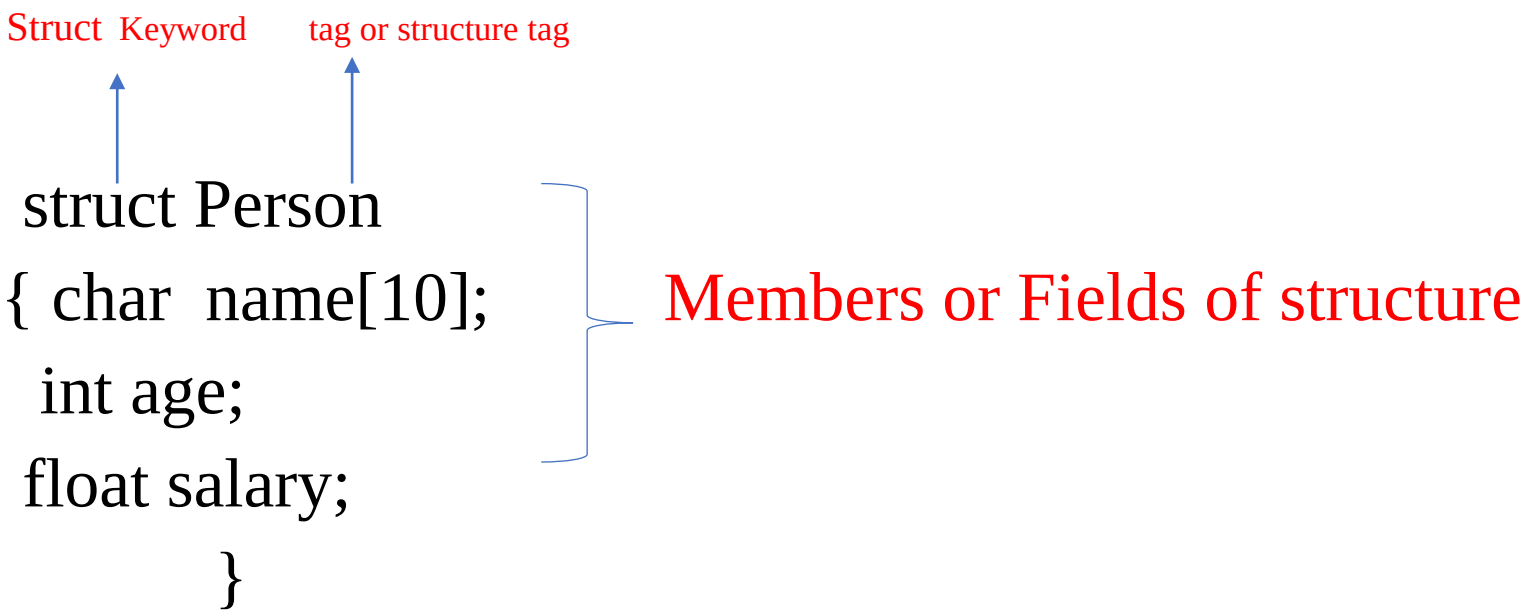
# Chapter 2
# Classes and Object

# C STRUCTURES REVISITED

- ***What is a structure?***
  A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single name.

Example:




***How to create a structure?***

'struct' keyword is used to create a structure.

***How to declare structure variables?***

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

Eg:- Person bill

```
// A variable declaration with structure declaration.
struct Point
{
int x, y;
} p1; // The variable p1 is declared with 'Point'
```

```
// A variable declaration like basic data types
struct Point
{ int x, y;
};
int main()
{
struct Point p1; // The variable p1 is declared like a
normal variable
}
```

***How to access structure elements?***

Structure members are accessed using dot (.) operator.

```
#include<iostream>

struct Point
{
   int x, y;
};

int main()
{
   struct Point p1 = {0, 1};

   // Accessing members of point p1
   p1.x = 20;
   cout<<p1.x<<“\n”<<p1.y;

   return 0;
}
```

# Limitations of C Structures

**The C structure does not allow the struct data type to be treated like built-in data types**

**We cannot use operators like +,- etc. on Structure variables. For example, consider the following code:**

```
struct number
{
        float x;
};
int main()
{
        struct number n1,n2,n3;
        n1.x=4;
        n2.x=3;
        n3=n1+n2;
        return 0;
}

/*Output:

prog.c: In function 'main':
prog.c:10:7: error:
invalid operands to binary + (have 'struct number' and 'struct number')
n3=n1+n2;

*/
```

# Limitation.....

- **No Data Hiding:** C Structures do not permit data hiding. Structure members can be accessed by any function, anywhere in the scope of the Structure.
- **Functions inside Structure:** C structures do not permit functions inside Structure
- **Static Members:** C Structures cannot have static members inside their body
- **Access Modifiers:** C Programming language do not support access modifiers. So they cannot be used in C Structures.
- **Construction creation in Structure:** Structures in C cannot have constructor inside Structures.

# Extension to Structure

- C++ define all the features of structure as defined in C.
- C++ add the concept of OOP.
- In C++, a structure can have both the functions and variables as members.

All these extensions are incorporated in another user defined type known as **CLASS**.

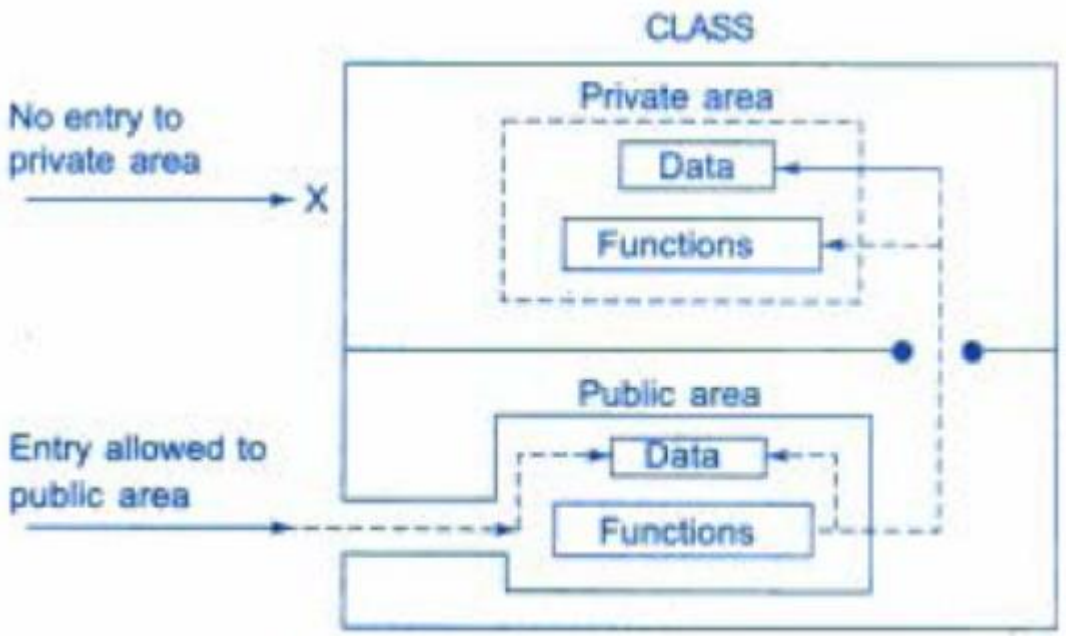
* By default, the member of class are private, while, by default , the members of a structures are public.

# Class & Object

- A class is a way to bind the data and its associated functions together. It allows the data and functions to be hidden, if necessary, from external use. A class declaration is similar syntactically to a structure.

Declaration of Class:

```
Class class_name
 {
  private:
      variable declaration;
       function declaration;
  public:
      Variable declaration;
       function declaration;
  }
```



# Points to remember!

- The variables declared inside the class definition are known as **data members** and the functions declared inside a class are known as **member functions**.
- Wrapping of data and function and function into a single unit (i.e. class) is known as **data encapsulation**.
- By default the data members and member function of a class are **private**.
- Private data members can be accessed by the functions that are wrapped inside the class.

## General steps to write a C++ program using class and object:

- Header files
- Class definition
- Member function definition
- void main function

# Example:

## # WAP to find the SUM of two integers using class and object:

```
 #include<iostream>
using namespace std
 class Add
{
int x, y, z;
public:
void getdata()
{
cout<<"Enter two numbers";
cin>>x>>y;
}
void calculate(void);
void display(void);
};
void Add :: calculate()
{
z=x+y;
}
void Add :: display()
{
cout<<z;
}
void main()
{
Add a;
a.getdata();
a.calculate();
a.display();
}
```

**Output:**

Enter two numbers 5 6

11

A member function can be defined:

(i) **Inside** the class definition

(ii) **Outside** side the class definition using scope resolution operator (**::**)

# Memory allocation for Objects

- Memory space for objects is allocated when they are declared not when class is specified.
- Member functions are created and placed in the memory space only once when they are defined.
- All objects belong to the class use the same member function
- Space for member variables is allocated separately for each object.

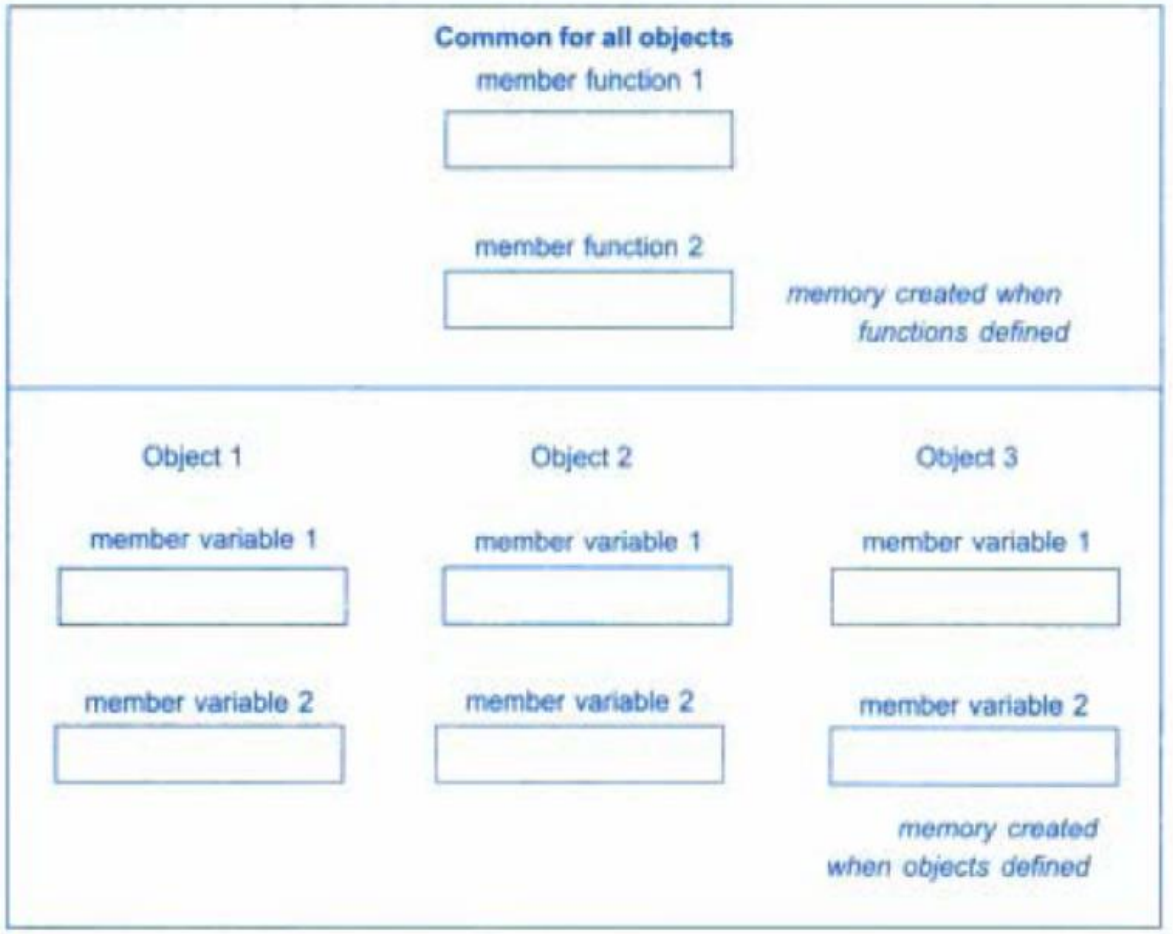



Fig. 5.3 ⇔ Object of memory

# Static Data member

- The data member of a class preceded by the keyword static is known as static member.
- When we precede a member variable's declaration with **static**, we are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Hence static variables are called class variables.
- Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable.
- All **static** variables are initialized to zero before the first object is created.
- Normal data members are called object variable but static data members are called class variables.

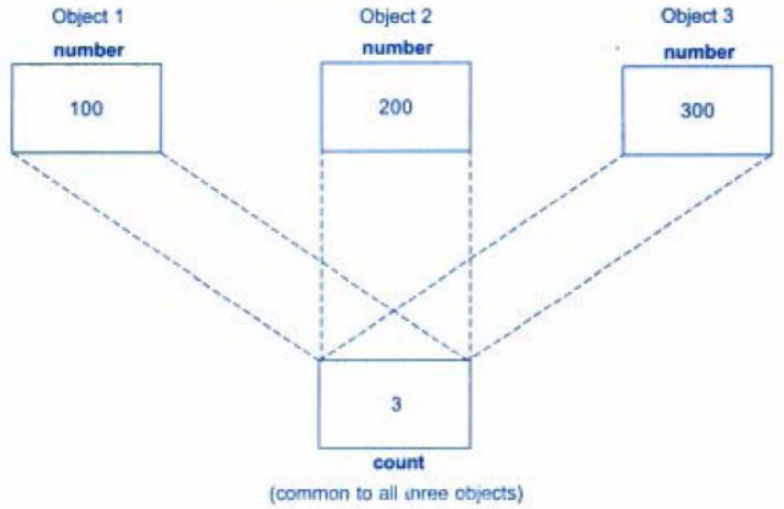



Fig. 5.4 ⇔ *Sharing of a static data member*

# Static Member function

- A static function can have access to only other static members (functions or variables) declared in the same class. (Of course, global functions and data may be accessed by static member functions.)
- It is accessed by class name and not by object's name i.e. class-name::function-name;
- The function name is preceded by the keyword **static**.
- A static member function does not have this pointer.
- There cannot be a static and a non-static version of the same function.
- A static member function may not be virtual.
- Finally, they cannot be declared as const or volatile.

| | setcode | showcode | showcount |
|---|---|---|---|
| t1 | Init count =0;<br>New Count=1<br>Code =1 | 1 | |
| t2 | Previous Count =1;<br>New count =2<br>Code = 2 | 2 | 2 |
| t3 | Prev count =2;<br>New count =3;<br>Code =3 | 3 | 3 |

# Array of Object

- Collection of similar types of object is known as array of objects

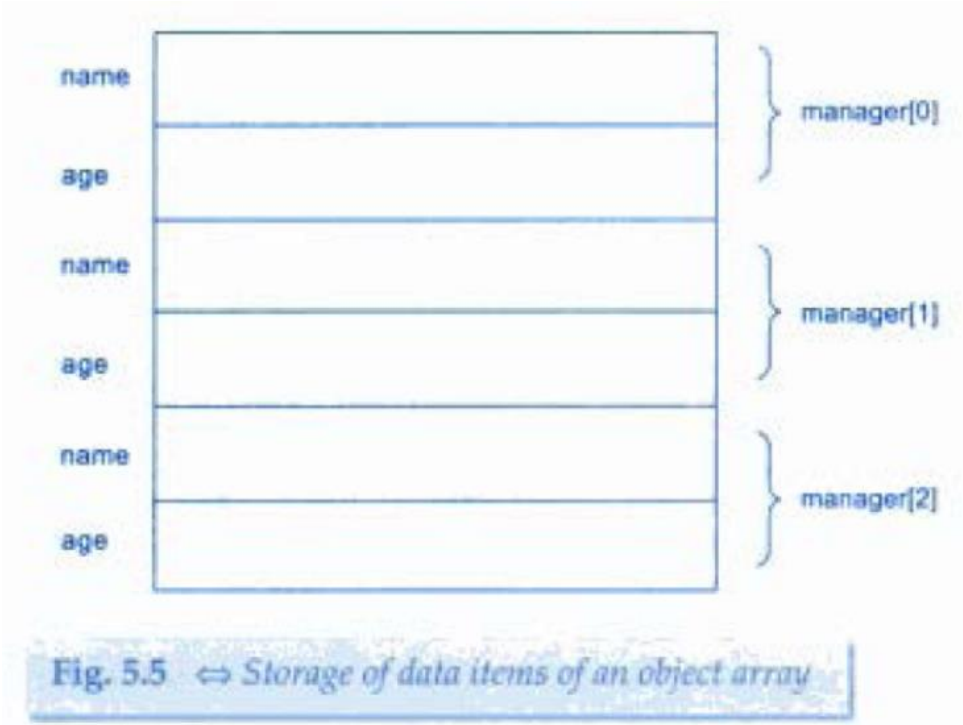



Fig. 5.5 ⇔ *Storage of data items of an object array*

# Object as function argument

- Object may be used as a function argument. This can be done in two ways

1. A copy of entire object is passed to the function (pass by value)
2. Only the address of the object is transferred to the function(pass by reference)

Pass by value:- Any change made to the object inside the function do not affect the object used to call the function.

Pass by reference:- Here address of the object is passed, this means any change made on object inside the function will reflect in the actual object.

# Friend Function

- Scope of a friend function is not inside the class in which it is declared.
- Since its scope is not inside the class, it cannot be called using the object of that class
- It can be called like a normal function without using any object.
- It cannot directly access the data members like other member function and it can access the data members by using object through dot operator.
- It can be declared either in private or public part of the class definition.
- Usually it has the objects as arguments.

Definition of Friend Function:

```
Class ABC
{
…..
….
Public:
…..
…..
Friend void xyz(void);
};
```

# Friend Class

It is possible for one class to be a **friend** of another class. When this is the case, the **friend** class and all of its member functions have access to the private members defined within the other class.

```
Class ABC{
....
....
Friend class min;
};
Class min{....
                .....};
```

Member function of one class can be the friend function of another class

(Friend int classname1 :: funct();)  //declaration in another class2.

# Function Overloading

- It is the process by which a single function can perform different task, depending upon no of parameters and types of parameters.
- Concept of function overloading is to design a family of function with one function name but with different argument lists.

How it works?

1. Function call first matches the prototype having the same number and types of the arguments and then call the appropriate function for execution .

Example:

prototype

1. int add(int a, int b);

2.int add(int a, int b, int c);

3.double add(double x, double y);

4.double add(int p, double q);

5. double add(double p, int q);

```
//function call
add(5,10); // prototype 1
add(15, 10.0); // prototype 4
add(1, 2, 3);   // prototype 2
add(11.2, 7.2); // prototype 3
add(5.4, 3); // prototype 5
```

# Constructor

- A constructor is a special member function whose task is to initialize the object of a class.
- Its name is same as the class name.
- A constructor does not have a return type.
- A constructor is called or invoked when the object of its associated class is created.
- It is called constructor because it constructs the values of data members of the class.
- A constructor cannot be virtual (shall be discussed later on).
- A constructor can be overloaded

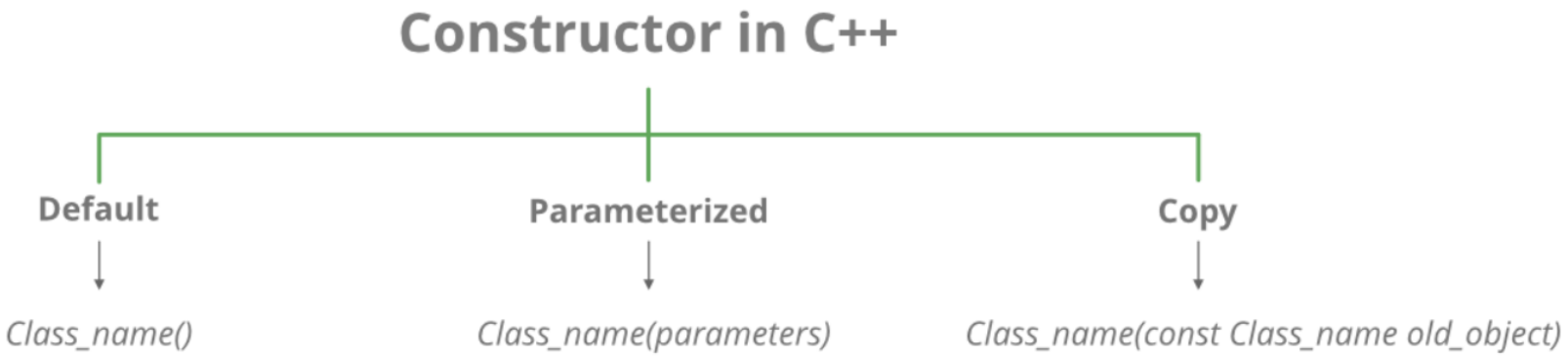
There three types of constructor:

(i) Default Constructor

(ii) Parameterized Constructor

(iii) Copy constructor

- **Default Constructor :** The constructor which has no arguments is known as default constructor.
- **Parameterized constructor**: The constructor which takes some argument is known as parameterized constructor.
- **Copy Constructor:**The constructor which takes reference to its own class as argument is known as copy constructor.



# Pointer in c++

- Pointer is a variable in C++ that holds the address of another variable. They have **data type** just like variables, for example an integer type pointer can hold the address of an integer variable and an character type pointer can hold the address of char variable

Syntax : data-type *pointer_name;

Declaration int *p;  //can  hold the address of int type variable

```
int var;
p = &var;
cout<<&var; // display the address of var
cout<<p; //display the address of var;
cout<<&p; //display the address of p;
cout<<*p; //display the value of var.
```

let: var = 101, memory address = 0x62ccf

p  memory address = 0x62cff

# Dynamic Constructor

- When allocation of memory is done dynamically using dynamic memory allocator new in a constructor, it is known as **dynamic constructor**. By using this, we can dynamically initialize the objects.

Dynamic Memory Allocator:- (new and delete operators)

Memory in your C++ program is divided into two parts −

**The stack** − All variables declared inside the function will take up memory from the stack.

**The heap** − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

new data-type;  (data-type can be any built in /user define data type or array)

Ex:

Int *pvalue = NULL;  //pointer initialize to null

pvalue = new int; // request memory for the variable

you can free up the memory that it occupies in the free store with the ‘delete’ operator as follows :

delete pvalue; //release memory pointed by the pvalue.

```
#include <iostream>
using namespace std;
int main ()
{   double  *pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
 *pvalue = 1000;     // Store value at allocated address
  cout << "Value of pvalue : " << *pvalue << endl;
 delete pvalue;         // free up the memory.
return 0;
}
```

o/p: 1000

# Dynamic Memory Allocation for Array

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char *pvalue = NULL;  //pointer initialized with null
 pvalue = new char [20]; //request memory for the variable
```

To remove the array that we have just created the statement would look like this −

```
 delete [ ] pvalue; //delete array pointed by the pvalue.
```

Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows −

```
double **pvalue = NULL;
 pvalue = new double[3][4]; // allocate memory for a 3x4 array
 delete [ ] pvalue; // delete array pointed to pvalue.
```

# Dynamic memory Allocation for Objects

- Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept −

```
Class Add {
        int a, b;
        public:
          void display();
        };
Int main()
{ Add *obj = new Add[10];
Delete [] obj; //delete array
```

# Pointer to member

- It is used to assign the address of member of a class and assign to the pointer.

Definition:

```
Class A
{ int m;
Public:
Void show();
};
Int A ::* ip = &A :: m;   //definition of pointer to member.
```

Here, ip pointer act like class member and it must be invoked by class object.

A::* (“pointer to member of A class”)

&A ::m (“address of the m member of A class”)

Invoke:

```
A a;
Cout<<a.*ip;
Cout<<a.m;
```

Both will display the same output.

```
A *ap = &a;  //pointer to object a
cout << ap -> *ip;
Cout<< ap -> m;
```

display m

- Dereferencing operator ->* used to access a member when we use pointers to both the object and the member.
- Dereferencing operator * is used when object itself is used with the member pointer

Pointer to member function

Object-name.*pointer to member function

Pointer-to-object ->* pointer- to-member-function

# Destructor

- It is a special member function which is executed automatically when an object is destroyed.
- Its name is same as class name but it should be preceded by the symbol ~.
- It cannot be overloaded as it takes no argument.
- It is used to delete the memory space occupied by an object.
- It has no return type.
- It should be declared in the public section of the class.

Tilde ~classname() { … }

A destructor is **automatically called** when:
1) The program finished execution.
2) When a scope (the { } parenthesis) containing **local variable** ends.

* Object are destroyed in reverse order.