
POINTERS

Pointer:

A pointer is an address variable which can store address of another variable where two variables must belong to same data type.

Pointers are developed for dealing with user defined functions such as call by reference (or) call by address (or) pass by reference.

Features of pointers:

- Pointers save the memory space.
- Execution time the pointer is faster because better is manipulated with the address i.e., direct access to memory location.
- The memory is accessed efficiently with pointers.
- The pointers are used with data structures. They are useful for representing 2D & multidimensional arrays.
- Dynamically memory is allocated.

Syntax : <Data type> <*pointer variable name>;

Eg: int *ptrname;

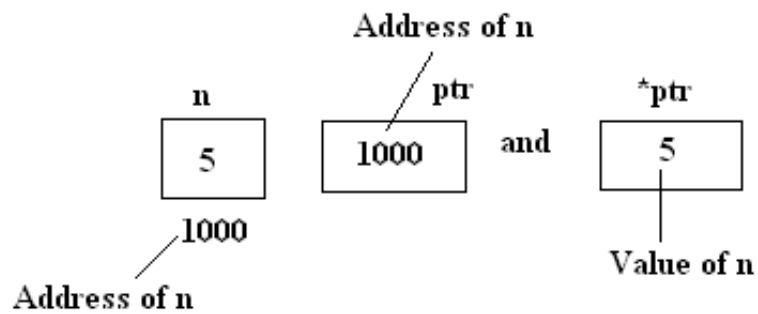
When a pointer variable is declared it must be preceded with astrick (*) is called indirection operator (or) dereferencing operator. When a pointer is dereferences, the value at that address stored by the pointer is retrieved.

When a pointer is dereferencing, the indirection operator indicates that the value at that memory location stored in the pointer is to be accessed, rather than address itself.

Note: Once pointer variable is declared, it can able to store address of the same caste (data type).

```
int *ptr, n;  
n=5;  
ptr=&n;  
ptr=1000    —————>address
```

Now 'ptr' stores '&n' (address of n), thus the *ptr contains value of n.



Use the format specifier '%u' (unsigned int data type), while printing the address of the pointer variable.

The following program illustrate the use of pointer variable in C.

Aim: To write a C program to illustrate the use of pointer variable in C.

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

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
    int n,*ptr;
```

```
    clrscr();
```

```
    printf("\n Enter an integer:");
```

```
    scanf("%d",&n);
```

```
    ptr=&n; //Stores the address of n
```

```
    printf("\n Value of n is: %d",n);
```

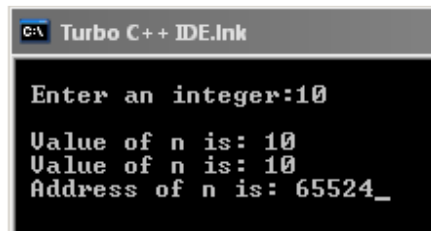
```
    printf("\n Value of n is: %d", *ptr);
```

```
    printf("\n Address of n is: %u", ptr);
```

```
    getch();
```

```
}
```

Output:

A screenshot of the Turbo C++ IDE window. The title bar reads 'C:\ Turbo C++ IDE.Ink'. The main window area has a black background with white text. It shows the prompt 'Enter an integer:10' followed by the output 'Value of n is: 10', 'Value of n is: 10', and 'Address of n is: 65524_'.

```
C:\ Turbo C++ IDE.Ink
Enter an integer:10
Value of n is: 10
Value of n is: 10
Address of n is: 65524_
```

Pointers with Arrays:

To represent array data with pointers we have to initialize the 1st element address to the pointer variable.

Ex: `int a[10], *ptr;`

From the above concept to read values for the `a[10]` with respect to pointers we have to assign address of 1st memory location of an array variable to the `ptr` variable i.e., `ptr=&a[0]`.

Now the `ptr` refers the 1st element location array only. To move to the next location the pointer variable should be increment i.e., `ptr++`;

After reading all elements the pointer refers the address of last array element location depending upon the condition. To write values from the starting index again the pointer should start from the 1st location i.e., `ptr=&a[0]`;

To print all the elements the pointer variable should be increment i.e., `ptr++`;

Ex: consider an array `a[10]`

In which, `n=5, *ptr;`

`ptr =&a[0];`

Now `ptr` refers the 1st element address, we read value at that address as like, `scanf("%d",ptr)`; To read next elements in the array we should increment `ptr` i.e., `ptr++`;

Now the pointer variable is at the last array location, to print array values again we have to assign the array first element address to the '`ptr`' i.e., `ptr=&a[0]`.

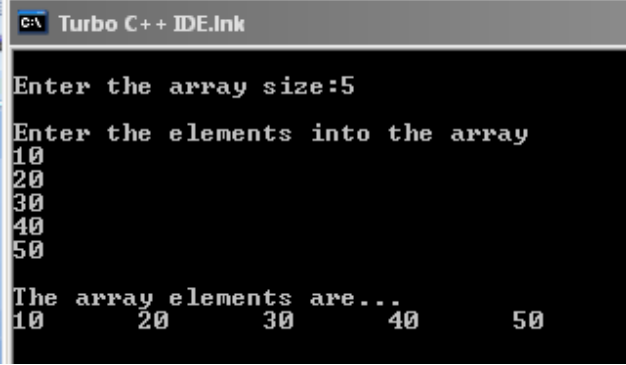
To print all elements in the array we have to increment the `ptr` variable as `ptr++`.

The following program illustrates the above concept.

Aim: To write a C program to read and print an array of elements by using a pointer variable.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n,a[10],*ptr,i;
    clrscr();
    printf("\nEnter the array size:");
    scanf("%d",&n);
    ptr=&a[0];
    //Reading array elements using pointer variable
    printf("\nEnter the elements into the array\n");
    for(i=1;i<=n;i++)
    {
        scanf("%d",ptr);
        ptr++;
    }
    //Printing array elements using pointer variable
    ptr=&a[0];
    printf("\nThe array elements are...\n");
    for(i=1;i<=n;i++)
    {
        printf("%d\t",*ptr);
        ptr++;
    }
    getch();
}
```

Output:



The screenshot shows the Turbo C++ IDE window titled "C:\ Turbo C++ IDE.lnk". The output window displays the following text:

```
Enter the array size:5
Enter the elements into the array
10
20
30
40
50

The array elements are...
10    20    30    40    50
```

Pointers with String:

We can also use pointers with strings to perform any type of string manipulation in fact every string is also gives address by itself. Hence while dealing with pointer strings no need to take the help of address operator (&). We can declare a string pointer as follows.

Syntax: char *stringvariablename;

Eg: char *s;

The following program illustrates the use of pointers in strings.

```
#include<stdio.h>

#include<conio.h>

void main()
{
    char *s;

    clrscr();

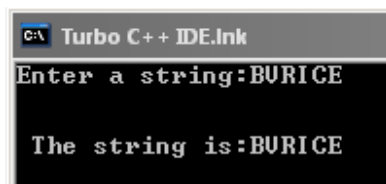
    printf("Enter a string:");

    scanf("%s",s);

    printf("\n\n The string is:%s",s);

    getch();
}
```

Output:



```
C:\ Turbo C++ IDE.Ink
Enter a string: BURICE

The string is: BURICE
```

Void pointer:

A pointer which is defined with the keyword “void” is known as “void pointer”. We cannot access void pointer without having explicit type casting, this is because the compiler cannot understand the size of the pointer that points to another variable. In fact we have void pointer but we cannot have void variables.

Syntax: void *variable_name;

Ex:

```
void *p;
```

```
int a;
```

If we want to convert the void pointer as integer pointer we have to make type casting to the data type integer.

i.e., p=&a; *(int *)p=20;

For char data types p=&c;

*(char *)p='a';

The following illustrates the use of void pointer in c.

Aim: To write a C program that illustrates the use of void pointer in C.

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

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
    void *p;
```

```
    int a=10;
```

```
    clrscr();
```

```
    p=&a;
```

```
    printf("\n\n a=%d",*(int*)p);
```

```
    getch();
```

```
}
```

Output:

a=10.

Parameter passing techniques:

Parameter passing techniques are two types. They are

1. Call by value. (Or) pass by value
2. Call by reference. (Or) pass by reference.

Call by value or Pass by value:

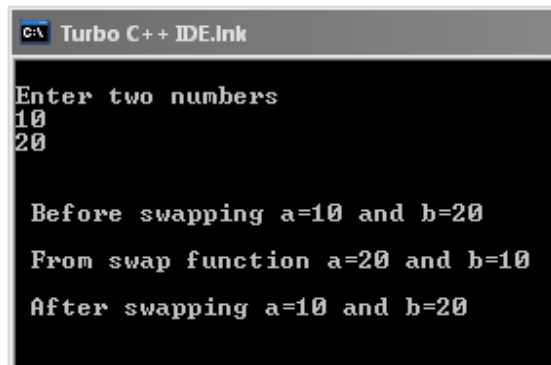
When a function is called by passing values then it is called as “call by value (or) pass by value”. In this any changes that has been made with the formal arguments will not effect to the actual parameters (Or) actual arguments. This is because the scope of the variables restricted from one function to another function.

The following program illustrates the concept of Call by value in C.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    void swap(int,int);
    int a,b;
    clrscr();
    printf("\nEnter two numbers\n");
    scanf("%d%d",&a,&b);
    printf("\n\n Before swapping a=%d and b=%d",a,b);
    swap(a,b);
    printf("\n\n After swapping a=%d and b=%d",a,b);
    getch();
}
```

```
void swap(int a,int b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
    printf("\n\n From swap function a=%d and b=%d",a,b);
}
```

Output:

A screenshot of the Turbo C++ IDE window titled "C:\ Turbo C++ IDE.Ink". The output window displays the following text: "Enter two numbers", followed by "10" and "20" on separate lines. Then, it shows "Before swapping a=10 and b=20", "From swap function a=20 and b=10", and "After swapping a=10 and b=20" on separate lines.

```
C:\ Turbo C++ IDE.Ink
Enter two numbers
10
20

Before swapping a=10 and b=20
From swap function a=20 and b=10
After swapping a=10 and b=20
```

Call by Reference:

In the call by value any changes made to formal arguments will not effected to the actual arguments. This is the one of the disadvantage of call by value. This disadvantage is overcome with call by reference.

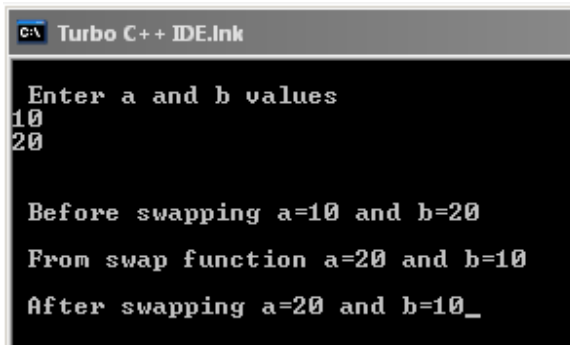
In call by reference instead of sending value to the called function, reference is send to the called function. So the changes made to the formal arguments will also effect to the actual arguments. This is because the manipulation has been done through the address (or) via address. In this way we can avoid the disadvantage of call by value.

The following example explains about call by reference

Aim: To write a C program that illustrates the use of call by reference in C.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    void swap(int *, int *);
    int a,b;
    clrscr();
    printf("\n Enter a and b values\n");
    scanf("%d%d",&a,&b);
    printf("\n\n Before swapping a=%d and b=%d",a,b);
    swap(&a,&b);
    printf("\n\n After swapping a=%d and b=%d",a,b);
    getch();
}
void swap(int *a,int *b)
{
    int *temp;
    *temp=*a;
    *a=*b;
    *b=*temp;
    printf("\n\n From swap function a=%d and b=%d",*a,*b);
}
```

Output:

A screenshot of the Turbo C++ IDE window titled "C:\ Turbo C++ IDE.lnk". The console output shows the program's execution: it prompts for "Enter a and b values", where "10" and "20" are entered. It then displays "Before swapping a=10 and b=20", followed by "From swap function a=20 and b=10", and finally "After swapping a=20 and b=10_".

```
C:\ Turbo C++ IDE.lnk
Enter a and b values
10
20

Before swapping a=10 and b=20
From swap function a=20 and b=10
After swapping a=20 and b=10_
```

Dynamic Memory Allocation (DMA):

Create memory at runtime by the programmer is known as “Dynamic Memory Allocation”. In static memory allocation some amount of space is unused. This unused space is empty. But it is filled with garbage values.

To avoid this problem, DMA technique was introduced. DMA allows allocating memory at run time, so no memory will be wasted. Here we are using different functions for manipulate memory. They are

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

malloc():

This function is used to allocate memory space in the form of bytes to the variables that variables are of different data types. This functions reserves size of bytes for the variables and returns a pointer to some amount of memory size. The memory pointed to will be on the heap, not on the stack, so make sure to free it when you are doing this.

Syn: pointer variable=(data type*) malloc (given size);

Ex: ptr=(int*) malloc (n);

The following program illustrated about malloc()

Aim: To write a C program that allocates the memory dynamically.

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
void main()
{
    int *a,n,i;
    clrscr();
    printf("\nEnter the size of the array:\n");
    scanf("%d",&n);
    //Allocating memory for n variables
    a=(int *)malloc(n*sizeof(int));
```

```

//Reading array elements
printf("\nEnter array elements\n");
for(i=0;i<=n-1;i++)
{
    scanf("%d",&a[i]);
}
//Printing array elements
printf("\nThe array elements are...\n");
for(i=0;i<=n-1;i++)
{
    printf("%d\t",a[i]);
}
getch();
}

```

Output:

```

Turbo C++ IDE Ink
Enter the size of the array:
5
Enter array elements
2
10
23
12
32
The array elements are...
2      10      23      12      32

```

calloc ():

This function is useful for allocating multiple blocks of memory. It is declared with two arguments. This function is usually used for allocating memory for array and structure. The calloc () can be used in place of malloc () function. and returns starting address of the memory to the pointer variable.

Syn: pointer variable= (datatype*) calloc (n,2);

Ex: ptr= (int*) calloc (4,2);

the above declaration allocates 4 blocks of memory, each block contains 2 bytes.

The following program illustrates calloc()

Aim: To write a C program that uses calloc () for DMA.

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

void main()

{

    int *a,n,i;

    clrscr();

    printf("\nEnter the size of the array:");

    scanf("%d",&n);

    //allocating memory using calloc()

    a=(int *)calloc(n,2);

    printf("\nEnter the elements into the array\n");

    for(i=0;i<=n-1;i++)

    {

        scanf("%d",(a+i));

    }

    printf("\nThe array elements are\n");

    for(i=0;i<=n-1;i++)

    {

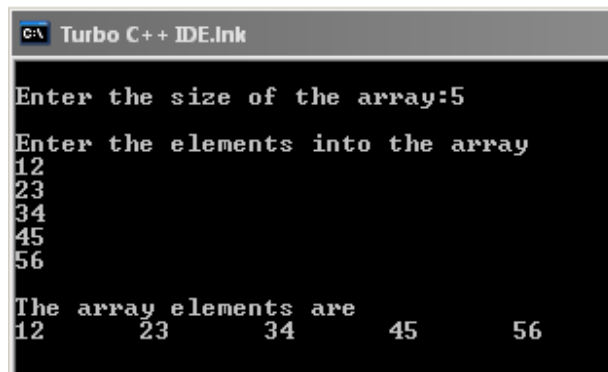
        printf("%d \t",*(a+i));

    }

    getch();

}
```

Output:

A screenshot of the Turbo C++ IDE window. The title bar reads 'C:\ Turbo C++ IDE.Ink'. The main window has a black background with white text. It shows a program that asks for the size of an array (5) and then for five elements (12, 23, 34, 45, 56). It then displays the array elements in a formatted output.

```
C:\ Turbo C++ IDE.Ink
Enter the size of the array:5
Enter the elements into the array
12
23
34
45
56
The array elements are
12      23      34      45      56
```

realloc():

This realloc() function is used to reallocate the memory i.e., enlarge the previously allocated memory by malloc() or calloc() functions. It returns the address of the reallocated block, which can be different from the original address. If the block cannot be reallocated realloc () returns NULL.

Syn: pointer_name=(data_type *) realloc(pointer_name, new_size);

The following program illustrates about realloc() function

Aim: To write a C program using realloc()

```
#include<stdio.h>

#include<conio.h>

#include<alloc.h>

void main()
{
    char *s;

    clrscr();

    //allocating memory for string 's'

    s=(char *)malloc(6);

    s="BVRICE";

    printf("\nBefore reallocating string:%s",s);
```

```
//reallocating memory of string 's'

s=(char *)realloc(s,15);

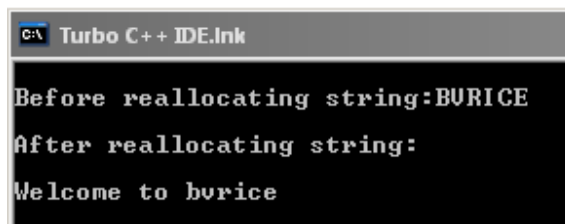
s="\n\nWelcome to bvrice";

printf("\n\nAfter reallocating string:%s",s);

getch();

}
```

Output:

A screenshot of the Turbo C++ IDE window. The title bar reads "C:\ Turbo C++ IDE.Ink". The main window area has a black background with white text. It displays the output of a program: "Before reallocating string: BURICE", "After reallocating string:", and "Welcome to bvrice".

```
C:\ Turbo C++ IDE.Ink

Before reallocating string: BURICE
After reallocating string:
Welcome to bvrice
```

free():

The free() function is used to release the memory allocated by memory allocating functions. Thus, using this function wastage of memory is prevented.

Syn: free(pointer variable);

The following program illustrates about free()

Aim: To write a C program that uses free()

```
#include<stdio.h>

#include<conio.h>

#include<alloc.h>

void main()

{

    char *s;

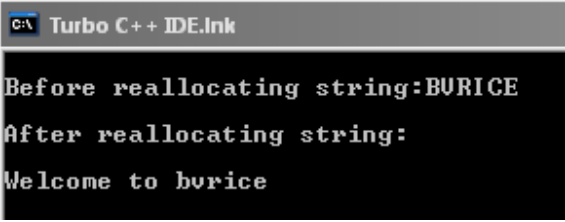
    clrscr();

    //allocating memory for string 's'

    s=(char *)malloc(6);
```

```
s="BVRICE";  
  
printf("\nBefore reallocating string:%s",s);  
  
//reallocating memory of string 's'  
  
s=(char *)realloc(s,15);  
  
s="\n\nWelcome to bvrice";  
  
printf("\n\nAfter reallocating string:%s",s);  
  
free(s);  
  
getch();  
  
}
```

Output:



```
C:\ Turbo C++ IDE.Ink  
  
Before reallocating string:BVURICE  
After reallocating string:  
Welcome to bvrice
```