CHAPTER 4

OBJECTS AND CLASSES.

(6HOURS)

Class:

- A Class in C++ is the building block, that leads to Object-Oriented programming.
- A C++ class is like a blueprint for an object.
- It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- 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 used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

Defining a class:

A class is defined in C++ using keyword **class** followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

Syntax

```
class  class_name{
        access specifier;  // can be public , private and protected.
       data members;    // variables to be used.
       member functions() {}    //methods to access data members.
};
```

Object:

- An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Declaring Objects:

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax

className objectname;

[To know]

Accessing data members and member functions:

- The data members and member functions of class can be accessed using the dot('.') operator with the object. Let say if name of object is **Ob** and we want to access member function with name **display()** then we will have to write **Ob.display();**

Accessing Data Members:

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

Member Functions in Classes:

- There are 2 ways to define a member function:
    - Inside class definition
    - Outside class definition
- To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

[* Note that all the member functions defined inside the class definition are by default **inline.**]

Program to understand basic about class and object:

```
#include <iostream>
 using namespace std;
  class Complex{        //class with name Complex.
   private:
    int x , y;                  // instance member variables.
   public:                      // Here, private, public are access specifier.
   void setdata (int a, int b){                // member function.
    x=a;
    y=b;
}
```

```
  void showdata(){       //member function.
  cout << "\n x"<< x<<"y"<<y;
}
};
 void main(){
  Complex c;   //Here c is an object of class Complex.
 c .  setdata (1,2);    //member access.
 c . showdata();
  return 0;
}
```

Constructor:

A constructor is a special type of member function that initializes an object automatically when it is created.

Compiler identifies a given member function is a constructor by its name and the return type.

Constructor has the same name as that of the class and it does not have any return type .Also the constructor is always public .A class constructor if defined is called whenever a program creates an object of that class.

It must be an instance member function , that is , it can never be static.

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself.
- Constructors don't have return type.
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

<u>There are 3 types of constructor:</u>

1. Default Constructor.
2. Parameterized Constructor.
3. Copy Constructor.

Syntax:

```
class  class_name{
private:
---------
public:
   class_name () { }    //default constructor.
   class_name (arguments){ } //parameterized constructor.
    class_name(class-name  &obj) { } //copy constructor.
};
```

Default constructor:

Default constructor is the constructor which doesn't take any argument. It has no parameters.

[**Note:** Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.]

[Note: Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case.]

Parameterized Constructor:

- It is possible to pass arguments to constructors.
- Typically, these arguments help initialize an object when it is created.
- To create a parameterized constructor, simply add parameters to it the way you would to any other function.
- When you define the constructor's body, use the parameters to initialize the object.

Uses of Parameterized constructor:

- It is used to initialize the various data elements of different objects with different values when they are created.
- It is used to overload constructors.

Copy Constructor:

Copy constructor is used to declare and initialize an object from another object.

```
class Complex{
 ------------;
   public:
    Complex (Complex &obj)   //copy constructor
  {
    -----------;
}
};
```

➢Why argument to a copy constructor must be passed as a reference?

Ans1: A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass an argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be passed by value.

Ans2: It is very essential to pass objects as reference. If an object is passed as value to the Copy Constructor then its copy constructor would call itself, to copy the actual parameter to the formal parameter. Thus an endless chain of call to the copy constructor will be initiated. This process would go on until the system run out of memory . Hence, in a copy constructor, the parameter should always be passed as reference.

Constructor Overloading:

- When more than one constructor function is defined in a class with different number of argument or same number of argument with different types ,then it is known as constructor overloading. Or, we can say, In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments . This concept is known as Constructor Overloading

Points to know:

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

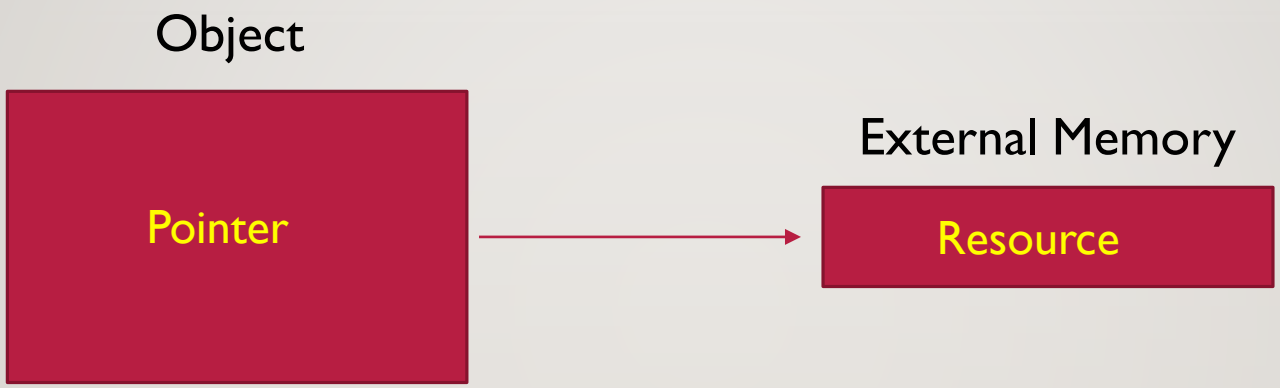
Destructor:

- Destructor is an instance member function of a class.
- The name of the destructor is same as the name of a class but preceded by tilde (~) symbol.
- Destructor can never be static.
- Destructor has no return type.
- Destructor takes no argument .Therefore, no overloading is possible.

Sample e.g.

```
#include <iostream>
 using namespace std ;
  class Complex{
   int id , price;
 public:
  ~Complex(){}
};
```

- It is invoked implicitly when object is going to destroy.
- Why destructor? It should be defined to release resources allocated to an object.

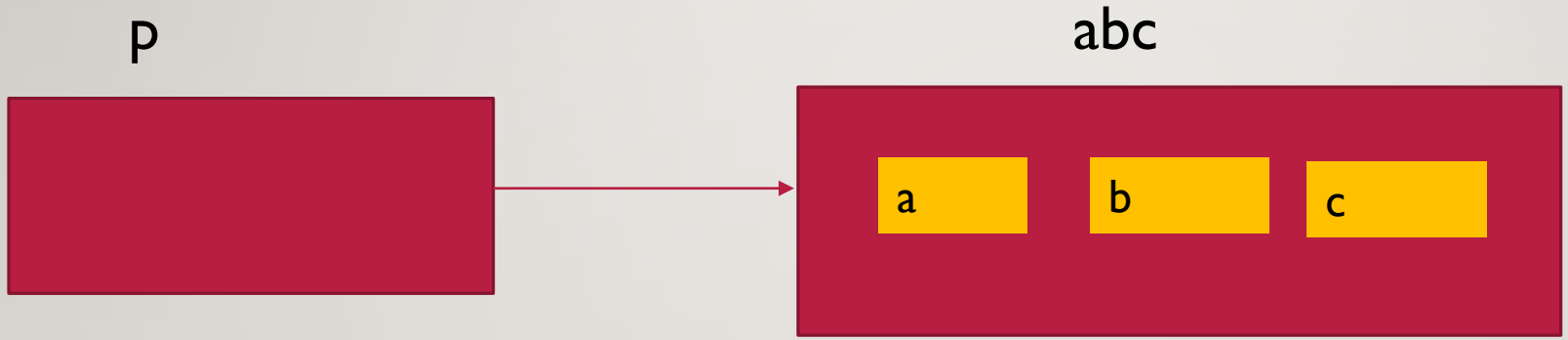


Constructor	Destructor
Constructor has the same name as class name.	Destructor also has the same name as class name but with **(~) tiled operator**.
Constructor is used to **initialize the instance of a class**.	Destructor **destroys the objects** when they are no longer needed.
Constructor can not be virtual.	Destructor can be virtual.
Constructor **allocates the memory**.	Destructor **releases the memory**.
Constructors can have arguments.	Destructor can not have any **arguments**.
Overloading of constructor is possible.	Overloading of Destructor is not possible.

ClassName()
{
//Body of Constructor
}
	~ ClassName()
	{
	}

Object Pointer(Pointer to Object):

A pointer contains address of an object is called object pointer. We can create pointer variable that will hold the address of the object, similar to other pointer to other standard variables.



Syntax: class_name *object pointer , object_name;

this Pointer:

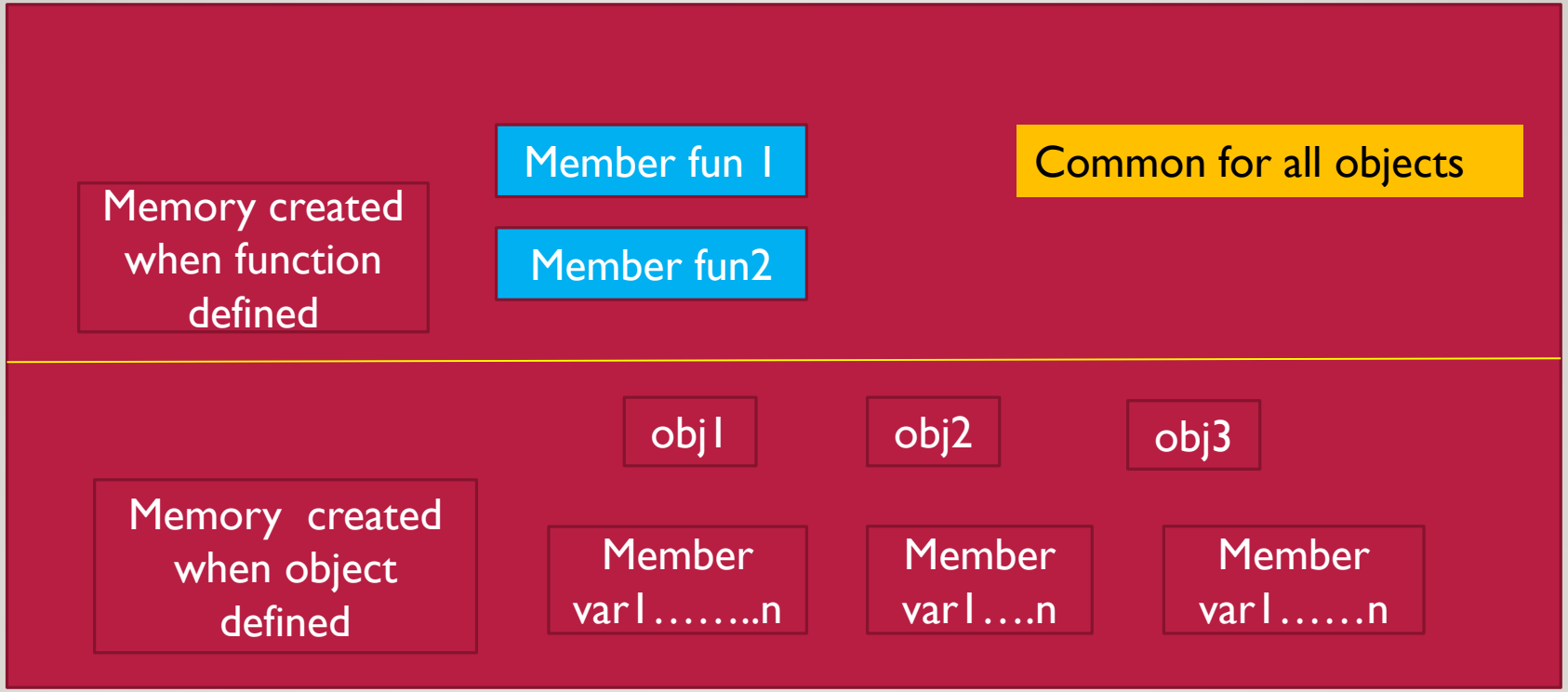
- this (keyword) is a local object pointer in every instance member function containing address of the caller object.
- this pointer can not be modify.
- It is used to refer caller object in member function.
- this pointer is a pointer accessible only within the **non-static** member functions of a class, struct , or union type. It points to the object for which the member function is called.
- Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Array Of Object:

- An object of class represents a single record in memory, if we want more than one record of class type, we have to create an array of class or object. As we know, an array is a collection of similar type, therefore an array can be a collection of class type.
- **Syntax:**

```
class class_name {
    member variables;
     member functions{ }
};
class_name  obj[size];
```

Relation of Object , Class and Memory:



Constant member functions and Constant Object:

- A function becomes **const** when **const** keyword is used in function's declaration.
- It is recommended practice to make as many functions **const** as possible so that accidental changes to objects are avoided.
- A const or a constant member function can only read or retrieve the data members of the calling object without modifying them.
- When a function is declared as const, it can be called on any type of object. Non-const functions can only be called by non-const objects.

Syntax of const member function:

```
return_type function_name (parameter_list) const
{
//body of the member function
}
```

Syntax of const object:

```
const  class_name  object;
```

Dynamic Memory allocation for object and object array:

This is the mechanism of allocating memory dynamically for object and object array .

Syntax:

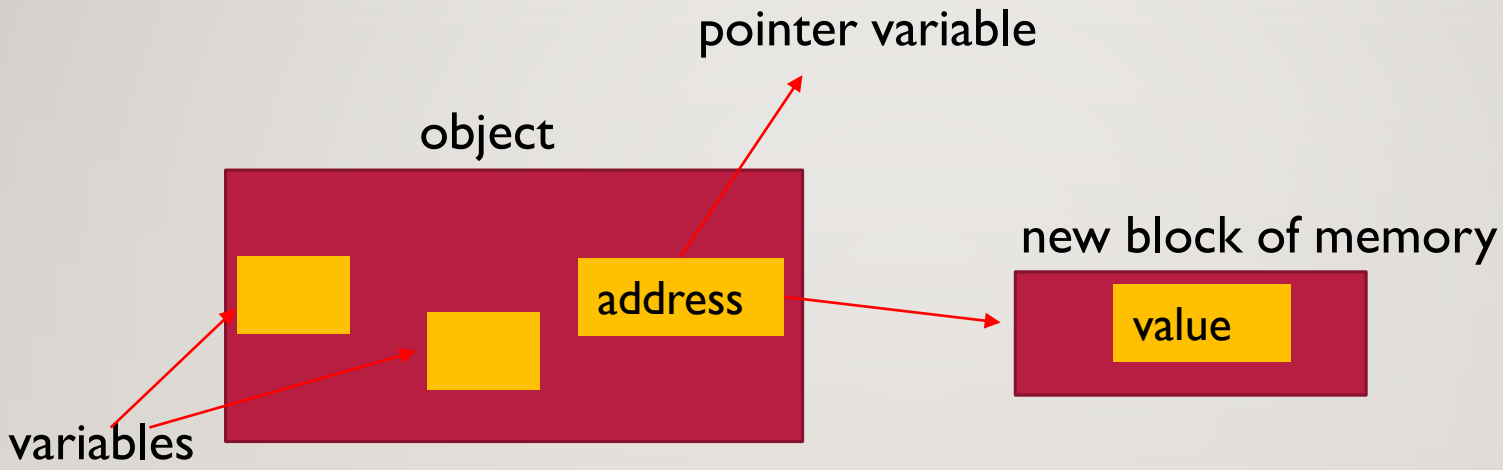
```
class_name  *object_pointer;

object_pointer = new class_name;  //allocating memory for single object.

object_pointer = new class_name [size ];  //allocating memory for an array of object.
```

Dynamic Constructor:

- Constructor can allocate dynamically created memory to the object.
- Thus object is going to use memory region, which is dynamically created by constructor.



[**The dynamic constructor does not create the memory of the object but it creates the memory block that is accessed by the object.]

const_cast Operator:

- const_cast is used to cast away the constness of variables.
- The const_cast operator is used to explicitly override const and/or volatile in a cast.
- const_cast can be used to change non-const class members inside a const member function.
- const_cast can be used to pass const data to a function that doesn't receive const.
- It is undefined behavior to modify a value which is initially declared as const.
- const_cast is considered safer than simple type casting.
- const_cast can also be used to cast away volatile attribute.
- **Syntax:** const_cast<type>(object) ; where type must be a pointer , reference , or pointer to member type and object must be either pointer or reference.

Mutable class member:

- The keyword **mutable** is mainly used to allow a particular data member of const object to be modified. Data members declared as **mutable** can be modified even though they are the part of object declared as const.
- The mutable storage class specifier is used only on a class data member to make it modifiable even though the member is part of an object declared as const.
- You cannot use the mutable specifier with names declared as static or const, or reference members.

1. PASSING OBJECTS AS AN ARGUMENTS IN A FUNCTION:

```
#include <iostream.h>
#include <conio.h>
  class Complex{
        private:
        int a , b;
        public:
         void set_data(int x, int y){
          a=x;
           b=y;
}
```

```
    void show_data () {
     cout <<“\n a”<<a<<“b”<<b;
}
 void add(Complex c1,Complex c2){ //member function
  a =c1.a+c2.a;
  b =c1.b+c2.b;
}
};
```

```
 void main(){
    Complex c1,c2,c3;
   c1.set_data(3,4);
    c2.set_data(5,6);
     c3.add(c1,c2);
     c3.show_data();
}
```

2. RETURNING OBJECT FROM FUNCTION:

```
#include <iostream.h>
#include <conio.h>
  class Complex{
        private:
        int a , b;
        public:
         void set_data(int x, int y){
          a=x;
           b=y;
}
```

```
 void show_data () {
     cout <<“\n a”<<a<<“b”<<b;
}
Complex add(Complex c){
Complex temp;
  temp . a =a + c . a;
   temp . b =b + c . b;
   return temp;
}};
```

```
 void main(){
    Complex c1,c2,c3;
   c1.set_data(3,4);
    c2.set_data(5,6);
   c3=c1.add(c2);   //c3=c1+c2;
     c3.show_data();
}
```

3. FRIEND FUNCTION AND FRIEND CLASS:

- **Friend Function:**

➢It is not a member function of a class to which it is friend.

➢It is declared in the class with friend keyword.

➢It must be defined outside the class to which it is friend.

➢It can access any member of the class to which it is friend.

➢A friend function of a class is defined outside that class scope but it has the right to access all private and protected members of the class.

➢It can not access members of the class directly.

➢It has no caller object.

➢It should not be defined with membership label.

➢Friend function can become friend to more than one class.(it is unique behavior of friend function).

E.g.

```
#include <iostream.h>
#include <conio.h>
 class Complex{
    private:
    int a , b;
     public:
      void setData (int x , int y){
        a=x;
        b=y;
       }
```

```
    void showData (){
   cout<<“a”<<a<<“b”<<b;
   }
   friend void fun (Complex);
};
 void fun (Complex c){
  cout<<“sum is:”<< c . a + c . b;
}
```

```
void main(){
   Complex c1;
    c1.setData(1,2);
    fun(c1);   //c1 object is passed as argument.
}
```

Syntax:

```
class className {
    ... .. ...
    friend return_Type function_Name(arguments);
    ... .. ...
}
```

Friend Class:

A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class.

Following are some important points about friend functions and classes:

1) Friends should be used only for limited purpose.
2) Friendship is not mutual. If class **A** is a friend of **B**, then **B** doesn't become a friend of **A** automatically.
3) Friendship is not inherited.
4) The concept of friends is not there in Java.

Friend Class:

```
class ClassB;
class ClassA {
        int x;
        friend class ClassB;   // friend class declaration
public:
         ClassA(){
         x=14;
         } };
class ClassB {
    int y;
    public:
         ClassB(){
         y=2;
         }
```

```
// member function to add x from ClassA and y from ClassB
    int sum( ClassA obj) {
        return obj.x + y;
    }
};

int main() {
    ClassB  objB;
    ClassA  objA;
    cout << "Sum: " << objB . sum (objA);
    return 0;
}
```

E.g. Program: [For your understanding]

```
#include <iostream>
using namespace std;
class B; //declaration of class B
class A{
int a;
public:
    friend void fun(A,B);
    };
```

```
class B{
    int b;
public:
    friend void fun(A,B);
    };
    void fun(A a1,B a2){
        cout<<"Enter a and b";
        cin >>a1.a>>a2.b;
    cout<<"sum is:"<<a1.a+a2.b;
    }
```

```
int main()
{
    A obj1;
    B obj2;
    fun(obj1,obj2);
    return 0;
}
```

4. STATIC MEMBER AND STATIC MEMBER FUNCTION:

- **Static local variables**:

➢They are by default initialized to zero.

➢Their lifetime is throughout the program.

E.g.

```
#include <iostream.h>
#include <conio.h>
 void func(){
 static int x;      //No garbage value.by default initialized to zero.
      int y;          // have garbage value
```

- **Static Member Variable:**

➢Declared inside the class body and also known as class member variable.

➢They must be defined outside the class.

➢They does not belong to any object but to the whole class.

➢There will be only one copy of static member variable for the whole class.

➢Any object can use the same copy of class variable.

➢They can also be used with class name.

- **Static Member Function:**

➢They are qualified with the keyword static.

➢They are also called class member function.

➢They can be invoked with or without object.

➢They can only access static member of the class.

```
#include <iostream>
using namespace std;
class Account{
private:
    int balance; // instance member variable
    static float roi; //class member variable
public:
    void setbalance (int b){
    balance=b;
    }
    static void setroi (float r){
    roi=r;
 cout<<"roi is:"<<roi;
}};
```

```
float Account::roi;
   int main(){
   Account a1,a2;
   a1.setroi(4.5f);  //if object exist.
   Account::setroi(5.5f);  //if there is no object.
   return 0;
   }
```