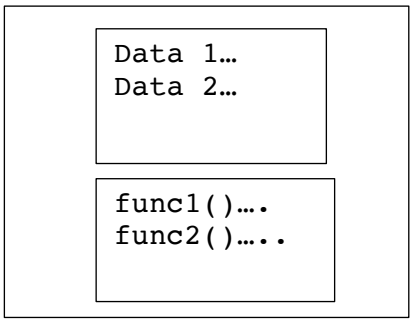
Classes & Objects: Introduction, Class Specification, examples, Class Objects, Access members, defining member functions, data hiding, constructors, destructors, parameterized constructors, static data members, static member functions, scope resolution operator, Passing Objects to Functions, Returning Objects, Object Assignment
Pointers and dynamic memory allocation: Pointers, Pointer as function arguments, Dynamic Allocation Operators new and delete, Initializing Allocated Memory, Initializing Allocated Memory, Allocating Arrays, Allocating Objects

2.1 Classes and objects: Introduction

- A class is a collection of objects having identical attributes and common behaviour (operations).
- A class encloses both the data and function into a single unit.



class groups data and functions as single unit

- Class is a user defined data types. Once a class been declared, the programmer can create a number of objects associated with that class.
- Defining objects of a class data type is known as a class instantiation.

2.2 Class Specification

- **Definition:** A Class is a user defined data type, which binds data and functions together.

2.2.1 Features of Class

- A class is an example of OOPs concept, encapsulation.
- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions that manipulate these variables.
- Class in C++ are similar to structures in C, the only difference being, class defaults to private access control, whereas structure defaults to public.

2.2.2 General form of class is:

```
class className
{
   private:
      Data_members;
      Member_Functions;
   public:
      Data_members;
      Member_Functions;
}; object_list
```

- Functions declared within the class as known as *member functions* or *methods* and the variables declared within the class are *data members* or *member variables*.
- By default, functions and data declared within the class are *private* to that class and can be accessed only through the member functions of that class. That is, the data and the functions those operate on the data and strongly binded together providing the data security(encapsulation mechanism).
- By specifying the access specifier as *public* we can make data and functions to be accessible outside the class.

2.3 Examples

- A typical class declaration would look like:

```
class Student {

      int rollNo; char name[10];
   public:
      void read( ) {
            cout<< "Enter Student Roll Number";
            cin >> rollNo;
            cout<< "Enter Student Name";
            cin >> name;
            }
      void show( ) {
            cout<< "Student Roll Number = " << rollNo;
            cout<< "Student Name = ">> name;
            }

};
```

2.4 Class Objects

- Definition: object is an instance of a class.
- Object can be treated as a variable of new data type class. Objects can either be created while declaring a class or later.
- For example:

```
class Student
{
      char name[10];
      int age;
      public:
            void read();
} s1, s2 ;      //object creation during class declaration
```

- After declaration of the class, if a programmer wants to create objects, he/she can use class name as a data type name as variable name. the general form
- Class_name v1, v2, v3..,
- For example:
- Student s1, s2;
- Now, s1 and s2 are the objects of a class student. Both s1 and s2 have their own physical memory locations (i.e memory address) separately. That is, the objects will havet heir

own copies of member variables of the class. Hence, every object is different from the other object of the same class.

2.5 Accessing Members

- The public members of class can be accessed through the dot(•) operator.
- The general form:

```
objectName.memberName;
```

- The dot operator is used to access both *member_variables* and *member_functions*.
- Example
  - `s1.age=25;`
  - `s2.marks=88.9;`
  - `s1.read();`
- Here `s1` and `s2` are the objects of `Student` and `age`, `marks`, `read()` are members of class `Student`.

2.6 Defining Member Functions

- Member functions of a class can be defined in any one of the following:
  - Declare and define inside the class
  - Declare and define outside the class

- **Declare and define inside the class**
  - A member function of a class can be defined inside a class. The definition of a member function is within the scope of its enclosing class.
  - The member functions defined inside a class definition are by default *inline functions*

```
Example:
class Demo {
  public:
      void show( ) // by default show is an inline function.
      { cout << "Member function inside class"; }
};
int main( )
{
      Demo ob;
      Ob.show();
}
```

- **Declare and define outside the class**
  - A member function of a class can also be defined outside the class. Member functions defined outside class are non-inline functions.
  - The declaration of a member function is written within the class scope but its definition can be defined outside the class scope using scope resolution operator (**::**).
  - The scope resolution operator (**::**) is used to specify that the function being defined is a member of the given class.
  - *The general form*:

```
returnType className :: functionName(<argument_list>)
     { //statements }
```

- Example

```
class Demo
{
     public:
       void show( );
};
void Demo::show( ) //defining member function outside
      { cout << "Member function out Side class"; }

int main( )
{
      Demo ob;
      Ob.show( );
}
```

- Member function defined outside class can be made inline using keyword **inline.**
- Syntax:

```
inline returnType className :: functionNAme() { //statements }
```

- Example

```
inline void Demo :: show ()
{
      cout << "Member function outSide class";
}
```

- Function show( ) is a member of Demo class.

2.7 Data Hiding

- Definition: It is a process of protecting the data members from outside manipulation.
- Data hiding can be achieved using following:
  - Encapsulation
  - Access Specifiers
- Encapsulation
  - Helps to bind important data under one unit and ensures enhanced Security
  - The data is concealed within a class, so that it cannot be accessed mistakenly by functions outside the class.
  - Example: class

- Access Specifiers
  - Access modifiers are used to implement the concept of Data Hiding. There are 3 types of access modifiers available in C++:
    - Private
    - Public
    - Protected
- *Private*

- Private members are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.
- Data members are generally defined as private.

- *Public*
  - Public members can be accessed by the objects of the same class from any other class or the function outside the class.
  - Member functions are generally defined as public.

- *Protected*
  - Class member declared as protected are inaccessible outside the class but they can be accessed by any subclass (derived class) of that class.

Example:

```
class Point {
  private: int x, y;
  public:
    void setData( )
      { X = 5, y = 6;}

    void showData( ) {
     cout << "X = " << x;
      cout << "Y = " << y;}
};
```

```
int main( )
{
  Point P1;
  P1.x = 20; //Invalid <private data cant be accessed>
  P1.setData();
  P1.showData();
}
```

2.8 Constructors and Destructors

- ***Constructors***
- *Definition*: *Constructor is a special type of member function which will be invoked automatically when the objects are created*.
- **Properties of constructor**
  - Constructors should be declared as public.
  - Constructor has the same name as the class.
  - Constructor is automatically called when the object is created which initialize an object.
  - Constructors do not have return types. (Not even void)
  - Constructors can be overloaded and we can have any number of constructors per class.
  - Constructors cannot be inherited.

Destructors

- Definition: Destructor is a member function which is invoked automatically when the object goes out of its scope
- Properties of Destructor
  - Destructor has the same name as the class.
  - Destructor is automatically called when the object is out of scope.
  - Destructor is used to destroy an object.
  - Destructor has no return types. (Not even void)
  - Destructor cannot be overloaded. We can have only one destructor per class.
  - Destructor does not have any parameters.

- Destructor is used to free the memory that is dynamically allocated by constructors

- *Example: illustration of constructor and destructor*

```
#include<iostream>
using namespace std;
class Test{
      public:
            Test( ) //constructor of the class Test
            {
                  cout<<"Inside constructor"<<endl;
            }
            ~Test( )
            {
                  cout<<"Inside Destructor";
            }
  };
int main(){
      Test t;
      cout<<"Hello";
}

Output:
Inside constructor
Hello
Inside Destructor
```

Example: Operations on stack to illustrate constructor and destructor

```
#include<iostream>
#define MAX 5
using namespace std;
class stack
{
  int st[MAX];
  int top;
  public:
      stack();
      ~stack();
      void push(int i);
      void pop();
};
stack::stack() // constructor to initialize the top
{
      top=-1;
      cout<<"stack intialised ";
}
stack::~stack() //destructor
{
      cout<<"stack destroyed";
}
void stack::push(int item)
{
      if(top==MAX-1) //check for stack overflow
      {
        cout<<"Stack Destroyed"<<endl;
```

```
      }
      top++;
      st[top]=item;
}
void stack::pop(){
      if(top==-1) //check for stack underflow
      {
       cout<<"Stack is empty";
       return;
      }
      cout<<"Deleted item is"<<st[top]<<endl;
      top--;
}
int main()
{
      stack s1, s2; //create two stack objects
      s1.push(11);
      s1.push(12);
      s2.push(21);
      s2.push(22);

      s1.pop();
      s2.pop()
      s2.pop();
}
```

Output:
Stack initialized
Stack initialized
Deleted element is 12
Deleted element is 22
Deleted element is 21
Stack destroyed
Stack destroyed

Thus, we can say that the same member variable of the class will have different values for different objects. And each object is a separate instance of a class independent of another object.

2.9 Types of Constructors

- The constructors can be classified as-
- Default Constructor
- Zero argument constructor
- Copy constructor

Default Constructor

- If a class is not having any constructor, then the compiler provides a constructor by default.
- Internally, this constructor is without any arguments and without any body/definition.
- For example: consider class declaration

```
class Test
{
      int a;
      public:
      void disp();
};
```

- The programmer has not defined any constructor for the above class. Still the compiler would have provided default constructor, that looks like:

```
Test () {       }
```

- Thus, when an object gets created for the class with no visible constructor, the compiler creates and calls this default constructor indicates no action is to be taken when the object is created.

***Note**: Whenever the programmer declares any type of constructor, then the compiler will not add default constructor.*

Zero argument Constructor

- A constructor defined by the programmer without any arguments, but with definition is known as zero-argument constructor.
- For example:

```
class Test
{
     public:
          Test(){
               cout<<"Inside constructor";
          }
};
```

- Here, the constructor has no arguments, but defines some action, which needs to be taken care when an object gets created.

Parameterized Constructor

- Definition: A Constructor with parameters is a parametrized constructor.
- Constructor are meant for automatic initialization. If it is required to initialize a variable of an object to a user-specified value, that value can be passed to the constructor as a parameter.
- Example for parameterised constructor

```
class Test{
      int a,b;
      public:
            Test(int x, int y){
                  A=x;
                  B=y;
            }
            void disp(){
                  cout<<"a ="<<a<<"\t b="<<b;
            }
};
int main(){
      Test t1(2,3);
      Test t2(10,20);
      t1.disp(); //prints  2      4
      t2.disp(); // prints 10     20
}
```

- In main function creates an object `t1` and also calls the constructor with two arguments `t1(2,3)` The values 2 and 3 are passed to respective variables `x` and `y` and then they are assigned to member variable a and b. Similarly for object `t2.`

Note: If a constructor has only one parameter, the initial value can be passed using an assignment operator.

- Example: Constructor with single parameter

```
class Test{
      int a;
      public:
       Test(int x)
       {
            A=x;
       }
       void disp(){
            cout<<"a ="<<a;
      }
};
int main()
{
      Test t1(3);   //value is provided as an argument
      Test t2 =10;   // value provided using assignment operator

      t1.disp();
      t1.disp();
}
```

Copy Constructor

- *Definition*: A Constructor used to create an object using an existing object is known as copy constructor.
- A copy constructor has the following general function prototype:

```
className (className &obj)
{
      //body
}
```

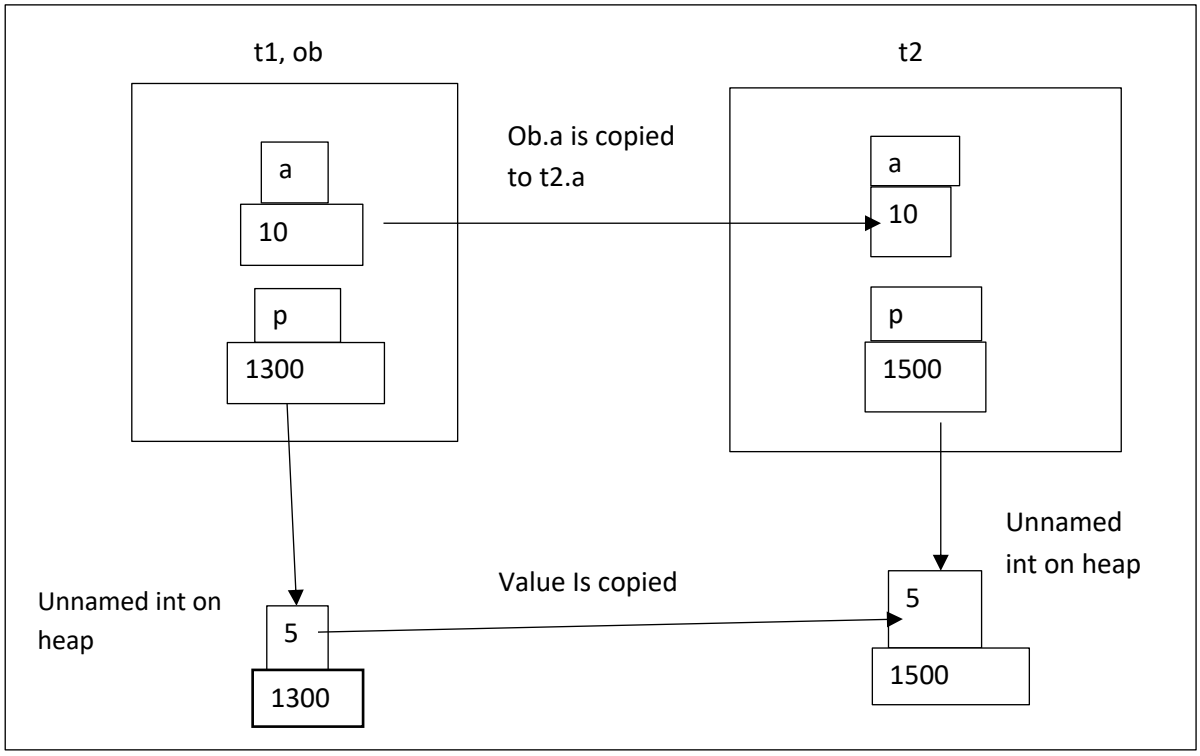
- The copy constructor takes only one argument, which is a reference to the object of same class.

Example on copy constructor

```
#include<iostream>
using namespace std;
class Test{
      int a;
      public:
      Test(int x)
      {
            a=x;
            p=new int(5);
            cout<<"Constructor "<<a<<endl;
      }
      Test(Test &ob)
      {
            a=ob.a;    //copy a as it is
```

```
            p=new int; //create dynamic memory separately
            *p=*(ob.p);      //copy value of dynamic variable

            cout<<"Inside copy constructor"<<endl;
      }
      void disp(){
            cout<<"a ="<<a<<"\t p="<<p<<endl;
      }
      ~Test(){
            cout<<"destructor"<<a<<endl;
            delete p;
      }
};
int main(){
      Test t1(10);
      t.disp();
      {
            Test t2=t1;      //copy constructor will be called
            t2.disp();
      }
      t1.disp();
}
Output:
Constructor 10
a=10  p=0xf5de1  ( address is system dependent and shown in hexa )
inside copy constructor
a=10  p=0x8f4r88
destructor 10
a=10  p=0x8fa10d
destructor 10
```

Memory map for above program



- Memory map for the objects `t1` and `t2` when copy constructor is used
- In the above program, when the statement-

  `Test t2=t1;`
- Is executed, the copy constructor is called. The object `t1` itself is passed as a parameter to copy constructor. The above statement is understood by the compiler as-

  `T2.Test(t1)`
- That is, `t2` is an object calling a constructor `Test` with the argument `t1`. Hence copy constructor parameter may internally seem to be.

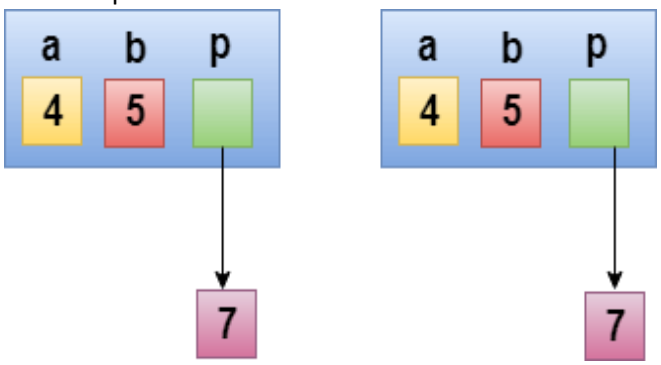
  `Test &ob=t1;`
- Thus, `ob` will act as a reference to the object`t1`. That is, `ob` is just an alias name to t1 and it represents same physical object.

Example 2 :on copy constructor

```
#include <iostream>
using namespace std;
class Demo
{
    public:
    int a;
    int b;
    int *p;

    Demo()
    {
        p=new int;
    }
    Demo(Demo &d)
    {
        a = d.a;
        b = d.b;
        p = new int;
        *p = *(d.p);
    }
    void setdata(int x,int y,int z)
    {
        a=x;
        b=y;
        *p=z;
    }
    void showdata()
    {
        std::cout << "value of a is : " <<a<< std::endl;
        std::cout << "value of b is : " <<b<< std::endl;
        std::cout << "value of *p is : " <<*p<< std::endl;
    }
};
int main()
{
  Demo d1;
  d1.setdata(4,5,7);
  Demo d2 = d1;
  d2.showdata();
  return 0;
}
```

Output:

```
value of a is : 4
value of b is : 5
value of *p is : 7
```



Note: 1. Copy constructor is needed in following situations

- When an object assignment is done
- When an object is passed as an argument to a function
- When an object is returned from a function
  - Here, the last two cases are also, in turn, object assignment only.
- Whenever the programmer opts to overload assignment operator, it is advised to create a copy constructor.

2.10 Static data members

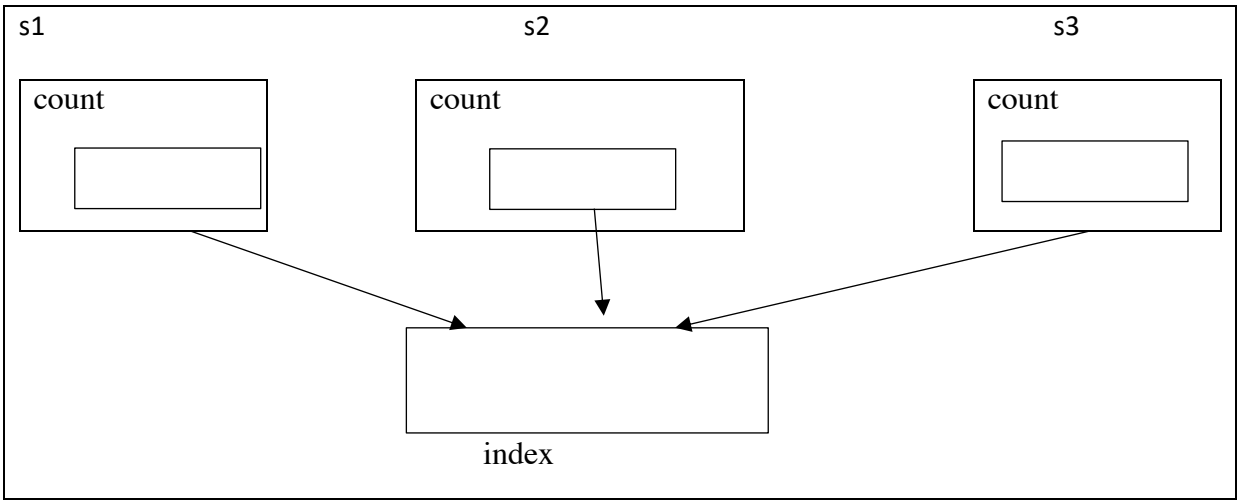
- *Definition: A member variable which is being shared by all objects of that class is known as static member variable.*
- When an object created, it gets its own copy of variables of that class. That is if two objects of one class are created, then each of these objects will own separate copies of the data members of that class. But if we need a member variable so that:
  - Only one copy of that variable must exit
  - All the objects of the class must share the value of that variable
  - Then we have to declare that variable as static.
- When we declare a static data member within class, we are not defining it. Because, class is just a prototype, and memory allocation happens only when an object of that class is created. But static member variables are independent of any object. Hence static member variables are independent of any object.
- Static data member must be defined explicitly outside the class as global definition. This is done using scope resolution operator(**::**).

```
class sample{
     static int index;
     int count;
     public:
          sample(){
               index++;
               count++;
          }
          void display(){
               cout<<"Index="<<index;
               cout<<"count="<<count;
          }
};
int sample :: index=0;   //actual definition of static data
int main(){
     Sample s1;
     s1.display();
     Sample s2;
     S2.display();
     Sample s3;
     s3.display();
}
```

Output

```
Index=1
Count=4567 //garbage value
Index=2
Count=1234 //garbage value
Index =3
Count=1544 //garbage value
```

When an object `s1` is created, the static member variable i.e index gets incremented through the constructor. But, as count is an ordinary integer variable. The objects `s1, s2 and s3` will get separate copies of it and since it has not been initialized, they will have some garbage value. Therefore when we call the function `display()` for `s1`, the value of index is `1` and the value for `count` is some garbage value. When we declare one more object `s2` of the class sample, then again the constructor gets called incrementing the value of `index`. Similarly, for the object `s3` too.



As memory map depicts, there will be different copies of the variable count for each object separately. Whereas, the static variable index is being shared by all the objects. Thus, uniqueness of each object of the same class can only be determined by non -static members are common for all objects.

Note:

1. A static data members is useful when all objects of one class must share a common item of information.
2. A static data member is available only within a class, but it continues to live till end of the program execution.
3. The static data members are defined outside the class. Defining it inside the class declaration would violate the concept that a class is just a blueprint and does not set aside any memory.
4. If you declare static variable inside the class and forgot to define it using the scope resolution operator**(::)** outside the class, then the compiler may pass it. But linking error will occur stating that "*you are trying to reference an undeclared external variable"*

2.11 Static Member Function

- A member function of a class may also be declared as *static*.
- A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.
- But following restrictions are imposed on static member function:
  - They can refer only the static data members
  - A static member function does not have this pointer.
  - There cannot be static and non-static versions of the same function.
  - It may not be a virtual function
  - They cannot be declared as const or volatile functions.

Example: illustration for static member function
// Program to number of object created by particular class

```
class sample{
     static int index;
     public:
          sample(){
               index++;
          }
          static int display(){    //static function
               return index;
          }
};
int sample :: index=0;   //actual definition of static data
int main(){
     cout<<"Now there are"<<sample::display()<<"objects";
     sample s1, s2;
     cout<<"Now there are"<<sample::display()<<"objects";
     sample s3;
     cout<<"Now there are"<<sample::display()<<"objects";
}

Output:
Now there are 0 objects
Now there are 2 objects
Now there are 3 objects
```

This program is used to count the number of objects created for a particular class. In this case `display()` is a static member function that can access only static data member i.e. index. Before creating any objects, the value of index is zero. If two objects `s1` and `s2` are created, then since the constructor gets called twice, the `index` value will be `2` and so on.

2.12 Scope Resolution Operator

- In C++, scope resolution operator is **::**
- Scope resolution operator is used to link a member name with the class. This operator also can be used to access the member in an enclosing scope, which is hidden by a local declaration of same name.
- It is used for following purposes.
  1) To access a global variable when there is a local variable with same name.
  2) To define a function outside a class.
  3) To access a class's static variables.
  4) In case of multiple Inheritance.
  5) For namespace.
  6) Refer to a class inside another class.

- Example: illustration for scope resolution operator

```
#include<iostream>
using namespace std;

int a=20;
int main()
{     int a=30;   //local declaration
      cout<<a;    //prints local variable a as 10
      cout<<(::a);        //access global variable 'a' using :: and it prints 20
}
```

2.13 Passing Objects to Function

- Objects can be treated as variable of the new data type (i.e class). Hence, similar to any other variable, the object also can be passed to a function as a parameter.
- An object can be passed to a function using any of the methods viz. call by value, call by address and call by reference.

```
class Test{
      int a;
      public:
       Test(int x)
        { a=x; }
       void disp(){
         cout<<a;
        }
};
void fun(Test t)    // function taking an object as a parameter
{     t.disp() }
int main(){
      Test ob(5);
      fun(ob);     //pass object as an argument to a function
}
```

Consider class `Test` in the program It contains a member variables `a` and a constructor to initialize that variable and a `disp()` function. There is an independent function `fun()` which takes an object of the class `Test` as an argument. In the main() function an object `ob` is created and then passed to `fun().` Note that here, the function parameter `t` takes the object `ob` as a value (i.e call by value method). Within `fun(),` using the object t, the member function `disp()` is called. So output of this program is 5.

2.14 Returning Objects

- Objects can be passed as a parameter to a function. Similarly, an object can be returned from a function to the calling function.

```
class Test{
      int a;
      Public:
            Test() { }
            Test(int x){
                  a=x;
            }
            void disp(){
                  cout<<a;
            }
};
Test myfun(){           //function whose return type is class Test
      test ob(10);      //function creation and parameterized constructor call
      return ob;        //returning an object
}
int main(){
      Test obj;      // default constructor call
      obj=myfun();  //function returns an object and store in obj
      obj.disp();   // call member function
}
```

In function `myfun(),` an object `ob` is created and its state has been decided by the member variable `a`. Then, it is being returned to the calling funcition `main().` Where object `obj` receives a copy of `ob`. Remember, the object `obj` in the `main()` function was just created using default constructor and was not been initialized. Now as `obj` receives the copy of `ob`, the value of member variable a inside `obj` also gets the value 10. Hence the output of the program would be 10

2.15 Object Assignment

- If two objects are of same type, that is, if they are of same class, then we can assign one object to another. By doing this, we can copy the data of the object at right side of the assignment operator to that at left side.
- Here, the copy of data takes place by bit-by-bit mechanism. That is, value in every member of one object is copied into the respective members of the other object. Such kind of bit-wise copy or element-by element copy is called as *shallow copy*.

```
class Test{
      int a;
      public:
            void put(int x){
                  a=x;
            }
            int get(){
                  return a;
            }
};

int main(){
      Test ob1,ob2;
      ob1.put(25);
      ob2=ob1;
      cout<<"The data in ob2 is: <<ob2.get();
}
Output
The data in ob2 is 25
```

2.16 Pointers

- Pointers is a derived datatype that refers to another variable or object.
- **Syntax**

```
datatype *pointer_variable;
```

- **Properties of pointers/Advantage**
  - Pointer provide direct access to memory
  - Pointers save memory space in call-by-reference
  - Memory can be dynamically allocated or de-allocated using pointers
  - Pointers has different applications like: file handling, stack, linked list, tress etc..,
  - Pointer are type compatibles.

2.17 Pointer as Function Arguments

- Types of calling function
  - Call-by-value
  - Call-by-reference
  - Call-by-pointers
- When we pass pointers as function arguments it actually passes address to function
- Passing pointers to function is called call-by-reference
- This helps to save memory space in function.
- Example: Swap values of two double variables using call-by-pointers as argument.

```
#include <iostream>
using namespace std;
void  swap (double  *x, double  *y)
{  double  temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
int main ()
 {   double  a = 100, b = 20.40;
     swap (&a, &b);

     cout << "**** Output After swap ****"<< endl;
     cout << " a = " << a << endl;
     cout << " b = " << b << endl;
     return 0;
}
```

Pointer to objects

- to refer the address of a variable of any data type, we can have a pointer of that type. Similarly, we can have pointers to store the address of object. To access the pointer members of the class, we will use arrow(**->**) operator instead of dot(**.**) operator.
- Just like any other variable, pointer arithmetic holds good for object pointers. That is increment/decrement by one in pointer points to next/previous object in the array.
- When there is a public member variable in a class, we can have a pointer to that and access the same using pointer directly.

```
#include<iostream>
using namespace std;
class myclass
{
        int a;
    public:
     myclass(){
          a=0;
     }
     myclass(int n){
          a=n;
     }
     int get(){ return a;}
};
main()
{ int k;
  myclass ob1(100), *ptr1,*ptr2;
  myclass ob2[3]={1,2,3};
  ptr1=&ob1;
  ptr2=&ob2;
  cout<<ptr->get();
  for(k=0;k<3;k++){
     cout<<ptr2->get();
     pt2++;}
}
```

this keyword

- A pointer to current object is known as this pointer

Or

- When an object invokes a member function, a pointer to that object is created and passed to the function automatically. Such pointer is known as this pointer.
- Each object gets its own copy of data members and all objects share a single copy of member functions.

```
#include <iostream>
using namespace std;
class Emp {
   public:
        int id;    //data member (also instance variable)
        string name;     //data member(also instance variable)
        float salary;
        Emp(int id, string name, float salary)
         {
              this->id = id;
             this->name = name;
             this->salary = salary;
         }
            void display()
            {
              cout<<id<<" "<<name<<" "<<salary<<endl;
            }
};
int main() {
    Emp e1 = Emp(101,"Sonoo",890000);  //creating an object of Employee
    Emp e2 = Emp(102,"Nakul",59000);   //creating an object of Employee
    e1.display();
    e2.display();
    return 0;
}
Output:
101  Sonoo  890000
102  Nakul  59000
```

Here, when the constructor is called for object `e1`,then inside constructor, `this` indicates the pointer to `e1`.when the constructor is called for `e2`, then `this` refers to the pointer to `e2`.

2.18 Dynamic Memory Allocation

- In every c/c++ program, memory is reserved for a variable when it gets declared. By default, the memory required for a variable is decided during compile time. But in some situations, such decisions during compile time (i.e static memory allocation), will prevent the user from allocating more/less memory as per the requirement during run time. Such situations are handled by allocating memory dynamically during run time.
- Dynamic memory management has to be done by the programmer explicitly.
- ***Dynamic memory allocation:*** in this method, the memory for variables will be allocated during run time based on the requirements arising at runtime. And the memory blocks are taken from ***heap segment***. When such memory locations are no longer

required the same can be de-allocated and returned to OS. This is known as ***dynamic memory de-allocation.***

2.19 Dynamic Allocation Operators new and delete

- In c++, dynamic memory allocation is achieved with the help of `new` operator.
- Syntax:

```
pointer_variable = new data_type;
```

- Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.
- Example:.

```
int *p = NULL; // Pointer initialized with NULL
p = new int;     // Then request memory for the variable
          OR
int *p = new int; // Combine declaration of pointer and their assignment
```

Delete operator

- Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.
- Syntax:

```
delete pointer-variable;    // Release memory pointed by pointer-variable
```

- Here, pointer-variable is the pointer that points to the data object created by new.
  Examples:

```
delete p;
```

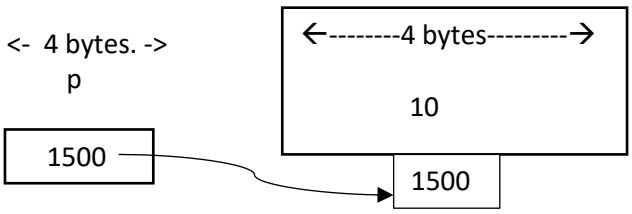
2.20 Initializing Allocated Memory

- We can also initialize the memory using new operator:

```
pointer-variable = new data-type(value);

Example:
int *p = new int(25);
float *q = new float(75.25);
```

Example: Dynamic memory allocation for integer

```
#include<iostream>
using namespace std;
int main(){
      int *p =new int;
      *p=10;
      cout<<"p="<<*p;

      delete p;
}

Output:
p=10;
```



Dynamic memory allocation with initialization

2.20 Allocating Arrays

- It is possible to allocate dynamic memory to arrays also.
- Syntax

```
data_type *ptr =new data_type[size];
```

- Example

```
int *p =new int[10];
```

```
int main()
{
      int *p, i,n;
      cout<<"Enter size of the array";
      cin>>n;
      p=new int[n];
      cout<<"\n enter array elements";
      for(i=0;i<n;i++)
            cin>>*(p+i);

      cout<<"\n elements are";
      for(i=0;i<n;i++)
            cout<<*(p+i);
}
Output:
Enter the size of the array :5
Enter array elements: 12 44 66 64 77
Elements are 12 44 66 64 77
```

2.21 Allocating objects

- Object can be assigned memory dynamically during run time using keyword new.

```
class Test{
      int a;
      public:
            Test(int x)
            {
                  A=x;
            }
            void disp()
            {
                  Cout<<a;
            }
};
int main()
{
      Test *p =new Test(10);
      p->disp();

      delete p;
}
```

Output:

10

In the above program, the class consist of one member variable, a constructor to initialize and a function to display the member variable. In main function the statement

```
Test *p = new Test(10);
```

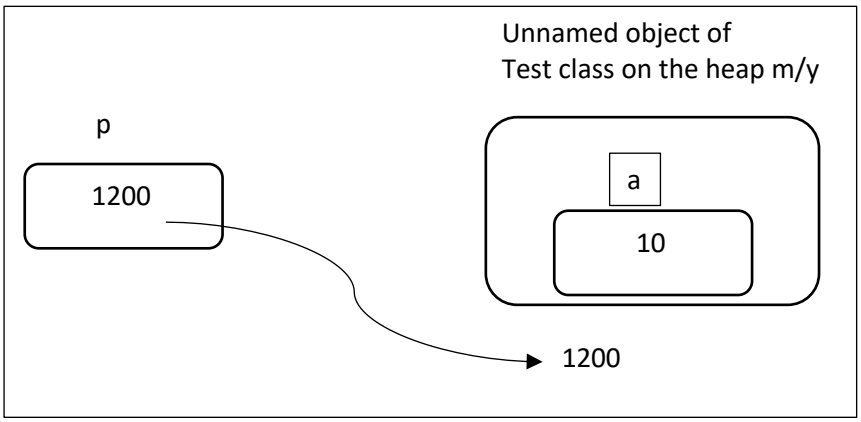
Can be rewritten as

```
Test *p;
p= new Test(10);
```

Does three operations viz.

- Creating a pointer of class type
- Allocating memory for an object from heap area
- Calling appropriate constructor

Memory map for dynamic object for the above program



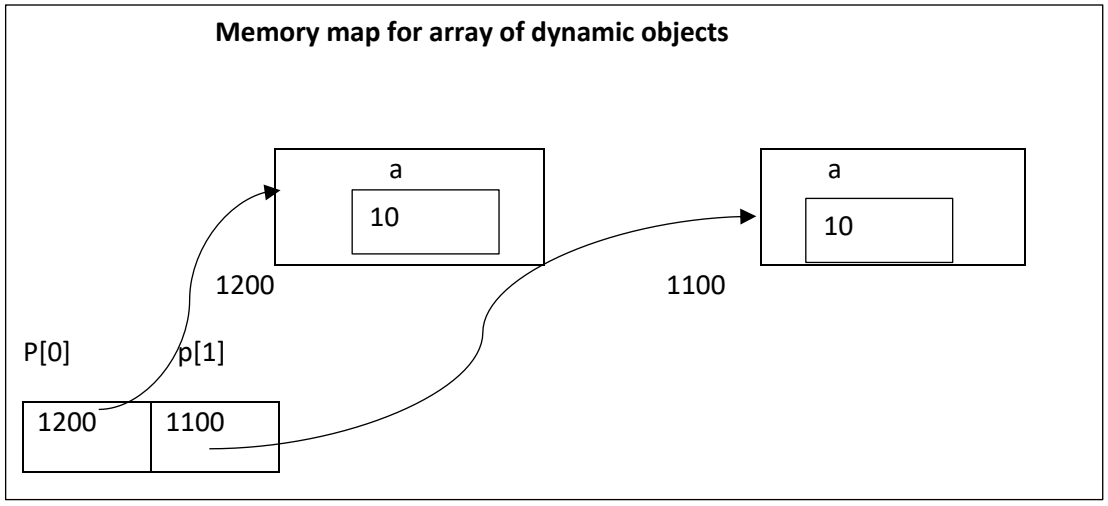
The memory allocated on the heap area(1200 in the above figure) will be released when the following statement is executed.

```
delete p;
```

An array of dynamic objects

- We can also create an array of dynamic objects using the following code.

```
int main()
{
      Test *p = {new Test(10), new Test(20)};

      for(int i=0;i<2;i++)
            p[i]->disp();

      delete [ ] p;
}
```



Program on constructor overloading

```
#include <iostream>
using namespace std;
class ABC
{
     private:
        int x,y;
     public:
        ABC ()       //constructor 1 with no arguments
       {
            x = y = 0;
        }
        ABC(int a)    //constructor 2 with one argument
       {
             x = y = a;
        }
        ABC(int a,int b)    //constructor 3 with two argument
        {
              x = a;
              y = b;
        }
        void display()
        {
              cout << "x = " << x << " and " << "y = " << y << endl;
        }
};

int main()
{
     ABC cc1; //constructor 1
     ABC cc2(10); //constructor 2
     ABC cc3(10,20); //constructor 3
     cc1.display();
     cc2.display();
     cc3.display();
     return 0;
 }  //end of program
```