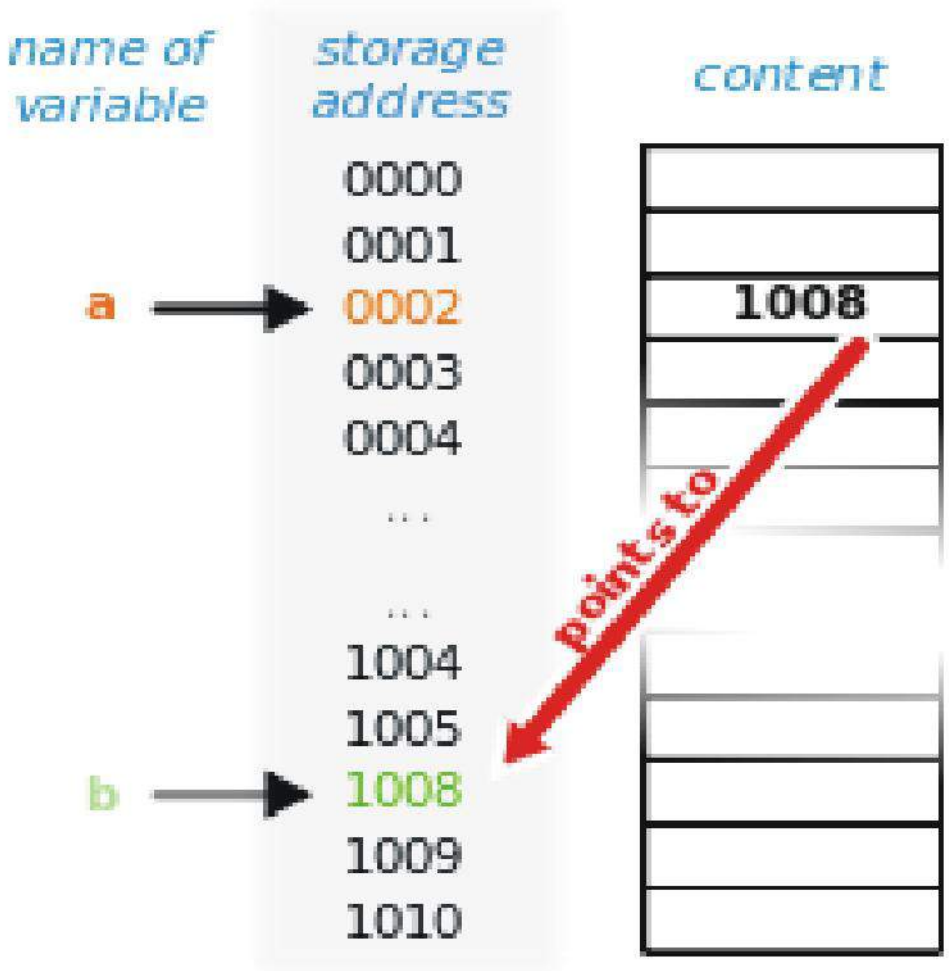
CHAPTER-11

POINTERS

Objectives:

- To understand the concepts of pointers.
- Usage of pointers.
- The role of pointers in array, strings, structures.
- Concepts of dynamic and static allocation of memory.
- Relationship between pointers and functions.
- Relationship between pointers and objects.



I11.1ntroduction:

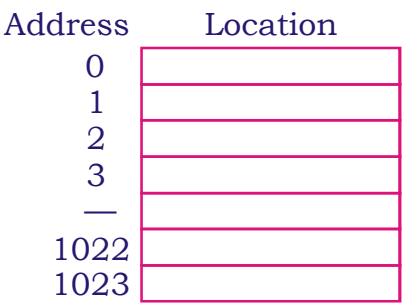
When writing a program, you declare the necessary variables that you will need in order to accomplish your work. When declaring variables, you are simply asking the computer to reserve a set amount of space in its memory for a particular object you want to use. When you declare a variable, the computer reserves an amount of space for that variable, and uses the variable's name to refer to that memory space. This will allow you to store the value of that variable, in that space. Indeed, the computer refers to that space using an address. Therefore, everything you declare has an address, just like the address of your house. You can find out what address a particular variable is using.

Pointers are a powerful concept in C++ and have the following advantages.

- It is possible to write efficient programs.
- Memory is utilized properly.
- Dynamically allocate & deallocate memory.
- Easy to deal with hardware components.
- Establishes communication between program and data.

I12.Memory representation of pointers.

Before understanding the concept of pointers it is necessary to know the memory organization. Memory is organized as an array of bytes. A byte is basic storage and accessible unit in memory. Each byte is identifiable by a unique number called address. Suppose we have 1KB of memory, since 1KB=1024 bytes, the memory can be viewed as an array of locations of size 1024 with the subscript range (0 to 1023). 0 represents the address of first location; 1 represents the address of second location; and so on 1023 represents the address of last location.



We know that variables are declared before they are used in a program. Declaration of a variable tells the compiler to perform the following.

- Allocate a location in memory. The number of location depends on data type.
- Establish relation between address of the location and the name of the variable.

Consider the declaration, int num;

This declaration tells the compiler to reserve a location in memory. We know that the size of int type is two byte. So the location would be two bytes wide.

Address	num
100	15
101	

In the above figure, num is the variable that stores the value 15 and address of num is 100. The address of a variable is also an unsigned integer number. It can also be retrieved and stored in another variable.

Pointer:

A pointer is a variable that holds a memory address, usually the location of another variable in memory.

11.3 Declaration and Initialization of pointer:

The general form is, data-type *variable_name;

data-type is any valid data type supported by C++ or any user defined type and variable_name is the name of pointer variable. The presence of * indicates that it is a pointer variable.

Defining a Pointer Variable:

```
int *iptr;      iptr is declared to be pointer variable of int type.
float *fptr;    fptr is declared to be pointer variable of float type.
char *cptr;     cptr is declared to be pointer variable of character type.
```

Pointer Variables Assignment:

We can assign the address of a variable to a pointer variable as follows:

```
int   num = 25;
int   *iptr;
iptr = &num;
```

In the above example, the variable num (=25) is assigned to pointer variable iptr.

11.4 The address-of operator (&):

& is a unary operator that returns the memory address of its operand. For example, if var is an integer variable, then &var is its address. This operator has the same precedence and right-to-left associativity as the other unary operators.

You should read & operator as “the address-of” which means &var will be read as “the address of var”.

Example: int num = 25;

```
int    *iptr;
iptr = &num;              // The Address of Operator &
```

11.5 Pointer operator or Indirection Operator (*):

The second operator is Indirection Operator *, and it is the complement of &. It is a unary operator that returns the value of the variable located at the address specified by its operand.

Example:

```
int     num = 25;
int     *iptr;              //Pointer operator (Indirection Operator *):
iptr = &num;
```

The following program executes the above two operations

```
#include <iostream>
#include <iomanip.h>
void main( )
{
        int       var;
        int   *ptr;
        int       val;

        var = 3000;
        ptr = &var;
        val = *ptr;

        cout << "Value of var: " << var << endl;
        cout << "Value of ptr: " << ptr << endl;
        cout << "Value of val: " << val << endl;

}
```

```
Value of var: 3000
Value of ptr: 0xbff64494
Value of val: 3000
```

11.6 Pointer Arithmetic:

As you understood, pointer is an address which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value.

There are four arithmetic operators that can be used on pointers: ++, —, +, and . (dot operator).

Following operations can be performed on pointers.

- We can add an integer value to a pointer.
- We can subtract an integer value from a pointer,

- We can compare two pointers, if they point the elements of the same array
- We can subtract one pointer from another pointer if both point to the same array.
- We can assign one pointer to another pointer provided both are of same type.

Following operations cannot be performed on pointers.

- Addition of two pointers.
- Subtraction of one pointer from another pointer when they do not point to the same array.
- Multiplication of two pointers.
- Division of two pointers.

Example:

a. Suppose if p is an integer pointer then p++ will increment p by 2 bytes. Each time a pointer is incremented by 1, it points to the memory location of the next element of its base type.

b. Suppose if p is a char pointer then p++ will incremented p by 1-byte.

c. p-- each time a pointer is decremented by 1, it points to the memory location of the previous element of its base type.

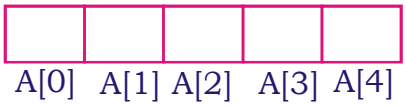
d. p=p + integer value.
p=p - integer value.

11.7 Pointers and Arrays:

There is a close relationship between arrays and pointers in C++.

Consider the declaration. int a[6];

The elements of the array can be referred to in the program as a[0], a[1], …. , a[9]. When the program is compiled, the compiler does not save the addresses of all the elements, but only the address of the first element, a[0]. When the program needs to access any element, a[i], it calculates its address by adding i units to the address of a[0]. The number of bytes in each "unit" is, in our example, equal to the sizeof(int). i.e., 2. In general, it is equal to the number of bytes required to store an element of the array.



The address of a[0] can be explicitly obtained using the & (address-of) operator. i.e., &a[0]. Since the data type of a[0] is int, the data type of &a[0] is, as usual, int* (pointer to int).

C++ allows us to use the name of the array a, without any subscript, as another name for &a[0].

The following example shows the relationship between pointer and one-dimensional array.

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main( )
{
        int     a[10], i, n;

        cout<<"How many elements?  ";
        cin>>n;
        cout<<"Enter array elements: ";
        for(i=0; i<n; i++)
                cin>>*(a+i);

        cout<<"The given array elements are ";
        for(i=0; i<n; i++)
                cout<<setw(4)<<*(a+i);

        getch();

}
```

```
How many elements?  5
Enter array elements:  1 2 3 4 5
The given array elements are 1  2  3  4  5
```

11.8 Array of pointers:

As we know that there is an array of integers, array of float, similarly, there can be an array of pointers. Since we know that pointer is a variable which stores address of another variable, an array of pointers means that it is a collection of addresses.

The example below shows the array of pointers.

```
int     *iptr[5];
int     i=10, j=20, k=30, l=40, m=50;

iptr[0] = &i;         *iptr[0] = 10;
iptr[1] = &j;         *iptr[1] = 20;
iptr[2] = &k;         *iptr[2] = 30;
iptr[3] = &l;         *iptr[3] = 40;
iptr[4] = &m;         *iptr[4] = 50;
```

11.9 Pointers and Strings:

We have already discussed that there is a close relationship between array and pointers. Similarly there is also a close relationship between strings and pointers in C++. String is sequence of characters ends with null ('\0') character. Suppose we have declared an array of 5 elements of the data type character.

```
char   s[5];
char   *cptr;
cptr = s;
```

Here, s is array of characters (strings). cptr is character pointer to string. s also represents character pointer to string.

The elements of the array can be referred to in the program as s[0], s[1], …. , s[5]. When the program is compiled, the compiler does not save the addresses of all the elements, but only the name of the array. Here, s gives the base address of the array. i.e., the address of the first character in the string variable and hence can be regarded as pointer to character. Since we know that string always end with null character, it is enough for us to know the starting address of a string to be able to access entire string. The number of bytes allocated for a string is determined by the number of characters within string.

Let us now consider a string constant "HELLO". **s** is pointer to the memory location where 'H' is stored. Here, **s** can be viewed as a character array of size 6, the only difference being that **a** can be reassigned another memory location.

```
char s[5] = "Hello";
```

H	E	L	L	O	\0
s[0]	s[1]	s[2]	s[3]	s[4]	s[5]

Here, s gives address of 'H'.
*a gives 'H'
a[0] gives 'H'
a++ gives address of 'E'
*a++ gives 'E'

11.10 Pointers as Function Parameters.

A pointer can be a parameter. It works like a reference parameter to allow change to argument from within the function.

```
void swap(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
swap(&num1, &num2);
```

11.11 Pointers and Structures

We can create pointers to structure variables.

```
struct student
{
        int     rollno;
        float   fees;
};

student       s;
student       *sp = &s;
(*sp).rollno = 104;
```

The above statements can be written using the operator -> as

```
        ptr -> member:
        sp -> rollno = 104;
```

11.12. Memory allocation of pointers (Dynamic and Static)

The compiler allocates the required memory space for a declared variable. For example, integer variable it reserves 2- bytes, float variable it reserves 4-bytes, character variable it reserves 1-byte and so on. Therefore every data and instruction that is being executed must be allocated some space in the main or internal memory. Memory allocation is done in two ways:

- Static allocation of memory
- Dynamic allocation of memory

11.12.1 Static allocation of memory

In the static memory allocation, the amount of memory to be allocated is predicted and pre known. This memory is allocated during the compilation itself. All the variables declared normally, are allocated memory statically.

```
Example:    int     a;      //Allocates 2 bytes of memory space during the
                            //compilation time.
```

11.12.2 Dynamic allocation of memory (new and delete)

In the dynamic memory allocation, the amount of memory to be allocated is not known. This memory is allocated during run-time as and when required.

C++ supports dynamic allocation and deallocation of objects using the **new** and **delete** operators. These operators allocate memory for objects from a pool called the **free store**. The new operator calls the special function operator **new** and the delete operator calls the special function operator **delete**.

We can allocate storage for a variable while program is running by using new operator. Dynamic allocation is perhaps the key to pointers. It is used to allocate memory without having to define variables and then make pointers

point to them. Although the concept may appear confusing, it is really simple. The following codes demonstrate how to allocate memory for different variables.

To allocate memory of type integer, int *iptr = new int;

```
int *pNumber;
pNumber = new int;
```

The first line declares the pointer, pNumber. The second line then allocates memory for an integer and then makes pNumber point to this new memory. Here is another example, this time using a double:

```
double *pDouble;
pDouble = new double;
```

To allocate memory for array, double *dptr = new double[25];

To allocate dynamic structure variables or objects,

student sp = new student; //student is tag name of structure

The formula is the same every time, so you can't really fail with this bit. What is different about dynamic allocation, however, is that the memory you allocate is not deleted when the function returns, or when execution leaves the current block. So, if we rewrite the above example using dynamic allocation, we can see that it works fine now:

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
int *p;
void SomeFunction()
{
   // make p pointer point to a new integer
   p = new int;
   *p = 25;
}

void main()
{
   SomeFunction();                    // make pPointer point to something
   cout<<"Value of *p: "<<*p;
}
```

Output | Value of *p: 25

When SomeFunction is called, it allocates some memory and makes p point to it. This time, when the function returns, the new memory is left intact, so p still points to something useful.

delete pointer:

Memory that is dynamically allocated using the NEW operator can be freed using delete operator. THe delete operator calls the operator delete functio, which frees memory back to the avalable pool.

Releasing Dynamic Memory

Use delete function to free dynamic memory as: delete iptr;

To free dynamic array memory, delete [ ] dptr;

To free dynamic structure, delete student;

Static allocation of memory	Dynamic allocation of memory
Memory is allocated before the execution of the program begins. (During Compilation)	Memory is allocated during the execution of the program.
No memory allocation or deallocation actions are performed during Execution.	Memory Bindings are established and destroyed during the Execution.
Variables remain permanently allocated.	Allocated only when program unit is active.
Implemented using stacks and heaps.	Implemented using data segments.

11.13 Free store(heap memory)

Free store is a pool of unallocated memory heap given to a program that is used by the program for dynamic allocation during execution.

11.14 Memory Leak

If the objects, that are allocated memory dynamically, are not deleted using delete, the memory block remains occupied even at the end of the program. Such memory blocks are known as orphaned memory blocks. These orphaned memory blocks when increase in number, bring adverse effect on the system. This situation is called memory leak.

11.15 Self Referential Structure

The self referential structures are structures that include an element that is a pointer to another structure of the same type.

```
struct node
{
        . . .    . . . . .
        . . . .  . . . . .
```

11.16 Pointers and functions

A function is named unit of a group of program statements designed to perform a specific task and returns single value. There is a close relationship between pointers and functions. We know that a function uses arguments in order to carry its assignment. The arguments are usually provided to the function. When necessary, a function also declares its own variable to get the desired return value. Like other variables, pointers can be provided to a function, with just a few rules. When declaring a function that takes a pointer as an argument, make sure you use the asterisk for each argument. When calling the function, use the references to the variables. The function will perform its assignment on the referenced variable(s). After the function has performed its assignment, the changed value(s) of the argument(s) will be preserved and given to the calling function. To pass pointer arguments, use the asterisks when declaring the function and use the ampersand (&) when calling the function.

Invoking of function can be done by following two methods:

- By passing the references.
- By passing the pointers.

11.16.1. Invoking functions by passing the references

When parameters are passed to the functions by reference, the formal parameters become reference (or aliases) to the actual parameters in the calling function. This means that invoking the called function does not create its own copy of original values, rather than, it refers to the original values by different names i.e., their references. Thus the called function works with the original data and any change in the values gets reflected to the data.

The call by reference method is useful in situation where the values of the original variable are to be changed using a function. Say, for instance a function is to be invoked that swap two variables that are passed by references. The following example program explains it.

Program to swap the values of two variables using pass-by-reference method:

```
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
void main()
{
```

```
        void    swap(int &, int &);
        int     a=10, b=30;

        cout<<"Original values:  ";
        cout<<"a = "<<a<<" and b = "<<b<<endl;
        swap(a,b);
        cout<<"values after swapping: ";
        cout<<"a = "<<a<<" and b = "<<b<<endl;
        getch();
}
void swap( int &x , int &y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
void swap( int &x , int &y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

Original values: a = 10 and b = 30
Values after swapping: a = 30 and b = 10

In the above program the function swap( ) creates reference x for first incoming integer and reference y for second incoming integer. Thus the original values are worked with, but by using the names x and y. Notice that the, function call statement is simple one. i.e., swap(a, b);

But the function declaration (prototype) and definition include the reference symbol &. The function declaration and definition, both start as: void swap(int &x , int &y)

Therefore, by passing the references the function works with the original values (i.e., the same memory area in which original values are stored) but in case alias names to refer to them. Thus the values are not duplicated. The same happens when pointers are passed but in different manner.

11.16.2. Invoking functions by passing the pointers

When the pointers are passed to the function, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using formal arguments (the addresses of original

values) in the called function, we can make changes into the actual arguments of the calling function, therefore here also, the called function does not create own copy of original values rather, it refers to the original values by the addresses(passed through pointers) it receives.

To swap two values, we have seen how the passing references method works. The same can be achieved by passing addresses through pointers. The following example program explains it.

Program to swap values of two variables using pass by references method:

```
#include <iostream.h>
#include <conio.h>
void main()
{
        void   swap(int *x int *y);
        int    a=10,b=30;

        cout<<"Original values: ";
        cout<<"a = "<<a<<" and b="<<b<<endl;

        swap(&a, &b);
        cout<<"values after swapping: ";
        cout<<"a = "<<a<<" and b = "<<b<<endl;
        getch();
}

void swap( int *x , int *y)
{
        int temp;

        temp= *x;
        *x = *y;
        *y = *temp;
}
```

```
        Original values: a = 10 and b = 30
        Values after swapping: a = 30 and b = 10
```

The above program invokes swap( ) by passing addresses of a and b. i.e., swap(&a , &b); Here, &a and &b pass the addresses of a and b respectively.

The function definition receives the incoming addresses in corresponding pointers x and y. Notice the function declaration.

```
void swap( int *x, int *y);
```

Thus, we have seen that using call-by-reference, in both ways, we are able to return more than one value at a time (we sent back changed values of two

variables a and b), which are not possible using ordinary return statement. A return statement can return only one value from a function at a time.

11.17 Memory Comes, Memory Goes

There's always a complication and this one could become quite serious, although it's very easy to remedy. The problem is that although the memory that you allocate using dynamic allocation is conveniently left intact, it actually never gets deleted automatically. That is, the memory will stay allocated until you tell the computer that you've finished with it. The upshot of this is that if you don't tell the computer that you've finished with the memory, it will be wasting space that other applications or other parts of your application could be using. This eventually will lead to a system crash through all the memory being used up, so it's pretty important, freeing the memory when you've finished with it is very simple:

11.18 Pointers and objects

As we know that there is pointer to variables, pointer to strings, pointer to structures, similarly there is pointer to objects. The pointers pointing to objects are referred to as object pointers.

Declaration of pointers to objects

class_name *object-pointer;

Here, class_name is the name of an already defined class and object-pointer is the pointer to an object of this classtype.

Example: employee *eptr;

Here, employee is an already defined class. When accessing members of a class using an object pointer, the arrow operator (->) is used instead of dot (.) operator.

The following program illustrates how to access an object given a pointer to it.

```
#include<iostream.h>
#include <iomanip.h>
#include<conio.h>
class emp
{
        private:
                int     empno;
                char    name[20];
                float   salary;
        public:
                void    get();
                void    display();
```

```
};
void emp::get()
{
        cout<<"Enter employee number: ";
        cin>>empno;
        cout<<"Enter employee name: ";
        cin>>name;
        cout<<"Enter employee salary: ";
        cin>>salary;
}
void emp::display()
{
        cout<<"Employee number: "<<empno<<endl;
        cout<<"Employee name: "<<name<<endl;
        cout<<"Employee salary: "<<salary;
}
void main( )
{
        emp e, *ep;
        ep = &e;
        clrscr( );
        ep->get( );
        ep->display( );
        getch( );
}
```

```
Enter employee number: 2505
Enter employee name: Harshini
Enter employee salary: 6500.00
Employee number: 2505
Employee name: Harshini
Employee salary: 6500.00
```

The given program is self referential. Here, *ep is pointer to an object.

11.19 this pointer

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a this pointer, because friends are not members of a class. Only member functions have this pointer.

Points to Remember:

- Pointers are a powerful concept in C++ and have following advantages.
  - It is possible to write efficient programs
  - Memory is utilized properly
  - Dynamically allocate & de allocate-memory
  - Easy to deal with hardware components
  - Establishes communication between program and data
- A pointer is a variable that holds a memory address, usually the location of another variable in memory.
- Following operations that can be performed over pointers.
  - We can add an integer value to a pointer.
  - We can subtract an integer value from a pointer,
  - We can compare two pointers. if they point the elements of the same array
  - We can subtract one pointer from another pointer if both point to the same array.
- Following operations that cannot be performed over pointers.
  - Addition of two pointers
  - Subtraction of one pointer from another pointer when they do not point to the same array
  - Multiplication of two pointers
  - Division of two pointers
- There is a close relationship between arrays and pointers in C++.
- C++ allows us to use the name of the array a , without any subscript, as another name for &a[0].
- Array of pointers means that it is a collection of address.
- There is also a close relationship between strings and pointers in C++.
- A pointer can be a parameter. It works like a reference parameter to allow change to argument from within function
- We can create pointers to structure variables
- In the static memory allocation, the amount of memory to be allocated is predicted and pre known.

- In the dynamic memory allocation, the amount of memory to be allocated is not known. This memory is allocated during run-time as and when required.
- We can allocate storage for a variable while program is running by using new operator.
- Use delete to free dynamic memory.
- Free store is a pool of unallocated heap memory given to a program that is used by the program for dynamic allocation during execution.
- Memory leak: If the objects, that are allocated memory dynamically, are not deleted using delete, the memory block remains occupied even at the end of the program. Such memory blocks are known as orphaned memory blocks. These orphaned memory blocks when increase in number, bring adverse effect on the system. This situation is called memory leak
- The self referential structures are structures that include an element that is a pointer to another structure of the same type.
- There is a close relationship between pointers and functions. We know that a function uses arguments in order to carry its assignment.
- Invoking of function can be done by following two methods.
  - By passing the references
  - By passing the pointers
- The pointers pointing to objects are referred to as object pointers.
- Every object in C++ has access to its own address through an important pointer called this pointer.

Review Questions

One marks questions.

1. What do you mean by pointer?.
2. Mention any one advantage of pointer?
3. What is address operator?
4. What is pointer operator?
5. How to declare pointer?
6. How to initialize pointer?
7. What is static memory?
8. What is dynamic memory?
9. What is free store?
10. Write a definition for a variable of type pointer to float.
11. What is new operator in C++?
12. What is delete operator in C++?

Two marks questions.

1. What do you mean by pointer? Explain with example.
2. Mention any 2 advantages of pointer?
3. What is address operator? Give example.
4. What is pointer operator? Give example.
5. How to declare pointer? Give example.
6. How to initialize pointer? Give example.
7. What is static memory?
8. What is dynamic memory?
9. What is free store?
10. Illustrate the use of “self referential structures” with the help of example.
11. What is new operator in C++?
12. What is delete operator in C++?
13. What is array of pointers? Give example.

Three marks questions:

1. What are the advantages of pointer?
2. How dynamic memory allocation is different from static memory allocation.
3. What is new operator in C++? Give example.
4. What is delete operator in C++? Give example.
5. Show the general form new and delete operator in C++?
6. What is array of pointers? Give example.
7. What is the relationship between array and pointers? Give example.
8 What is the relationship between string and pointers? Give example.
9. What is the relationship between structures and pointers? Give example.
10. What is the relationship between object and pointers? Give example.

Five marks questions:

1. Show the general form new and delete operator in C++?
2. What is the relationship between object and pointers? Give example.
3. Explain with example by passing the reference.
4. Explain with example by passing the pointers.

*****