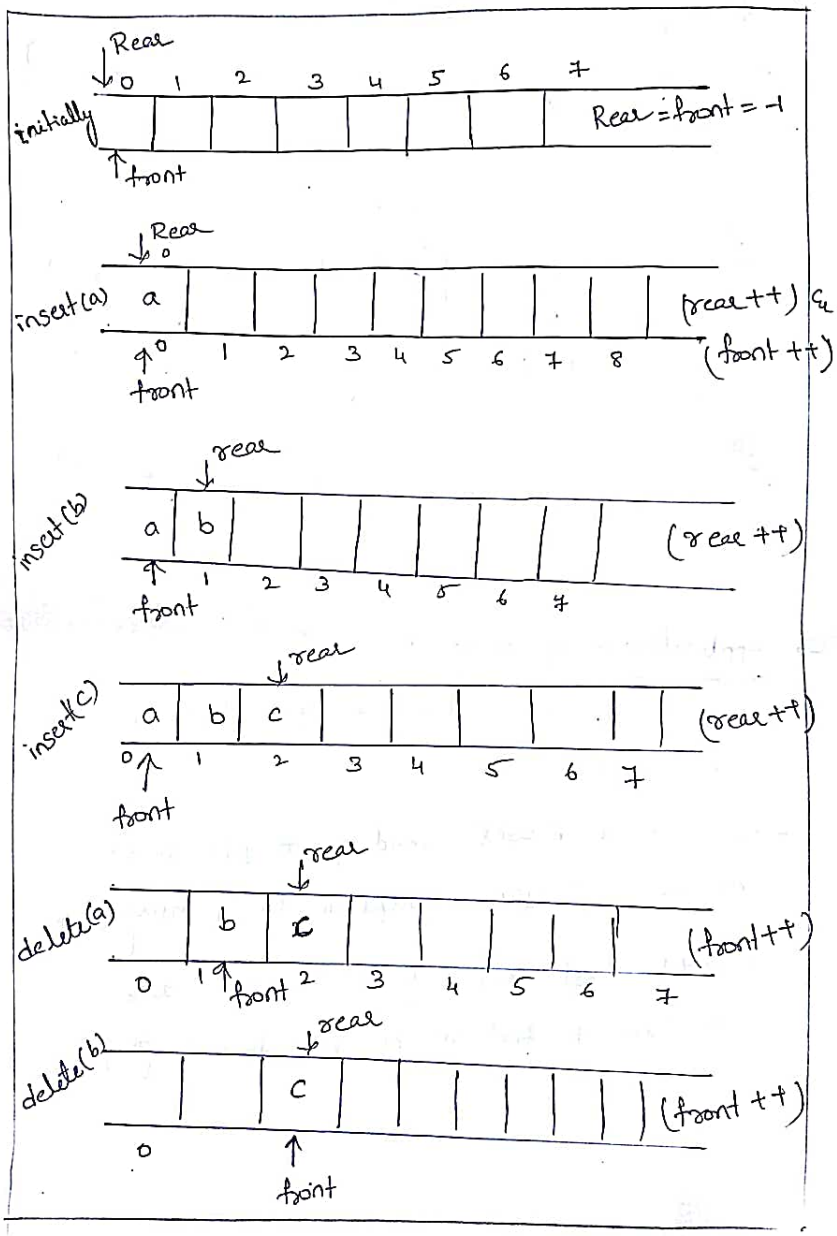
Operations on Queue:

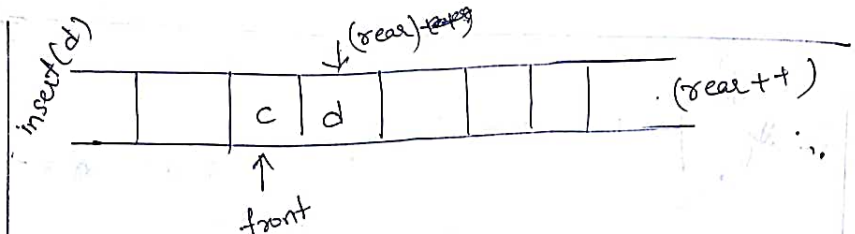
① Insert operation - placing data item into queue. (enQueue)

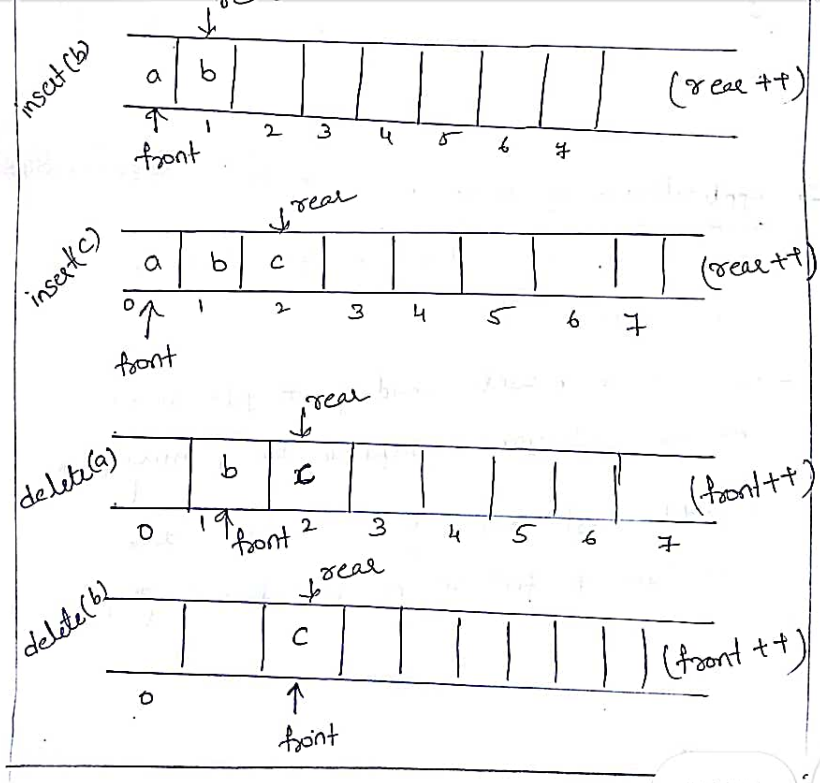
② Delete operation - Removing data item into queue (deQueue)

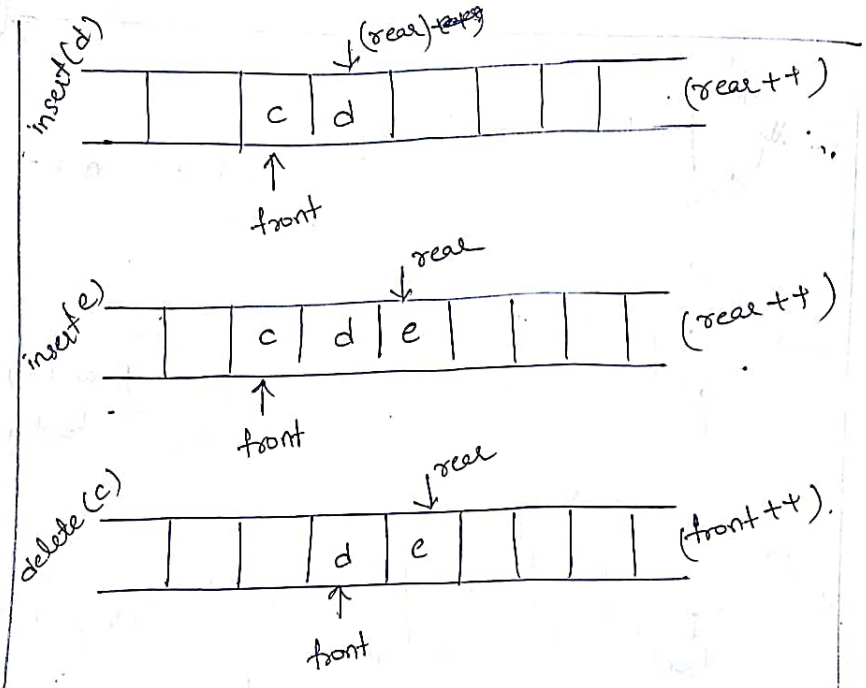
③ Display operation - used to display the elements in queue.

Note: When we insert "first item" into queue, both rear and front will be incremented.









→ **Applications of queue:**

- It is used to schedule the jobs to be processed by the cpu.
- When multiple users send print jobs to be printer, each job is kept in the printing queue.
- Breadth first search uses a queue data structure to find an element from a graph.

Unit - 2

Array

→ An array is a collection of elements of the same datatype.

→ We can use arrays to represent not only simple list of values but also tables of data in two or three dimensional array, the types of arrays are:

1) Single dimensional array
2) Two-dimensional array
3) multi-dimensional array

Single dimensional arrays:-

→ A list of elements can be given one variable name using only one subscript and such variable is called a single-dimensional array.

Syntax

```
type Variable_name [size];
```

Eg:- int A[5];

Initialization:-

→ We can initialize the elements of array in the same way as the ordinary variables when they are declared. The general form of initialization of array is:

```
type arr_name[size] = {list of values};
```

Eg:- int A[5] = {1, 2, 3, 4, 5};

Eg:- Write a program for finding largest elements in the array.

```
#include<stdio.h>
```

Applications of Arrays

In c programming language, arrays are used in wide range of applications. Few of them are as follows...

- **Arrays are used to Store List of values**

In c programming language, single dimensional arrays are used to store list of values of same datatype. In other words, single dimensional arrays are used to store a row of values. In single dimensional array data is stored in linear form.

- **Arrays are used to Perform Matrix Operations**

We use two dimensional arrays to create matrix. We can perform various operations on matrices using two dimensional arrays.

- **Arrays are used to implement Search Algorithms**

We use single dimensional arrays to implement search algorihtms like ...

1. Linear Search
2. Binary Search

- **Arrays are used to implement Sorting Algorithms**

We use single dimensional arrays to implement sorting algorihtms like ...

1. Insertion Sort
2. Bubble Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort, etc.,

- **Arrays are used to implement Datastructures**

We use single dimensional arrays to implement datastructures like...

1. Stack Using Arrays
2. Queue Using Arrays

- **Arrays are also used to implement CPU Scheduling Algorithms**

People also ask

What is the use of sorting algorithm?

A Sorting Algorithm is used **to rearrange a given array or list elements according to a comparison operator on the elements**. The comparison operator is used to decide the new order of element in the respective data structure. For example: The below list of characters is sorted in increasing order of their ASCII values. 22-Apr-2022

Inter Function Communication in C

When a function gets executed in the program, the execution control is transferred from calling a function to called function and executes function definition, and finally comes back to the calling function. In this process, both calling and called functions have to communicate with each other to exchange information. The process of exchanging information between calling and called functions is called inter-function communication.

In C, the inter function communication is classified as follows...

- **Downward Communication**
- **Upward Communication**
- **Bi-directional Communication**

Static Memory Allocation	Dynamic Memory Allocation
Static Memory Allocation memory is allocated at compile time.	Dynamic Memory Allocation memory is allocated at run time.
Memory can not be Changed while executing a program.	memory can be Changed while executing a program.
Used in an array.	Used in the linked list.
It is fast and saves running time.	It is a bit slow.
It allocates memory from the stack.	It allocates memory from the heap
Allocated memory stays from start to	Memory can be allocated at any time and can be

saves running time.	It is a bit slow.
It allocates memory from the stack.	It allocates memory from the heap
Allocated memory stays from start to end of the program.	Memory can be allocated at any time and can be released at any time.
It is less efficient than the Dynamic allocation strategy.	It is more efficient than the Static allocation strategy.
Implementation of this type of allocation is simple.	Implementation of this type of allocation is complicated.
Example: int i; float j;	**Example:** p = malloc(sizeof(int));

Dynamic memory allocation in C

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

Allocation?

Dynamic memory allocation is a process of allocating memory at run time. There are four library routines, calloc(), free(), realloc(), and malloc() which can be used to allocate memory and free it up during the program execution. These routines are defined in the header file called stdlib.h.

What is malloc() ?

It is a function which is used to allocate a block of memory dynamically. It reserves memory space of specified size and returns the null pointer pointing to the memory location.

The pointer returned is usually of type void. It means that we can assign malloc function to any pointer. The full form of malloc is memory allocation.

In this tutorial, you will learn:

What is calloc() ?

Calloc() function is used to allocate multiple blocks of memory. It is a dynamic memory allocation function which is used to allocate the memory to complex data structures such as arrays and structures. If this function fails to allocate enough space as specified, it returns null pointer. The full form of calloc function is contiguous allocation.

Why use malloc() ?

Here are the reasons of using malloc()

Here is a Syntax of malloc()

```
ptr = (cast_type *) malloc
(byte_size);
```

In above syntax, ptr is a pointer of cast_type. The malloc function returns a pointer to the allocated memory of byte_size.

```
Example: ptr = (int *) malloc
(50)
```

When this statement is successfully executed, a memory space of 50 bytes is reserved. The address of the first byte of reserved space is assigned to the pointer “ptr”of type int.

Syntax of calloc()

Here is a Syntax of malloc()

```
ptr = (cast_type *) calloc (n,
size);
```

Scope Rules in C

Written By - admin

Scope Rules in C:

Scope rules in C or scope of a variable means that from where the variable may directly be accessible after its declaration.
The scope of a variable in C programming language can be declared in three places :

Scope	Place
Local Variable	Inside a function or block
Global variable	Outside of all function(can be accessed from anywhere)
Formal Parameters	In the function parameters

What is the use of type qualifier?

A type qualifier is used **to refine the declaration of a variable, a function, and parameters**, by specifying whether: The value of an object can be changed. The value of an object must always be read from memory rather than from a register. More than one pointer can access a modifiable memory address.

https://www.ibm.com › cv_qualifiers

View / Hide Units

Advantages of recursion

1. The code may be easier to write.
2. To solve such problems which are naturally recursive such as tower of Hanoi.
3. Reduce unnecessary calling of function.
4. Extremely useful when applying the same solution.
5. Recursion reduce the length of code.
6. It is very useful in solving the data structure problem.
7. Stacks evolutions and infix, prefix, postfix evaluations etc.

Disadvantages of recursion

1. Recursive functions are generally slower than non-recursive function.
2. It may require a lot of memory space to hold intermediate results on the system stacks.
3. Hard to analyze or understand the code.
4. It is not more efficient in terms of space and time complexity.
5. The computer may run out of memory if the recursive calls are not properly checked.

Previous

GCD (Example of recursive algorithm)

Hurray! this unit is completed

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

https://www.tutorialspoint.com › c_r...

C - Recursion - Tutorialspoint

About featured snippets Feedback

People also ask

What is recursion in C and types?

Recursion is **the process in which a function calls itself up to n-number of times**. If a program allows the user to call a function inside the same function recursively, the procedure is called a recursive call of the function. Furthermore, a recursive function can call itself directly or indirectly in the same program.

https://www.javatpoint.com › types-...

Types of Recursion in C - javatpoint

STACK:

A linear Datastructure in which data item is inserted or deleted on one end.

- the end where the data items are inserted or deleted is called "TOP OF THE STACK".
- STACK is called as "LastInFirstOut (LIFO)" (or) "FirstInLastOut (FILO)" structure.
- the initial value of the TOP OF THE STACK is "-1"
- Where we want to insert a value into a stack, increment the stop value by one. and then insert.
- Where we want to delete the value from the stack, the delete the top value and decrement the top value by one.

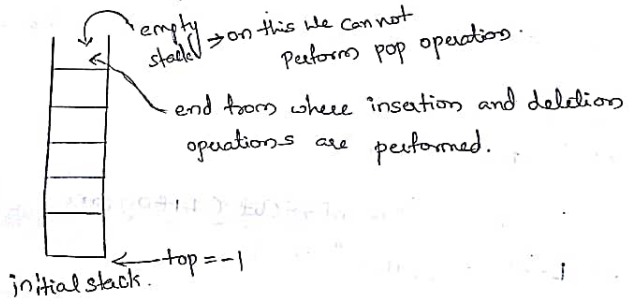
Operations on stack:

① push operation – Insert some data into stack → top incremented by 1 (top++)

② pop operation – Deleting an element from stack → Top decremented by 1 (top--)

③ overflow – if we try to insert some value, when the stack is full. this operation is executed.

④ underflow. ↓ if we try to pop some elements, when the stack is empty, this operation is executed.



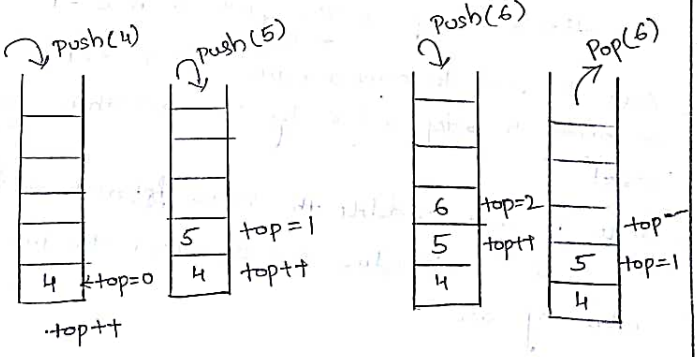


fig: → when the stack size = 6

Applications of stack:

→ Handling dynamic Memory Allocations.

stack size = 6

Applications of stack:

→ Handling dynamic Memory Allocations.
→ stack is used to evaluate the expressions.
→ stack is used to convert infix to postfix / prefix form.
→ During a function call the return address and arguments are pushed onto a stack. and return they are popped off.

Implementation of stacks:

→ two types:
① implementation of stack using Arrays.
② implementation of stak using pointers.

① implementation of stack using Arrays:

→ To create a empty stack:-

step1: include headerfiles which are used in the program. and define a constant size with specific value.

step2: create one dimensional array with fixed size. (int stack[SIZE].)

step3: define an integer variable "top" and initize with '-1'. (int top = -1)

push operation:

step1: check whether stack is full. (top == SIZE-1)
step2: if stack is full insertion is not possible.
step3: If not full, then increment top value. (top++). and set stack[top] to value. (stack[top] = value).

Pop operation:

step1: check whether stack is empty (top == -1).
step2: if stack is empty then display stack is empty "Deletion is not possible".
step3: if it is not empty then delete "stack[top]" & decrement top value by one. (top--)

Example program :-

```
/* implementation of stack using arrays */
#include <stdio.h>
```

decrement top value by one. (top--)

Example programs :-

```
/* implementation of stack using arrays */
#include <stdio.h>
#include <conio.h>
void main()
{
int s[20], i, a, size, top=-1; choice;
```



```
clrscr();
printf("enter a stack size:\n");
scanf("%d", &size);
while(1)
{
Printf("enter choice 1. push\n 2. pop\n 3. display\n
        4. top\n 5. exit\n");
scanf("%d", &choice);
switch(choice)
 {
  case 1: if (top == size-1)
          {
          printf("stack is full\n");
          }
          else
          {
          printf("enter the element tobe
                  insated");
          scanf("%d", &a);
           top++;
           s[top]=a;
          }
         break;
```



```
case 2: if (top==-1)
         {
          printf("stack is empty");
         }
         else
         {
          printf("element popped %d", s[top]);
          s[top]=0;
          top--;
         }
         break;

case 3: if (top==-1)
        {printf("stack is empty\n");
        }
        else
        {
         printf("elements of stack are\n");
         for(i=top; i>=0; i--)
          printf("%d\n", s[i]);
        }
        break;
case 4: printf("top element = %d", s[top]);
        break;
case 5: exit(1);
        break;
default: printf("please enter choice between 1 to 5
                only");
        break;
 }
 }getch();
```


O/p:

```
enter the stack size : 5 ↵

enter choice : 1. push
               2. pop
               3. display
               4. top
               5. exit.

1 ↵
enter the element tobe inserted : 15 ↵

1
2
3
5 ↵
exit.
```

Implementation of Queue :-

two types:

① implimentation of queue using arrays.

② implimentation of queue using pointers.

→ **implementation of Queue using Arrays:**

To create a empty stack:-

step1: include headerfiles which are used in the program and define a constant size with specific value.

step2: create one dimensional array with fixed size (int queue[SIZE])

step3: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front=-1, rear=-1).

enQueue / insertion operation:

step1: check whether Queue is full (rear==size-1)

step2: if Queue is full insertion is not possible.

step3: if it is notfull, then increment rear value. (rear++). and set "queue[rear]= value".

deQueue / Deleting operation :-

step1: check whether Queue is empty. (front == rear).

step2: if Queue is empty then display Queue is empty. "Deletion is not possible".

step3: if it is not empty then increment the front value by one (front++). then display Queue[front] as 'deleted element.

→ then check whether both front and rear are equal (front == rear), if it true, then set both front and rear to '-1'. (front = rear = -1).

Example program:

```
/* implementation of Queue using array */.
#include <stdio.h>
#include <conio.h>
void main()
{
 int s[20], i, a, size, front = -1, rear = -1,
     choice;
 clrscr();
 printf("enter a queue size:\n");
 scanf("%d", &size);
 while(1)
  {
   printf("enter choice 1. insert \n 2. delete\n
          3. display \n 4. exit \n);
   scanf("%d", &choice);
   switch(choice)
   {
case1: if(rear == size-1)
       {
        printf("queue is full\n");
       }
```



```
      else
       {
        if(front == -1) //initially queue is empty)
        {
         front = 0;
        }
        pf("enter element to add in queue \n");
        sf("%d", &add_value);
        rear = rear+1;
        a[rear] = add_value;
       }
       break;
case2: if(front == -1 || front > rear)
       {
        pf("Queue is empty \n");
       }
       else
       {
        pf("element deleted from queue%d\n",
                                a[front]);
        front = front+1;
       }
       break;
```



```
case3: if(front == -1)
       {
        pf("Queue is empty\n");
       }
       else
       {pf("Queue is \n");
        for(i=front; i<=rear; i++)
        {
         pf("%d", s[i]);
        }
       break;
case4: exit(1);
       break;
default: printf("please enter choice between
                 1 to 4 only");
         break;
       }
  }
 getch();
}.
```


O/P:

```
enter the Queue size : 5
enter choice : 1. insert
               2. delete
               3. display
               4. exit.
1
enter element to add in queue : 10
.
.
.
4
exit:
```

enclosed in braces.

Eg:- int A[2][3] = {0,0,0,1,1,1};

Eg:- Addition of a matrices.

```
#include <stdio.h>
#include <conio.h>
Void main()
{
   int i,j, a[3][3], b[3][3], c[3][3];
   clrscr();
   Printf ("Enter the first matrix:\n");
   for (i=0; i<3; i++)
   {
     for(j=0; j<3; j++)
```

Scanned by CamScanner

```
printf ("%d", a[i][j]");
Scanf ("%d", &a[i][j]");
}
}
printf (" Enter the second matrix:\n");
for(i=0; i<3; i++)
{
  for(j=0; j<3; j++)
   {
     Printf ("%d", b[i][j]");
     Scanf ("%d", &b[i][j]);
   }
}
printf ("The entered matrices are:\n");
for(i=0; i<3; i++)
{
  printf ("\n");
  for(j=0; j<3; j++)
  Printf ("%d\t", a[i][j]);
  Printf("\t");
  for(j=0; j<3; j++)
  Printf("%d\t", b[i][j]);
}
```

Scanned by CamScanner


```
for(i=0; i<3; i++)
for(j=0; j<3; j++)
c[i][j] = a[i][j] + b[i][j];
printf ("\n the sum of two matrices are");
for(i=0; i<3; i++)
{
printf ("\n\t");
for(j=0; j<3; j++)
 Printf ("%d\t", c[i][j]);
}
getch();
}
```

```c
#include <stdio.h>
int main() {
  int r, c, a[100][100], b[100][100], sum[100][100], i,
j;
  printf("Enter the number of rows (between 1
and 100): ");
  scanf("%d", &r);
  printf("Enter the number of columns (between 1
and 100): ");
  scanf("%d", &c);

  printf("\nEnter elements of 1st matrix:\n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("Enter element a%d%d: ", i + 1, j + 1);
      scanf("%d", &a[i][j]);
    }

  printf("Enter elements of 2nd matrix:\n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("Enter element b%d%d: ", i + 1, j + 1);
      scanf("%d", &b[i][j]);
    }

  // adding two matrices
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      sum[i][j] = a[i][j] + b[i][j];
    }

  // printing the result
  printf("\nSum of two matrices: \n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("%d   ", sum[i][j]);
      if (j == c - 1) {
        printf("\n\n");
      }
    }
```

they cannot get multiplied by each other, then this program will generate an error message.

```
#include<stdio.h>

int main(void)
{
  int c, d, p, q, m, n, k, tot = 0;
  int fst[10][10], sec[10][10], mul[10][10];

  printf(" Please insert the number of rows and
columns for first matrix \n ");
  scanf("%d%d", &m, &n);

  printf(" Insert your matrix elements : \n ");
  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
      scanf("%d", &fst[c][d]);

  printf(" Please insert the number of rows and
columns for second matrix\n");
  scanf(" %d %d", &p, &q);

  if (n != p)
    printf(" Your given matrices cannot be
multiplied with each other. \n ");
  else
  {
    printf(" Insert your elements for second matrix
\n ");

    for (c = 0; c < p; c++)
      for (d = 0; d < q; d++)
        scanf("%d", &sec[c][d] );

    for (c = 0; c < m; c++) {
      for (d = 0; d < q; d++) {
        for (k = 0; k < p; k++) {
          tot = tot + fst[c][k] * sec[k][d];
        }
        mul[c][d] = tot;
        tot = 0;
```

```
  printf(" Insert your matrix elements : \n ");
  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
      scanf("%d", &fst[c][d]);

  printf(" Please insert the number of rows and
columns for second matrix\n");
  scanf(" %d %d", &p, &q);

  if (n != p)
    printf(" Your given matrices cannot be
multiplied with each other. \n ");
  else
  {
    printf(" Insert your elements for second matrix
\n ");

    for (c = 0; c < p; c++)
      for (d = 0; d < q; d++)
        scanf("%d", &sec[c][d] );

    for (c = 0; c < m; c++) {
      for (d = 0; d < q; d++) {
        for (k = 0; k < p; k++) {
          tot = tot + fst[c][k] * sec[k][d];
        }
        mul[c][d] = tot;
        tot = 0;
      }
    }

    printf(" The result of matrix multiplication or
product of the matrices is: \n ");
    for (c = 0; c < m; c++) {
      for (d = 0; d < q; d++)
        printf("%d \t", mul[c][d] );
      printf(" \n ");
    }
  }
  return 0;
}
```

2:47 pm ✓✓

Linear search:-

→ The linear search or the sequential searching is most simple searching method.

→ The key, which is to be searched is compared with each element of list one by one, if the match exists, the search is terminated.

→ If the end of the list is reached, it means that the search has failed and the key has no matching element in the list.

Eg:-

0	1	2	3	4	5	6	7
60	20	10	55	32	12	50	99

Key = 12

Step1

Search element 12 is compared with first element 60

0	1	2	3	4	5	6	7
60	20	10	55	32	12	50	99

12

Both are not matching. So move to next element.

Step2

Search element 12 is compared with next element 20.

0	1	2	3	4	5	6	7
60	20	10	55	32	12	50	99

12

Both are not matching, so move to next element 10.

Step3

Search element 12 is compared with next element 10

0	1	2	3	4	5	6	7
60	20	10	55	32	12	50	99

12

Both are not matching, so move to next element 55

Step4

Search element 12 is compared with next element 55

0	1	2	3	4	5	6	7
60	20	10	55	32	12	50	99

12

Both are not matching, so move to next element 32

Step5

Search element 12 is compared with next element 32

0	1	2	3	4	5	6	7
60	20	10	55	32	12	50	99

12

Both are not matching, so move to next element

Step6

Search element 12 is compared with next element 12

0	1	2	3	4	5	6	7
60	20	10	55	32	12	50	99

12

Both are matching, so we stop comparing and display element found at index 5.


```
#include <stdio.h>
#include <conio.h>
main()
{
int A[5] = {10,20,30,40,50};
int Key,flag=0;
printf("Enter search Key");
scanf("%d",&Key);
for(i=0; i<5; i++)
{
  if(Key==A[i])
    {
      flag=1;
      break;
    }
}
if(flag==-1)
{
  printf("Key is found");
}
  else
    {
      printf("Key not found");
    }
}
```

Binary search:-

-> The main constraint of binary search is that the elements should be in ascending order.

-> The given list of elements are divided into two equal halves, the given key is compared with the middle element of the list. Now three situations may occur:

i) The middle element matches with the key - the search will end peacefully here.

ii) The middle element is greater than the key - then the value which we are searching is in the first half of the list.

iii) The middle element is lower than the key - then the value which are searching is in the second half of the list.

Eg:-

0	1	2	3	4	5
20	23	52	56	89	93

Key = 93

Step 1

0	1	2	3	4	5
20	23	52	56	89	93
low		mid			high

-the key element is not matched with middle element

∵ 52 < 93, so the searching is moved to right side of the list.

Step 2

0	1	2	3	4	5
20	23	52	56	89	93
			low	mid	high

∵ 89 < 93 (moved to right side part of mid element)

Step 3

20	23	52	56	89	93
					high (key)

∵ key is found.

Binary search:-

```
#include <stdio.h>
#include <conio.h>
main()
{
 int A[5] = {10,20,30,40,50};
 int key = 50, flag = 0;
 int low = 0; high = 4;
 while (low < high)
```



```
{
 mid = (low+high)/2;
 if(key == A[mid])
 {
   flag = 1;
   break;
 }
 else if (key < A[mid])
  high = mid - 1;
 else
  low = mid + 1;
 }
 if(flag == 1)
 printf("key found");
 else
 printf("key not found");
}
```

Bubble sort :-

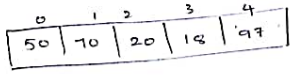
→ The simplest and the oldest sorting technique is bubble sort.

→ The method takes two elements at a time, it compares these two elements, if the first element is less than the second element, they are left undisturbed, if the first element

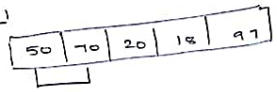
∴ greater than the second element then they are swapped.

→ The procedure continues with the next two elements goes and ends when all the elements are sorted.

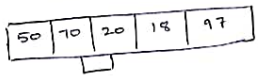
Eg :-



pass 1

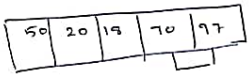


∵ 50 < 70 (No need to swap)



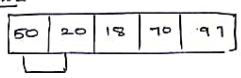
∵ 70 > 20 (need to swap)

∵ 70 > 18 (need to swap)

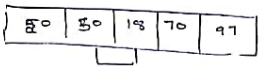


∵ 70 < 97 (no need to swap)

pass 2



∵ 50 > 20 (need to swap)

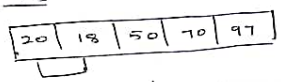


∵ 50 > 18 (need to swap)

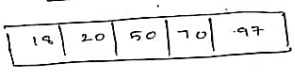
∵ 50 < 70 (no need to swap)

∵ 70 < 97 (no need to swap)

pass 3



∵ 20 > 18 (need to swap)



∴ Finally elements are in ascending order.

Bubble sort :-

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i,j,temp=0;
 for(i=0;i<n;i++)
 {
  for(j=i+1;j<n;j++)
  {
   if(arr[j]>arr[j+1])
   {
    temp=arr[j];
    arr[j]=arr[j+1];
    arr[j+1]=temp;
   }
  }
 }
}
```

People also ask

What are the categories of user-defined functions?

Types of user defined functions in C

- Category I: Functions with no arguments and no return values.
- Category 2: Functions with no arguments and with return values.
- Category 3: Functions with arguments and no return values.
- Category 4: Functions with arguments and with return values.

06-Mar-2020

https://www.faceprep.in › c › types-...

Types of user defined functions in C - FACE Prep

More results

What are the three types of user-defined functions?

What are the different categories of functions explain with example?

be returned, the function body should be written in such a way that it reads the input value, increases its value by 10 and prints the result within the function itself.

```
include <stdio.h>
void modify()                //function definition
{
int a;
printf("Enter a value:");
scanf("%d",&a);
a=a+10;
printf("The modified value is %d",a);
return;
}

int main()
{
modify();                    //function call
getch();
return 0;
}

output: if a=1
The modified value is 11.
```

```
include <stdio.h>
void modify()
{
int a;
printf("Enter a value:" )
scanf("%d",&a);
a=a+10;
return a;
}

int main()
{
int x;
x=modify();
printf("The modified value is %d",x);
getch();
return 0;
}
```

```
include <stdio.h>
void modify(int x)              // function takes the ar
{
x = x + 10;
printf("The modified value is %d", x);
return;
}

int main()
{
int a;
printf("Enter a value:");
scanf("%d",&a);
modify(a);                      // pass the argument
getch();
return 0;
}
```

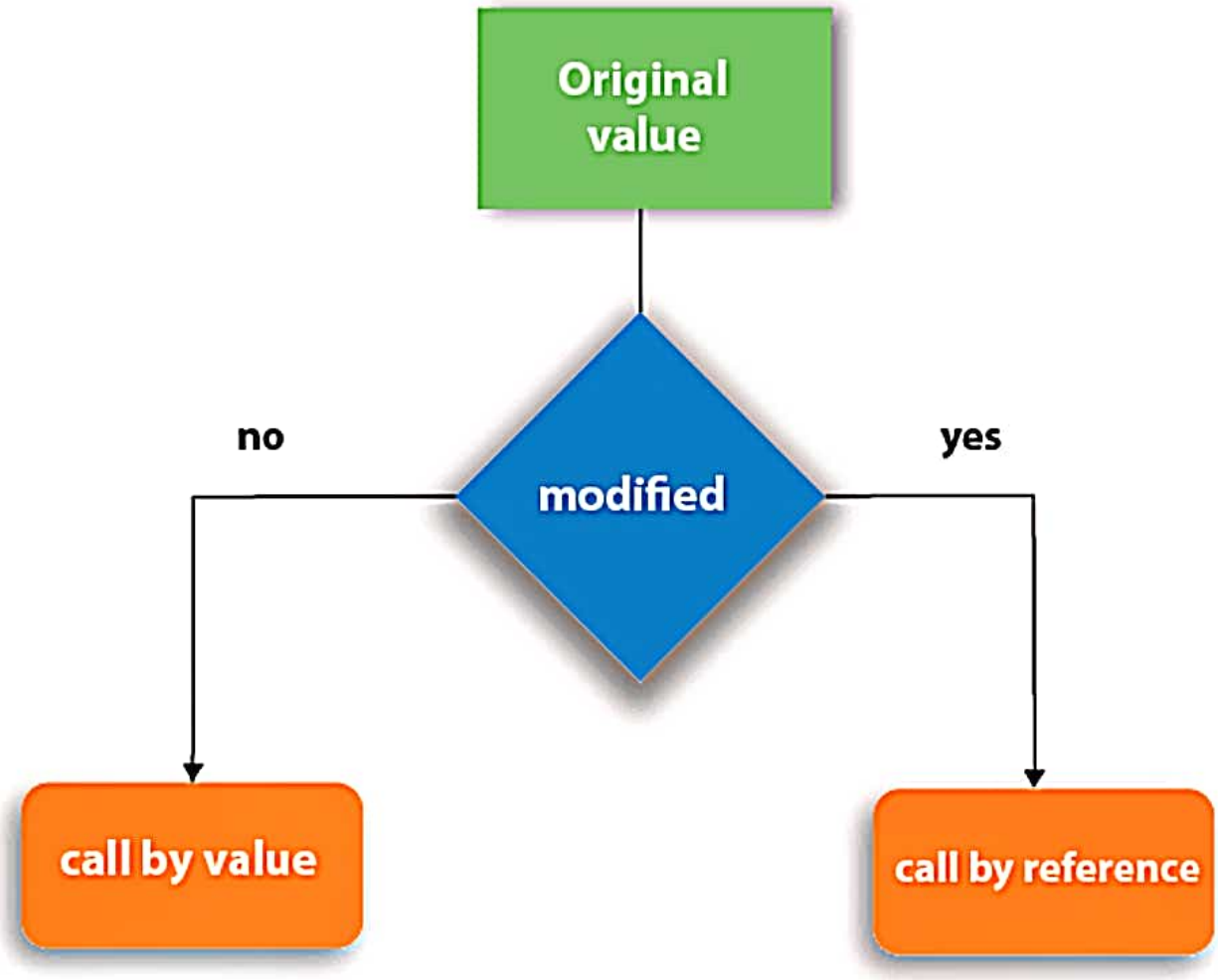
```
include <stdio.h>
void modify(int x)    // function takes the argume
{
x = x + 10;
return x;
}

int main()
{
int a;
printf("Enter a value:");
scanf("%d",&a);
printf("the modified value is %d", modify(a));   // p
getch();
return 0;
}
```

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int n, a, b;
   clrscr();
   printf("Enter any number\n");
   scanf("%d", &n);
   a = recfactorial(n);
   printf("The factorial of a given number using
recursion is %d \n", a);
   b = nonrecfactorial(n);
   printf("The factorial of a given number using
nonrecursion is %d ", b);
   getch();
}
int recfactorial(int x)
{
   int f;
   if(x == 0)
   {
    return(1);
   }
   else
   {
    f = x * recfactorial(x - 1);
    return(f);
   }
}
int nonrecfactorial(int x)
{
   int i, f = 1;
   for(i = 1;i <= x; i++)
   {
     f = f * i;
   }
   return(f);
```

2:58 pm

Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.



Let's understand call by value and call by reference in c language one by one.

Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

```c
#include<stdio.h>
void change(int num) {
    printf("Before adding value inside function num=%d \n"
    num=num+100;
    printf("After adding value inside function num=%d \n", n
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(x);//passing value in function
    printf("After function call x=%d \n", x);
return 0;
}
```

)utput

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```
#include<stdio.h>
void change(int *num) {
    printf("Before adding value inside function num=%d \n"
    (*num) += 100;
    printf("After adding value inside function num=%d \n", *
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x);//passing reference in function
    printf("After function call x=%d \n", x);
return 0;
}
```

3a. Describe different storage classes available in C.

A storage class defines the scope (visibility) and life-times of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program −

- auto
- register
- static
- extern

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables within the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
```



```
  register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

  while(count--) {
    func();
  }

  return 0;
}

/* function definition */
void func( void ) {

  static int i = 5; /* local static variable */
  i++;

  printf("i is %d and count is %d\n", i, count);
}
```

When the above code is compiled and executed, it produces the following result −

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
```

```
/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

  while(count--) {
    func();
  }

  return 0;
}

/* function definition */
void func( void ) {

  static int i = 5; /* local static variable */
  i++;

  printf("i is %d and count is %d\n", i, count);
}
```
When the above code is compiled and executed, it produces the following result –
```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
```



```
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c
```
#include <stdio.h>

int count ;
extern void write_extern();

main() {
  count = 5;
  write_extern();
}
```
Second File: support.c
```
#include <stdio.h>

extern int count;

void write_extern(void) {
  printf("count is %d\n", count);
}
```
Here, extern is being used to declare *count* in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows –
```
$gcc main.c support.c
```
It will produce the executable program **a.out**. When this program is executed, it produces the following result –
```
count is 5
```

C language passing an array to function example

```
#include<stdio.h>
int minarray(int arr[],int size){
int min=arr[0];
int i=0;
for(i=1;i<size;i++){
if(min>arr[i]){
min=arr[i];
}
}//end of for
return min;
}//end of function

int main(){
int i=0,min=0;
int numbers[]={4,5,7,3,8,9};//declaration of array

min=minarray(numbers,6);//passing array with size
printf("minimum number is %d \n",min);
return 0;
}
```

3:07 pm ✓✓

Passing Array to Function in C

In C, there are various general problems which requires passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

As we know that the array_name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter.

GCD of Two Numbers using Recursion

```
#include <stdio.h>
int hcf(int n1, int n2);
int main() {
    int n1, n2;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);
    printf("G.C.D of %d and %d is %d.", n1, n2,
hcf(n1, n2));
    return 0;
}

int hcf(int n1, int n2) {
    if (n2 != 0)
        return hcf(n2, n1 % n2);
    else
        return n1;
}
```

3:10 pm ✓✓

Agenda:

★**Functions:** Designing Structured Programs,

★ Function in C,

★User Defined Functions,

★Inter-Function Communication,

★ Standard Functions

★Passing Array to Functions,

★ Passing Pointers to Functions,

★ Recursion,

Designing Structured Programs

★ Structured programming is a programming technique in which a larger program is divided into smaller subprograms to make it easy to understand, easy to implement and makes the code reusable, etc.

★ Structured programming enables code reusability.

★ **Code reusability** is a method of writing code once and using it many times. Using a structured programming technique, we write the code once and use it many times.

★ Structured programming also makes the program easy to understand, improves the quality of the program, easy to implement and reduces time.

★In C, the structured programming can be designed using **functions** concept.

★ Using functions concept, we can divide the larger program into smaller subprograms and these subprograms are implemented individually.

★Every subprogram or function in C is executed individually.

Functions in c

★ When we write a program to solve a larger problem, we divide that larger problem into smaller subproblems and are solved individually to make the program easier.

★In C, this concept is implemented using functions.

★ Functions are used to divide a larger program into smaller subprograms such that the program becomes easy to understand and easy to implement. A function is defined as follows...

★Function is a subpart of a program used to perform a specific task and is executed individually.

★ In programming, a function is a segment that groups code to perform a specific task. A C program has at least one function main().Without main() function, there is technically no C program.

★ Every C program must contain at least one function called main(). However, a program may also contain other functions.Every function in C has the following...

1. Function Declaration (Function Prototype)
2. Function Definition
3. Function Call

Parameter Passing in C

★ When a function gets executed in the program, the execution control is transferred from calling-function to called function and executes function definition, and finally comes back to the calling function. When the execution control is transferred from calling-function to called-function it may carry one or number of data values. These data values are called as parameters.

★Parameters are the data values that are passed from calling function to called function.

★In C, there are two types of parameters and they are as follows...

1. Actual Parameters
2. Formal Parameters

★The **actual parameters** are the parameters that are speficified in calling function.

★ The **formal parameters** are the parameters that are declared at called function. When a function gets executed, the copy of actual parameter values are copied into formal parameters.

★ In C Programming Language, there are two methods to pass parameters from calling function to called function and they are as follows...

★A function can be called by two ways. They are:

1. Call by value
2. Call by reference

Call by Value:

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming language uses *call by value* method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

← CPPS_UNIT II_ARRAYS.docx

UNDERSTANDING ARRAYS:

ARRAY: Array is a data structure, which provides the facility to store a collection of data of same type under single variable
name. Just like the ordinary variable, the array should also be declared properly. The declaration of array includes the type of array that is the type of value we are going to store in it, the array name and maximum number of elements.(or) An Array is an index collection of fixed no. of homogenous data elements.

- ★ C Programming Arrays is the Collection of Elements
- ★ C Programming Arrays is collection of the Elements of the same data type.
- ★ All Elements are stored in the Contiguous memory
- ★ All elements in the array are accessed using the subscript variable.
- ★ The type may be any valid type supported by C. Array names, like other variable names, can contain only letter, digit and underscore characters.
- ★ Array names cannot begin with a digit character. I.e., The rule for giving the array name is same as the ordinary variable.
- ★ The size should be an individual constant.
- ★ To refer to a particular location or element in the array, we specify the name of the array and the index of the particular element in the array.
- ★ The index specifies the location of the element in the array.
- ★ The array index starts from zero. The maximum index value will be equal to the size of the array minus one.
- ★ The array index can be an integer variable for an integer constant.
- ★ The main advantage of array is we can represent no. of values by using single variable so that the readability of code will be improved .
- ★ The main disadvantage is fixed in size that is once we create an array there is no chance to increase or decrease the size.

TYPES OF C ARRAYS:

There are 2 types of C arrays. They are,

1. One dimensional array 2.Multi dimensional array
   - ★ Two dimensional array
   - ★ Three dimensional array
   - ★ four dimensional array etc...

ONE OR SINGLE DIMENSIONAL ARRAY IN C:

- ★ Single or One Dimensional array is used to represent and store data in a linear form.
- ★ Array having only one subscript variable is called One-Dimensional array
- ★ It is also called as Single Dimensional Array or Linear Array