Object Oriented Programming with C++

Unit-4 Object and Classes

Outline

- Basics of Object and Class in C++
- Private and Public Members
- Static data and Function Members
- Constructors and their types
- Destructors
- Operator Overloading
- Type Conversion

Object and Class in C++

What is an Object?

Pen

Board

Laptop

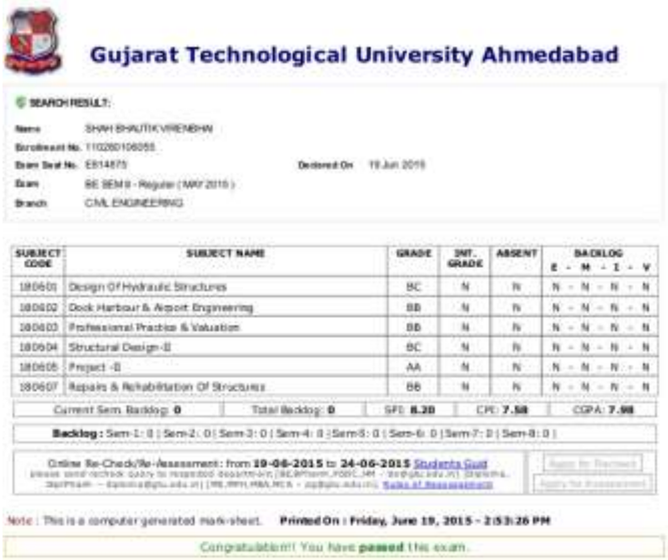
Bench

Projector

Bike

Physical objects…

What is an Object? (Cont…)



Gujarat Technological University Ahmedabad

SEARCH RESULT:

Name	SHAH BHAUTIK VIRENBHAI		
Enrollment No.	110280106055		
Exam Seat No.	E814875	Declared On	19 Jun 2015
Exam	BE SEM 8 - Regular (MAY 2015)		
Branch	CIVIL ENGINEERING		

SUBJECT CODE	SUBJECT NAME	GRADE	INT. GRADE	ABSENT	BACKLOG E - M - I - V
180601	Design Of Hydraulic Structures	BC	N	N	N - N - N - N
180602	Dock Harbour & Airport Engineering	BB	N	N	N - N - N - N
180603	Professional Practice & Valuation	BB	N	N	N - N - N - N
180604	Structural Design-II	BC	N	N	N - N - N - N
180605	Project -II	AA	N	N	N - N - N - N
180607	Repairs & Rehabilitation Of Structures	BB	N	N	N - N - N - N

Current Sem. Backlog: **0**	Total Backlog: **0**	SPI: **8.20**	CPI: **7.58**	CGPA: **7.98**

Backlog : Sem-1: 0 | Sem-2: 0 | Sem-3: 0 | Sem-4: 0 | Sem-5: 0 | Sem-6: 0 | Sem-7: 0 | Sem-8: 0 |

Online Re-Check/Re-Assessment : from **19-06-2015** to **24-06-2015** Students Guid
please send recheck query to respected department [BE,BPharm,PDDC,MM – be@gtu.edu.in] Diploma, DipPharm – diploma@gtu.edu.in | [ME,MPH,MBA,MCA – pg@gtu.edu.in] Rules of Reassessment

Note : This is a computer generated mark-sheet. **Printed On : Friday, June 19, 2015 - 2:53:26 PM**

Congratulation!! You have **passed** this exam.

Result

Bank Account

Logical objects…

Attributes and Methods of an Object

Bank Account

Object: Person

Attributes

Name

Age

Weight

Methods

Eat

Sleep

Walk

Object: Car

Attributes

Company

Color

Fuel type

Methods

Start

Drive

Stop

Object: Account

Attributes

AccountNo

HolderName

AccountType

Methods

Deposit

Withdraw

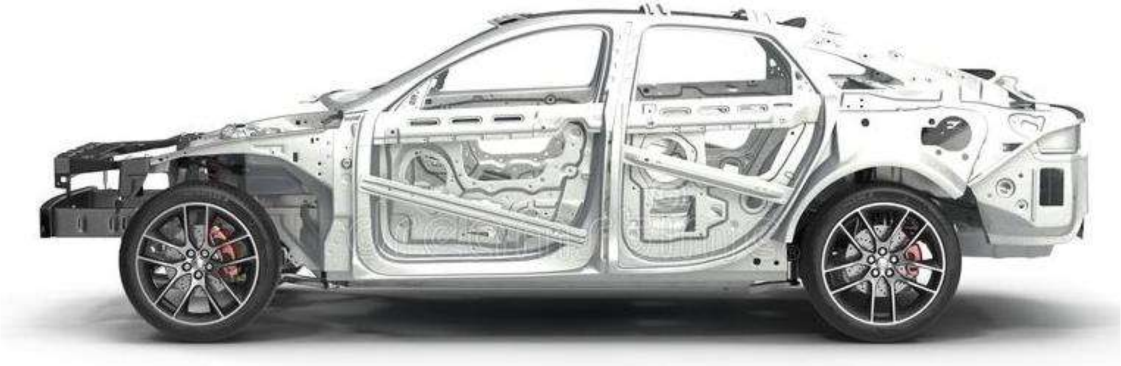
Transfer

Class

A Class is a blueprint of an object

A Class describes the object

Class car



Class: Car

Class: Car

Properties (Describe)
Company
Model
Color
Mfg. Year
Price
Fuel Type
Mileage
Gear Type
Power Steering
Anti-Lock braking system

Methods (Functions)
Start
Drive
Park
On_break
On_lock
On_turn

Objects of Class Car

Honda City

Hyundai i20

Sumo Grand

Mercedes E class

Swift Dzire

Class in C++

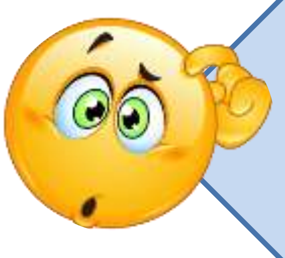
- A **class** is a **blueprint** or **template** that describes the object.
- A class specifies the attributes and methods of objects.

Example:

```
class car
{
    // data members and member functions
}car1;
```

- In above example class name is **car**, and **car1** is object of that class.

Specifying Class



How to declare / write class ?

How to create an object (instance/variable of class)?

How to access class members ?

How to declare / write class ?

Class

```
class car
{
  private:
    int price;
    float mileage;
  public:
    void start();
    void drive();
 };
```

Car

Attributes

Price

Mileage

Methods

Start

Drive

How to create an object ?

Syntax:

```
className   objectVariableName;
```

Class

```
class car
{
  private:
    int price;
    float mileage;
  public:
    void start();
    void drive();
 };
```

Object

```
int main()
{
  car c1;
  c1.start();
}
```

Object in C++

- An **object** is an instance of a class
- An **object** is a variable of type class

Class

```
class car
{
  private:
    int price;
    float mileage;
  public:
    void start();
    void drive();
 };
```

Object

```
int main()
{
  car c1;
  c1.start();
  c1.drive();
}
```

Program: class, object

- Write a C++ program to create class Test having data members mark and spi.
- Create member functions **`SetData()`** and **`DisplayData()`** to demonstrate class and objects.

Program: class, object

```
#include <iostream>
using namespace std;
class Test
{
   private:
     int mark;
     float spi;
   public:
     void SetData()
     {
       mark = 270;
       spi = 6.5;
     }
     void DisplayData()
     {
       cout << "Mark= "<<mark<<endl;
       cout << "spi= "<<spi;
     }
} ;
```

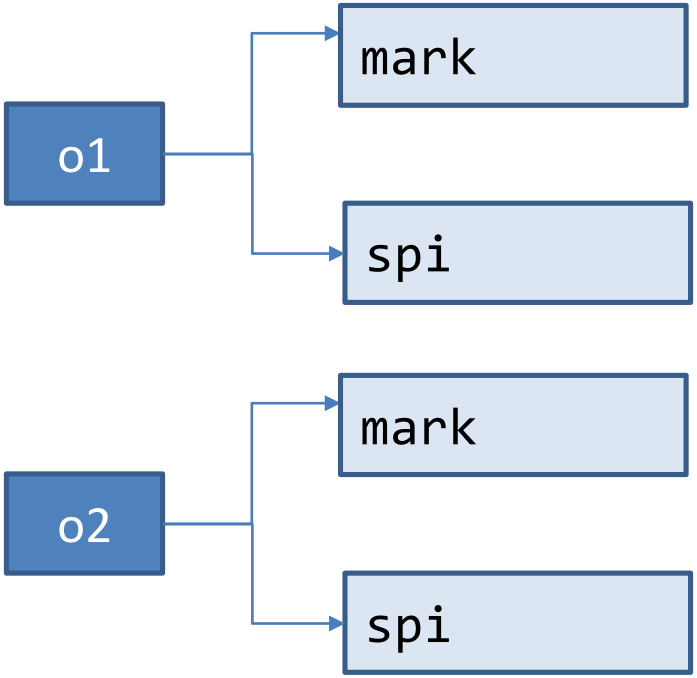
```
int main()
{
   Test o1;
   o1.SetData();
   o1.DisplayData();
   return 0;
}
```

- Control jumps to definition of `DisplayData()`

```
class Test
{
   private:
     int mark;
     float spi;
   public:
    void SetData()
    {
     cin>>mark;
     cin>>spi;
    }
    void DisplayData()
    {
     cout << "Mark= "<<mark;
     cout << "spi= "<<spi;
    }
} ;
```

```
int main()
{
   Test o1,o2;


   return 0;
}
```



Program: class, object

- Write a C++ program to create class Car having data members Company and Top_Speed.
- Create member functions **`SetData()`** and **`DisplayData()`** and create two objects of class Car.

Program: class, object

```
class Car
{
  private:
    char company[20];
    int top_speed;
  public:
    void SetData(){
       cout<<"Enter Company:";
       cin>>company;
       cout<<"Enter top speed:";
       cin>>top_speed;
    }
    void DisplayData()
    {
       cout << "\nCompany:"<<company;
       cout << "\tTop Speed:"<<top_speed;
    }
} ;
```

```
int main()
{
   Car o1;
   o1.SetData();
   o1.DisplayData();
   return 0;
}
```

Program: class, object

- Write a C++ program to create class Employee having data members Emp_Name, Salary, Age.
- Create member functions `SetData()` and `DisplayData()`.
- Create two objects of class Employee

Program: class, object

```
class Employee
{
   private:
     char name[10];
     int salary, age;

   public:
     void SetData()
     {
        cin>>name>>salary>>age;
     }
     void DisplayData()
     {
        cout << “Name= "<<name<<endl;
        cout << “salary= "<<salary<<endl;
        cout << “age= "<<age;
     }
} ;
```

```
int main()
{
   Employee o1;
   o1.SetData();
   o1.DisplayData();
   return 0;
}
```

Private and Public Members

Private

Public

Private Members

Class

```
class car
{ Private:
   long int price;
   float mileage;
   void setdata()
   {
      price = 700000;
      mileage = 18.5;
   }
};
```

- **Private** members of the class can be accessed **within the class** and from **member functions** of the class.

By default all the members of class are **private**

- A **private** member variable or function **cannot** be accessed, or even viewed from outside the class.

Private Members

- **Private** members of the class can be accessed within the class and from member functions of the class.
- They cannot be accessed outside the class or from other programs, not even from inherited class.
- If you try to access private data from outside of the class, compiler throws error.
- This feature in OOP is known as **Data hiding / Encapsulation**.
- If any other access modifier is not specified then member default acts as Private member.

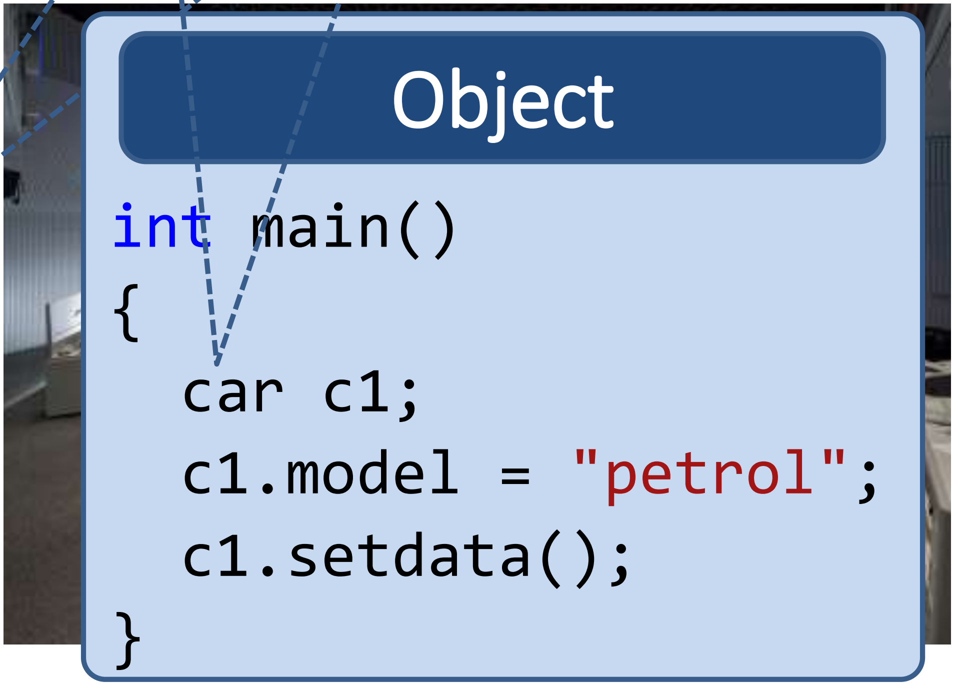
Public Members

Class

```
class car
{
  private:
      long int price;
      float mileage;
  public:
      char model[10];
      void setdata()
      {
        price = 700000;
        mileage=18.53;
      }
};
```

The **public** members of a class can be accessed outside the class using the **object name** and dot operator '.'

Public members are accessible from anywhere **outside the class** but within a program.



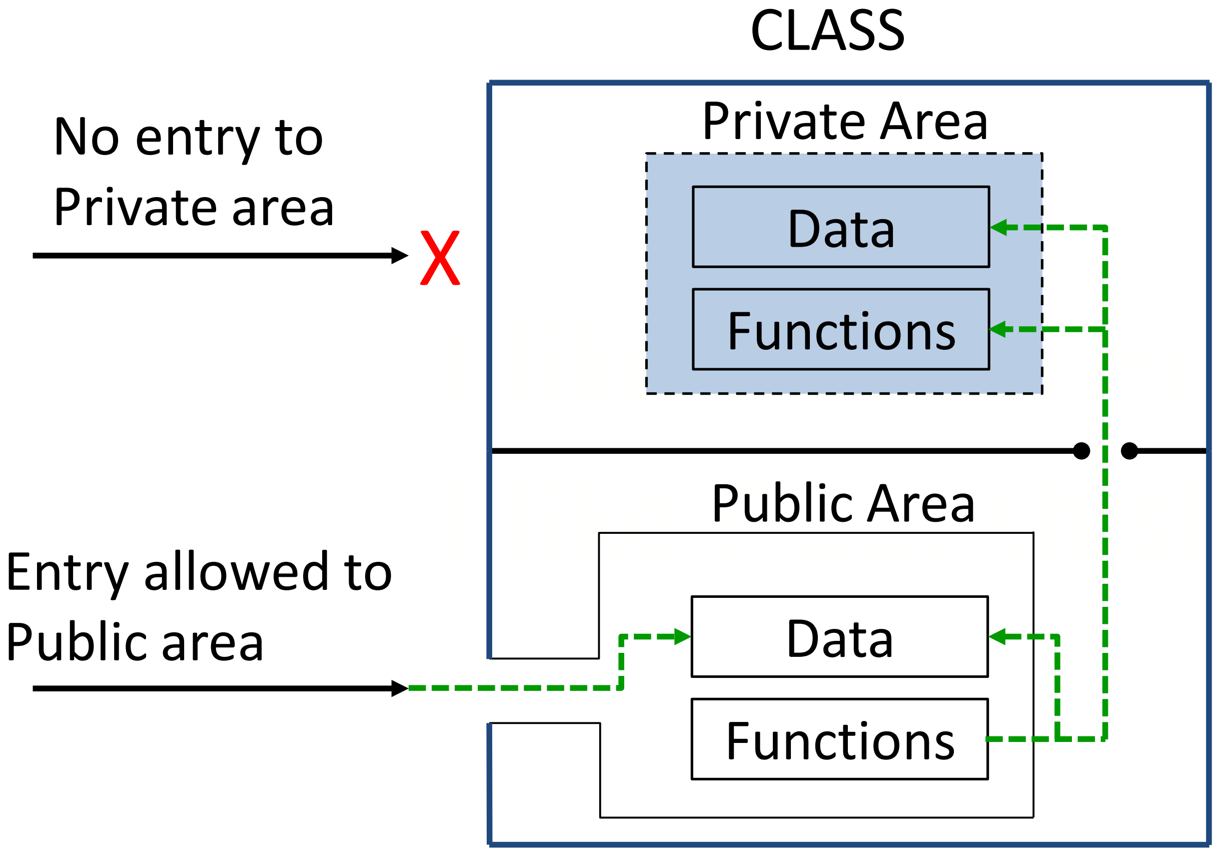
Object

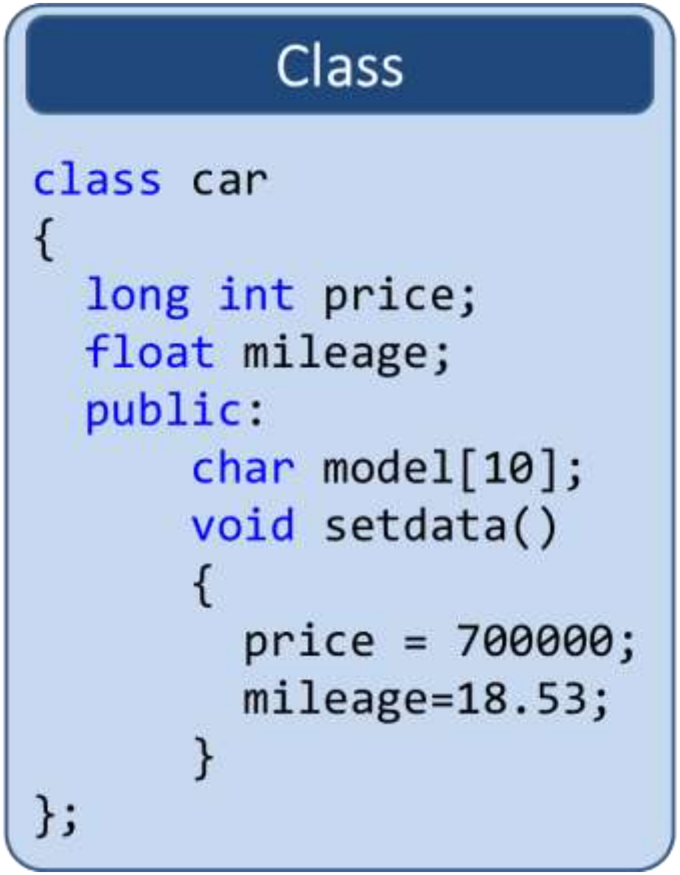
```
int main()
{
   car c1;
   c1.model = "petrol";
   c1.setdata();
}
```

Public Members

- The **public** keyword makes data and functions public.
- Public members of the class are accessible by any program from anywhere.
- Class members that allow manipulating or accessing the class data are made public.

Data Hiding in Classes





```
class car
{
  long int price;
  float mileage;
  public:
      char model[10];
      void setdata()
      {
        price = 700000;
        mileage=18.53;
      }
};
```

Example Class in C++

```
class Test
{

          int data1;
          float data2;
   public:
          void function1()
          {
            data1 = 2;
          }
          float function2()
          {
            data2 = 3.5;
            return data2;
          }
};
```

By Default the members of a class are private.

private is a Keyword

Private data and functions can be written here

public is a Keyword

Public data and functions can be written here

Function Definition Outside Class

Function definition outside the class

Syntax:

```
Return-type class-name :: function-name(arguments)
{
    Function body;
}
```

- The membership label **class-name::** tells the compiler that the function belongs to class

Example:

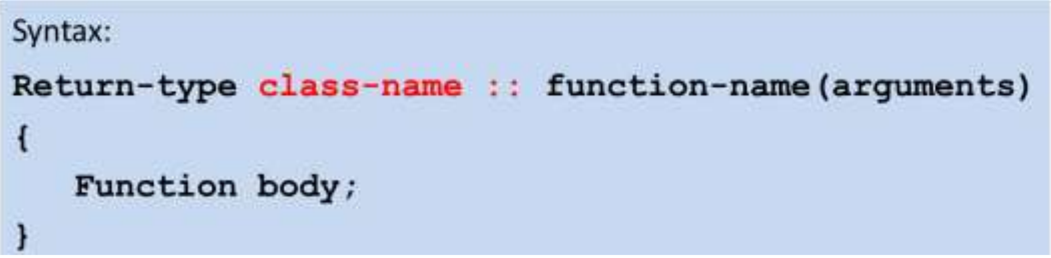
```
void            SetData(int i,int j)
{
    mark = i;
    spi = j;
}
```

Function Definition outside class

```
class car
{
  private:
    float mileage;
  public:
   float updatemileage();
    void setdata();



};
```



Syntax:

```
Return-type class-name :: function-name(arguments)
{
    Function body;
}
```

```
void car :: setdata()
{
   mileage = 18.5;
}
```

```
float car :: updatemileage()
{
  return mileage+2;
}
```

```
int main()
{
  car c1;
  c1.setdata();
  c1.updatemileage();
}
```

Program: function outside class

```
class Test
{
  private:
   int mark;
   float spi;
  public:
   void SetData(int,float);
   void DisplayData();
};
void Test :: SetData(int i,float j){
   mark = i;
   spi = j;
}
void Test :: DisplayData()
{
   cout << "Mark= "<<mark;
   cout << "\nspi= "<<spi;
}
```

```
int main()
{
  Test o1;
  o1.SetData(70,6.5);
  o1.DisplayData();
  return 0;
}
```

- The membership label **Test::** tells the compiler that the **SetData()** and **DisplayData()** belongs to **Test** class

Member Functions with Arguments

Program: Function with argument

- Define class **`Time`** with members **`hour, minute`** and **`second`**. Also define function to **`setTime()`** to initialize the members, **`print()`** to display time. Demonstrate class **`Time`** for two objects.

Program: Function with argument

```
#include<iostream>
using namespace std;
class Time
{
   private :
     int hour, minute, second;
   public :
     void setTime(int h, int m, int s);
     void print();
};
```

Program: Function with argument

```
void Time::setTime(int h, int m, int s)
{
   hour=h;
   minute=m;
   second=s;
}
void Time::print()
{
   cout<<"hours=\n"<<hour;
   cout<<"minutes=\n"<<minute;
   cout<<"seconds=\n"<<second;
}
```

Program: Function with argument

```
int main()
{
  int h,m,s;
  Time t1;

  cout<<"Enter hours="; cin>>h;
  cout<<"Enter minutes="; cin>>m;
  cout<<"Enter seconds="; cin>>s;

  t1.setTime(h,m,s);
  t1.print();
  return 0;
}
```

Program: Function with argument

- Define class **Rectangle** with members **width** and **height**. Also define function to **set_values()** to initialize the members, **area()** to calculate area. Demonstrate class **Rectangle** for two objects.

Program: Function with argument

```
class Rectangle
{
   int width, height;
   public:
     void set_values (int,int);
     int area(){
        return width*height;
     }
};
void Rectangle::set_values (int x, int y){
   width = x;  height = y;
}
int main(){
   Rectangle rect;
   rect.set_values(3,4);
   cout << "area: " << rect.area();
   return 0;
}
```

Program: Function with argument

- Define class **Employee** with members **age** and **salary**.

1. Also define function to **setdata()** to initialize the members.
2. Define function **displaydata()** to display data.
3. Demonstrate class **Employee** for two objects.

```
int main(){
   Employee yash,raj;
   yash.setData(23,1500);
   yash.displaydata();

   raj.setData(27,1800);
   raj. displaydata();
   return 0;
}
```

Program: Function with argument

```
class Employee{
   private :
     int age; int salary;
   public :
     void setData(int , int);
     void displaydata();
};
void Employee::setData(int x, int y){
   age=x;
   salary=y;
}
void Employee::displaydata(){
   cout<<"age="<<age<<endl;
   cout<<"salary="<<salary<<endl;
}
```

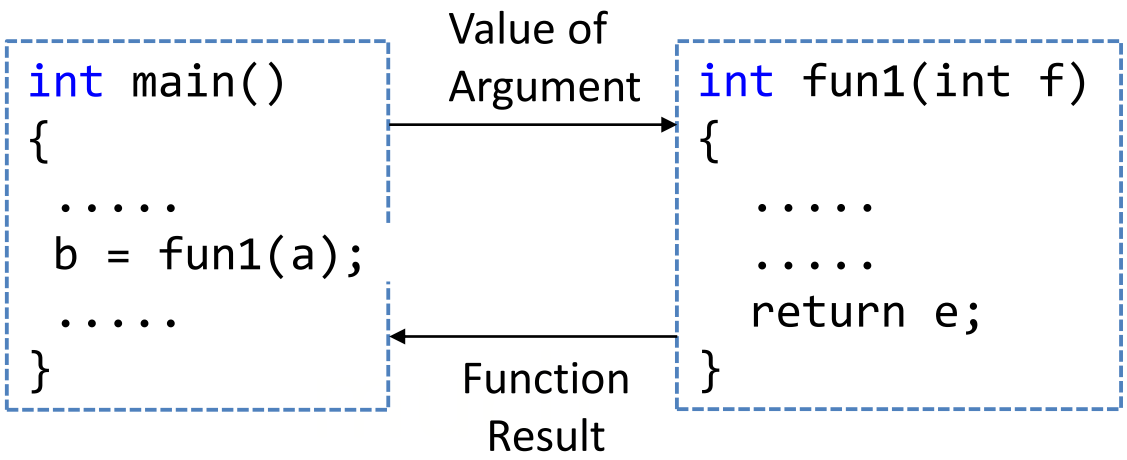
Passing Objects as Function Arguments

Function with argument and returns value

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

int add(int, int);

int main(){
   int a=5,b=6,ans;
   ans = add(a,b);
   cout<<"Addition is="<<ans;
   return 0;
}
int add(int x,int y)
{
   return x+y;
}
```



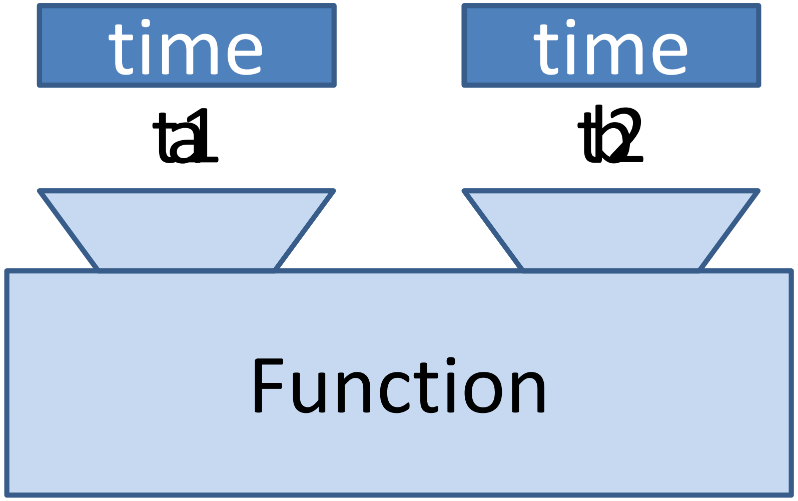
```
int main()
{
 .....
 b = fun1(a);
 .....
}
```

Value of Argument

```
int fun1(int f)
{
   .....
   .....
   return e;
}
```

Function Result

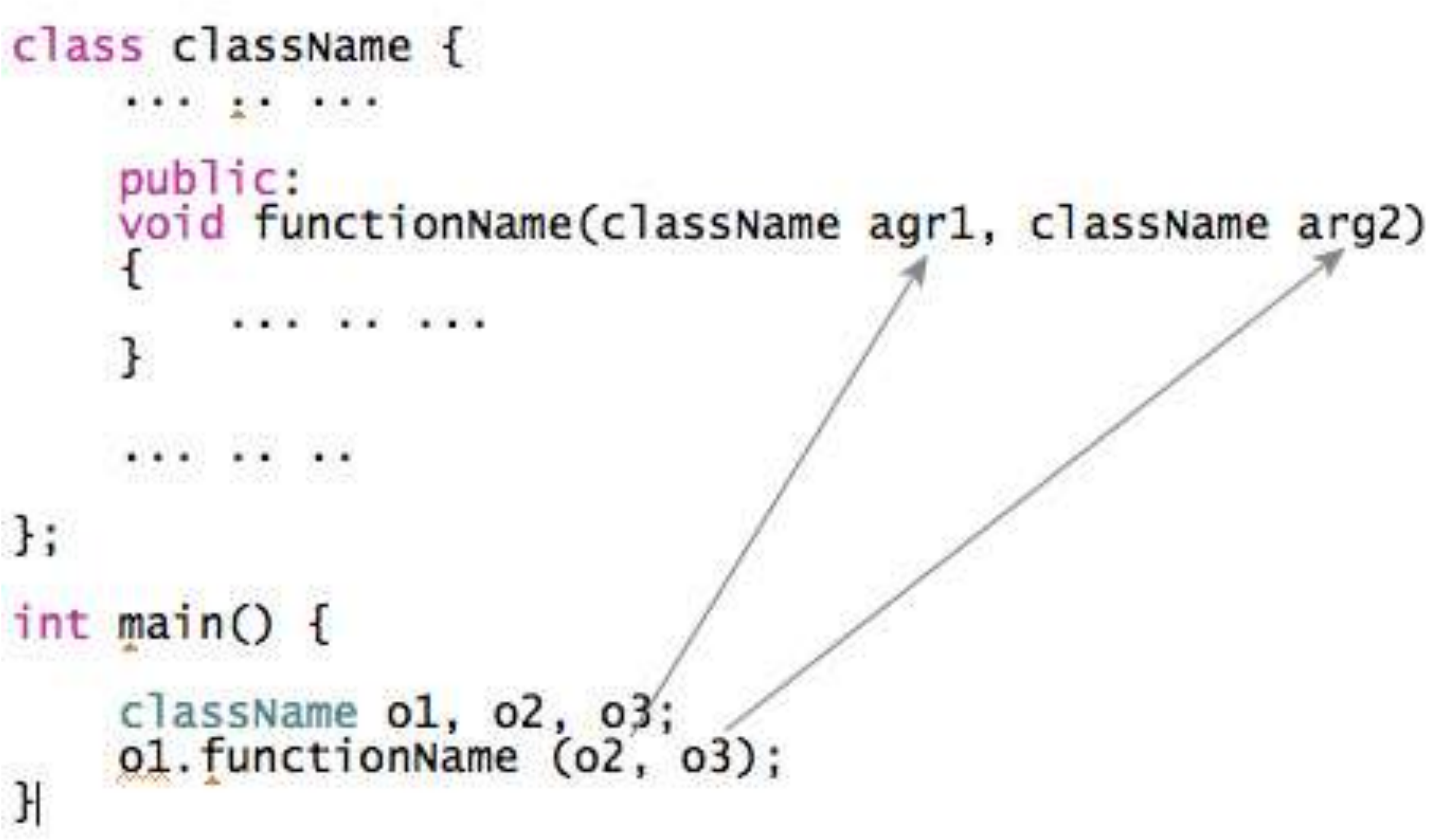
Object as Function arguments



```
void add(int x, int y)
{
   statements…
}
int main()
{
   int a=5,b=6;
   add(a,b);
}
```

```
void addtime(time x, time y)
{
   statements…
}
int main()
{
   time t1,t2,t3;
   t3.addtime(t1,t2);
}
```

Object as Function arguments



```
class className {
    ... .. ...

    public:
    void functionName(className agr1, className arg2)
    {
        ... .. ...
    }

    ... .. ..

};

int main() {

    className o1, o2, o3;
    o1.functionName (o2, o3);
}
```

Program: passing object as argument

```
class Time
{
     int hour, minute, second;
   public :
     void getTime(){
       cout<<"\nEnter hours:";cin>>hour;
       cout<<"Enter Minutes:";cin>>minute;
       cout<<"Enter Seconds:";cin>>second;
     }
     void printTime(){
       cout<<"\nhour:"<<hour;
       cout<<"\tminute:"<<minute;
       cout<<"\tsecond:"<<second;
     }
     void addTime(Time x, Time y){
       hour = x.hour + y.hour;
       minute = x.minute + y.minute;
       second = x.second + y.second;
     }
};
```

Program: passing object as argument

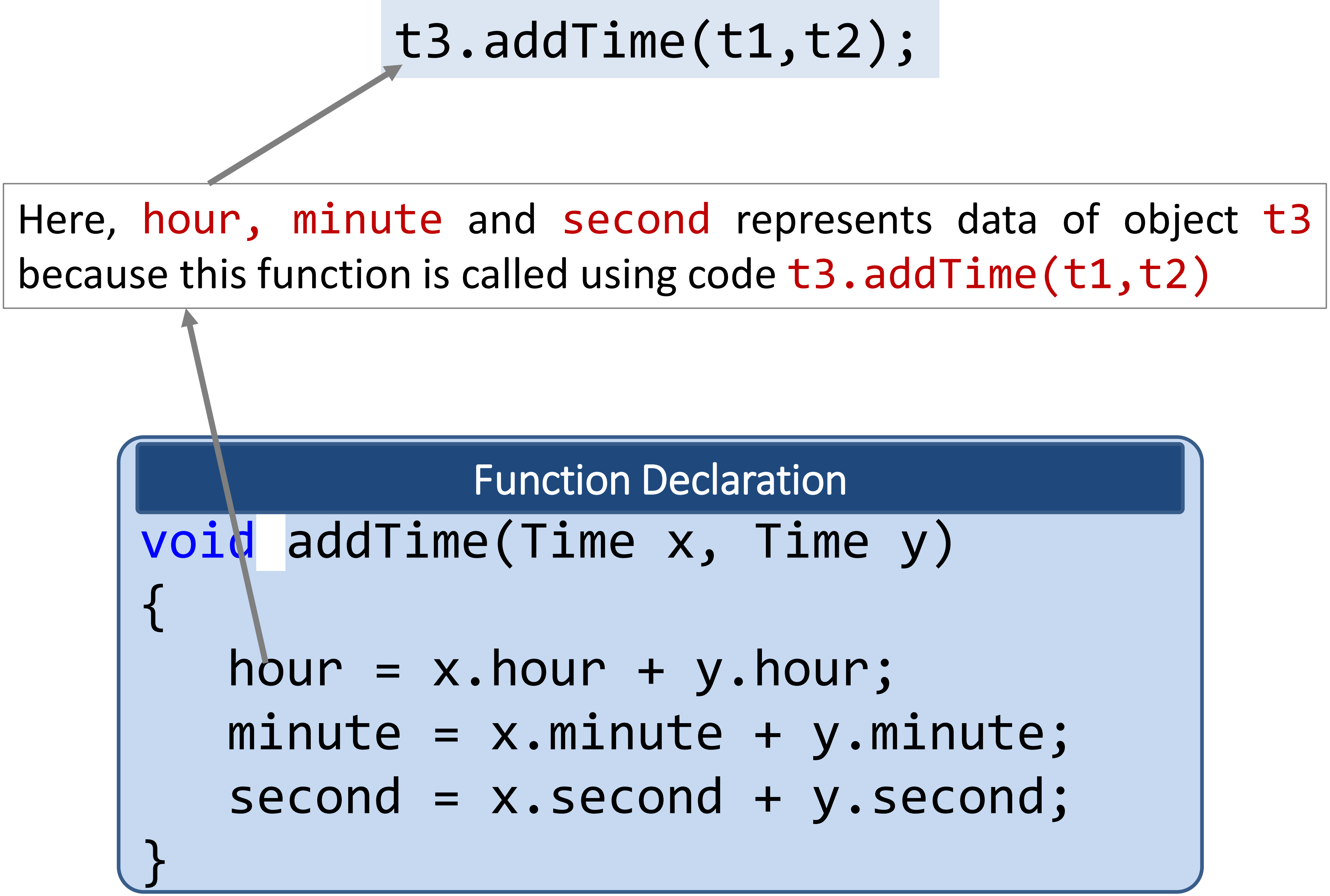
```
int main()
{
   Time t1,t2,t3;

   t1.getTime();
   t1.printTime();

   t2.getTime();
   t2.printTime();

   t3.addTime(t1,t2);
   cout<<"\nafter adding two objects";
   t3.printTime();

   return 0;
}
```



```
t3.addTime(t1,t2);
```

Here, hour, minute and second represents data of object t3 because this function is called using code t3.addTime(t1,t2)

Function Declaration

```
void addTime(Time x, Time y)
{
    hour = x.hour + y.hour;
    minute = x.minute + y.minute;
    second = x.second + y.second;
}
```

Program: Passing object as argument

- Define class **`Complex`** with members **`real`** and **`imaginary`** . Also define function to **`setdata()`** to initialize the members, **`print()`** to display values and **`addnumber()`** that adds two complex objects.
- Demonstrate concept of passing object as argument.

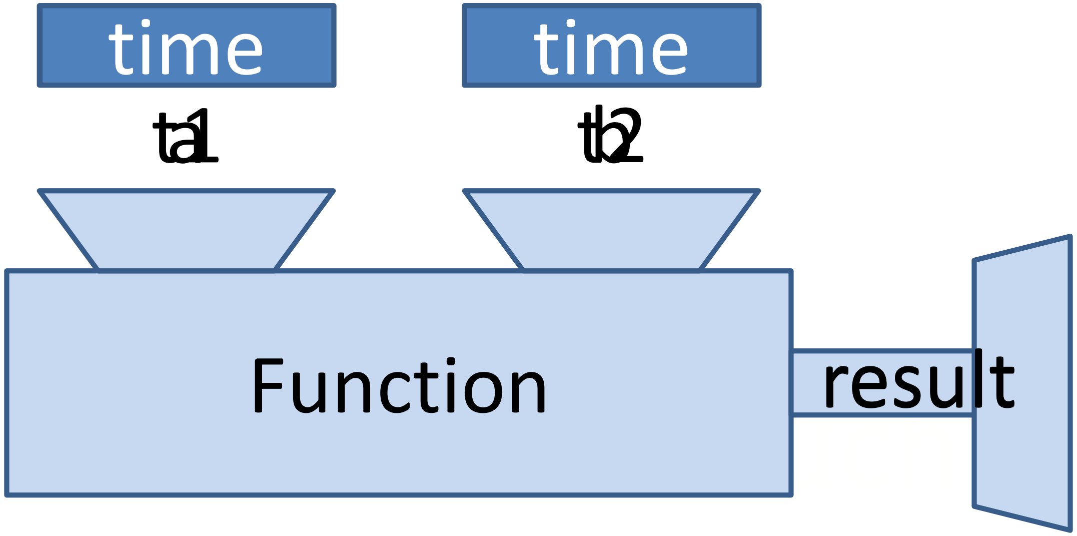
Program: Passing object as argument

```
class Complex
{
 private:
   int real,imag;
 public:
   void readData()
   {
     cout<<"Enter real and imaginary number:";
     cin>>real>> imag;
   }
   void addComplexNumbers(Complex comp1, Complex comp2)
   {
     real=comp1.real+comp2.real;
     imag=comp1.imag+comp2.imag;
   }
   void displaySum()
   {
     cout << "Sum = " << real<< "+" << imag << "i";
   }
};
```

```
int main()
{
   Complex c1,c2,c3;
   c1.readData();
   c2.readData();
   c3.addComplexNumbers(c1, c2);
   c3.displaySum();
}
```

Passing and Returning Objects

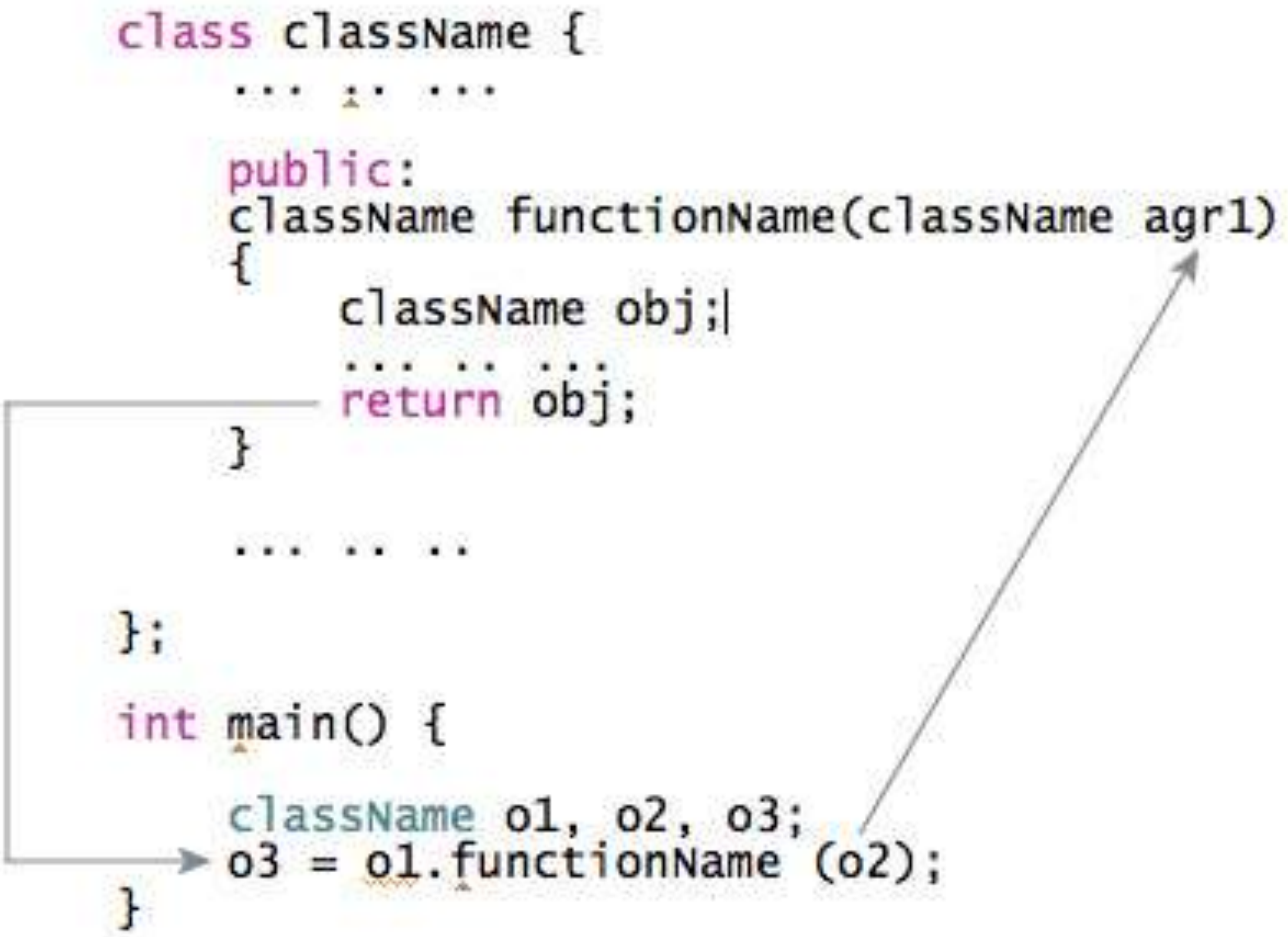
Passing and returning object



```
int add(int x, int y)
{
   return
}
int main()
{
   int a=5,b=6,result;
   result = add(a,b);
}
```

```
time addtime(time x, time y)
{
   return //object of class time
}
int main()
{
 time t1,t2,t3,result;
 result = t3.addtime(t1,t2);
}
```

Passing and returning object



```
class className {
    ... .. ...

    public:
    className functionName(className agr1)
    {
        className obj;
        ... .. ...
        return obj;
    }

    ... .. ..

};

int main() {

    className o1, o2, o3;
    o3 = o1.functionName (o2);
}
```

Program: Passing and Returning an Object

- Define class `Time` with members `hour, minute` and `second`. Also define function to `getTime()` to initialize the members, `printTime()` to display time and `addTime()` to add two time objects. Demonstrate class `Time`.

  1. Passing object as argument
  2. Returning object

Program: Returning object

```cpp
class Time{
     int hour, minute, second;
   public :
     void getTime(){
      cout<<"\nEnter hours:";cin>>hour;
      cout<<"Enter Minutes:";cin>>minute;
     }
     void printTime(){
      cout<<"\nhour:"<<hour;
      cout<<"\tminute:"<<minute;
     }
     Time addTime(Time t1, Time t2){
      Time t4;
      t4.hour = t1.hour + t2.hour;
      t4.minute = t1.minute + t2.minute;
      return t4;
     }
};
```

Program: Returning object

```
int main()
{
  Time t1,t2,t3,ans;

  t1.getTime();
  t1.printTime();

  t2.getTime();
  t2.printTime();

  ans=t3.addTime(t1,t2);
  cout<<"\nafter adding two objects";
  ans.printTime();

  return 0;
}
```

Program: Returning object

- C++ program to add two complex numbers by **Pass and Return Object** from the Function.

Program: Returning object

```
class Complex
{
 private:
   int real,imag;
 public:
   void readData()
   {
     cout<<"Enter real and imaginary number:";
     cin>>real>> imag;
   }
   Complex addComplexNumbers(Complex comp1, Complex comp2)
   {
     Complex temp;
     temp.real=comp1.real+comp2.real;
     temp.imag=comp1.imag+comp2.imag;
     return temp;
   }
   void displaySum()
   {
     cout << "Sum = " << real<< "+" << imag << "i";
   }
};
```

Program: Returning object

```
int main()
{
   Complex c1,c2,c3,ans;
   c1.readData();
   c2.readData();
   ans = c3.addComplexNumbers(c1, c2);
   ans.displaySum();
}
```

Nesting Member Functions

Nesting Member functions

- A member function of a class can be called by an object of that class using dot operator.
- A member function can be also called by another member function of same class.
- This is known as nesting of member functions.

```
void set_values (int x, int y)
{
   width = x;
   height = y;

   printdata();
}
```

Program: Nesting member function

- Define class **Rectangle** with member **width,height**. Also define function to **setvalue()**, **displayvalue()**. Demonstrate nested member functions.

Program: Nesting member function

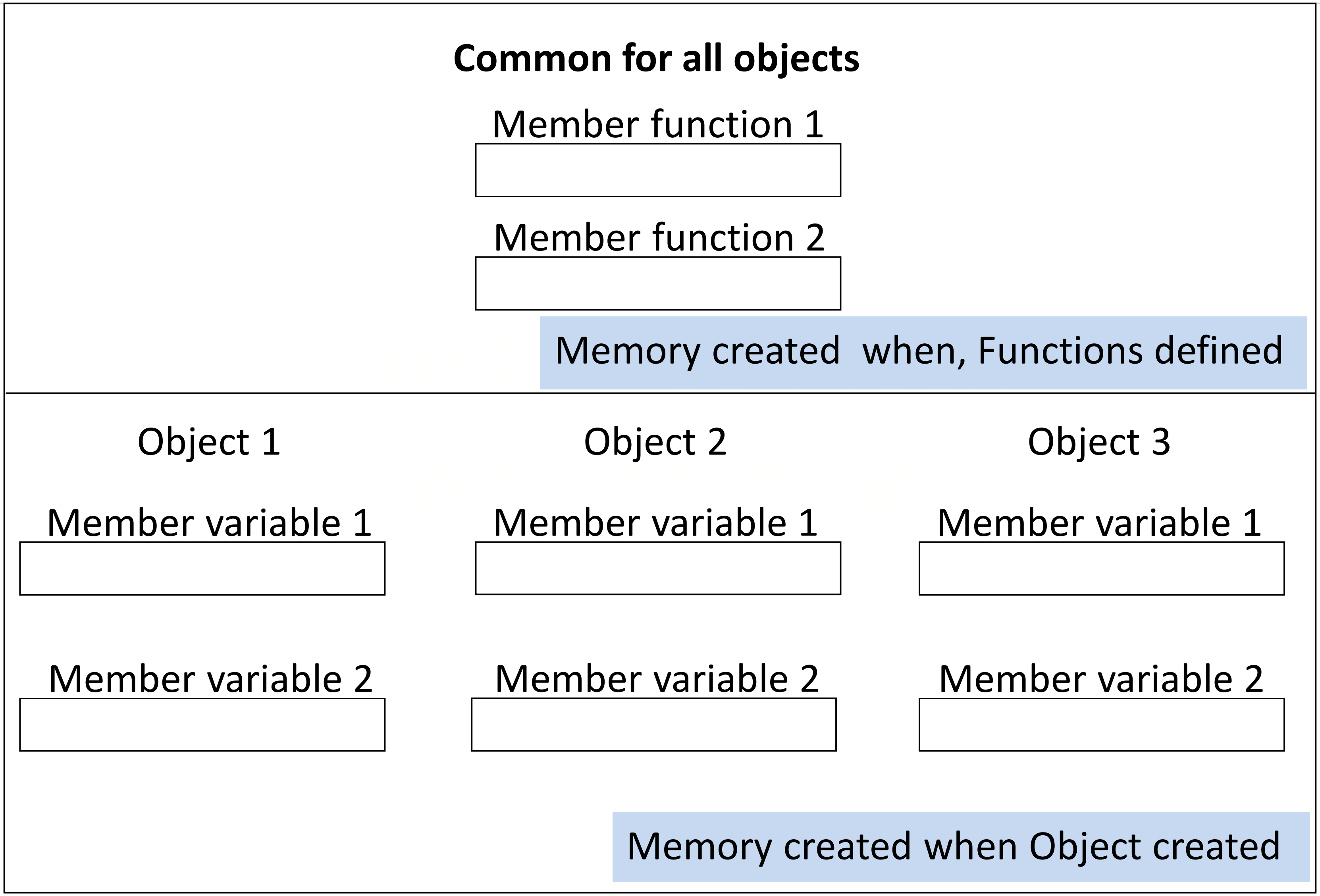
```
class rectangle{
  int w,h;
  public:
  void setvalue(int ww,int hh)
  {
    w=ww;
    h=hh;
    displayvalue();
  }
  void displayvalue()
  {
    cout<<"width="<<w;
    cout<<"\t height="<<h;
  }
};
```

```
int main(){
  rectangle r1;
  r1.setvalue(5,6);
  r1.displayvalue();
  return 0;
}
```

Memory allocation of objects

- The **member functions** are created and placed in the memory space **only once** at the time they are defined as part of a class specification.
- No separate space is allocated for member functions when the **objects** are created.
- Only space for **member variable** is allocated separately for each **object** because, the member variables will hold different data values for different objects.

Memory allocation of objects(Cont…)



Common for all objects

Member function 1

Member function 2

Memory created when, Functions defined

Object 1

Member variable 1

Member variable 2

Object 2

Member variable 1

Member variable 2

Object 3

Member variable 1

Member variable 2

Memory created when Object created

```
class Account
{
  int Account_no,Balance;
  char Account_type[10];
 public:
  void setdata(int an,char at[],int bal)
  {
      Account_no = an;
      Account_type = at;
      Balance = bal;
  }
};
```

```
int main(){
 Account A1,A2,A3;
 A1.setdata(101,“Current“,3400);
 A2.setdata(102,“Saving“,150);
 A3.setdata(103,“Current“,7900);
 return 0;
}
```

Object	A1
Account No	
Account Type	
Balance	

Object	A2
Account No	
Account Type	
Balance	

Object	A3
Account No	
Account Type	
Balance	

Static Data members / variables

Static Data members

A static data member is useful,
when all objects of the same class must **share a common information**.

Just write static keyword prefix to regular variable

It is initialized to zero when first object of class created

Only one copy is created for each object

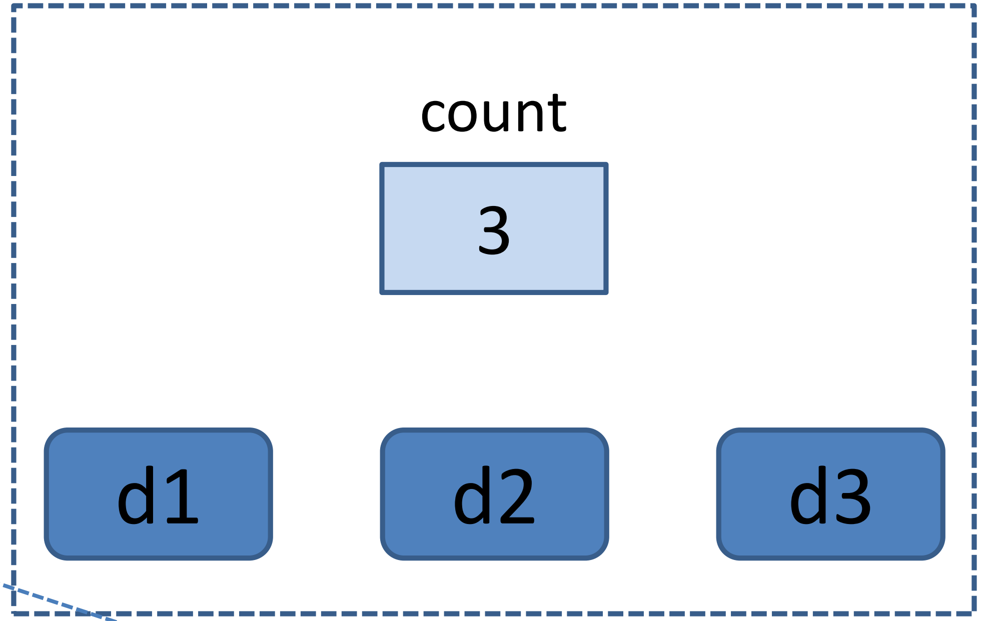
Its life time is entire program

Static Data members

```cpp
class demo
{
  static int count;
  public:
   void getcount()
   {
    cout<<"count="<<++count;
   }
};

int demo::count;

int main()
{
  demo d1,d2,d3;
  d1.getcount();
  d2.getcount();
  d3.getcount();
  return 0;
}
```

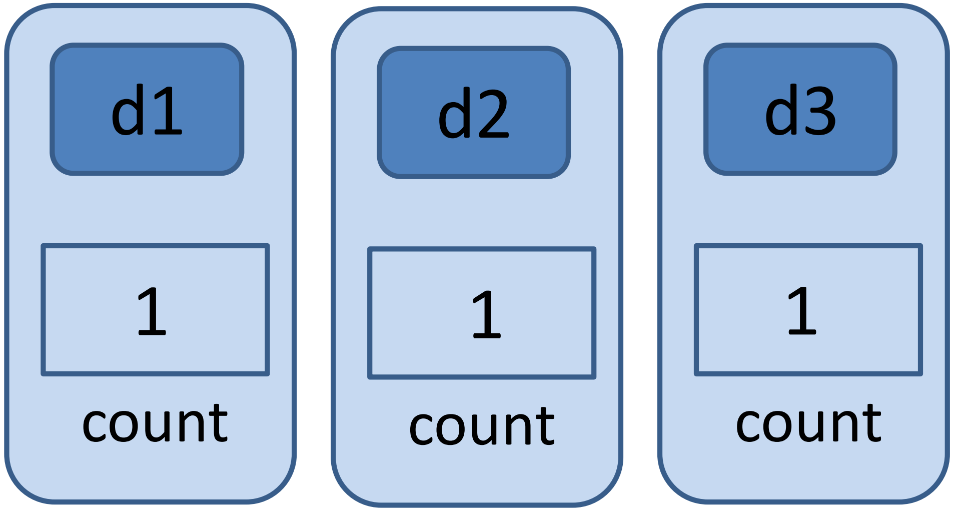


Static members are **declared** inside the class and **defined** outside the class.

Regular Data members

```
class demo
{
  int count;

  public:
    void getcount()
    {
      count = 0;
      cout<<"count="<< ++count;
    }
};
int main()
{
  demo d1,d2,d3;
  d1.getcount();
  d2.getcount();
  d3.getcount();
  return 0;
}
```



Static Data Members

- Data members of the class which are shared by all objects are known as **static** data members.
- **Only one copy** of a static variable is maintained by the class and it is common for all objects.
- **Static members** are **declared** inside the class and **defined** outside the class.
- It is initialized to **zero** when the first object of its class is created.
- you cannot initialize a static member variable inside the class declaration.
- It is visible only within the class but its lifetime is the entire program.
- **Static members** are generally used to maintain values common to the entire class.

Program : Static data member

```
class item
{
   int number;
   static int count;// static variable declaration
   public:
   void getdata(int a){
     number = a;
     count++;
   }
   void getcount(){
     cout<<"\nvalue of count: "<<count;
   }
};
int item :: count;   // static variable definition
```

Program : Static data member

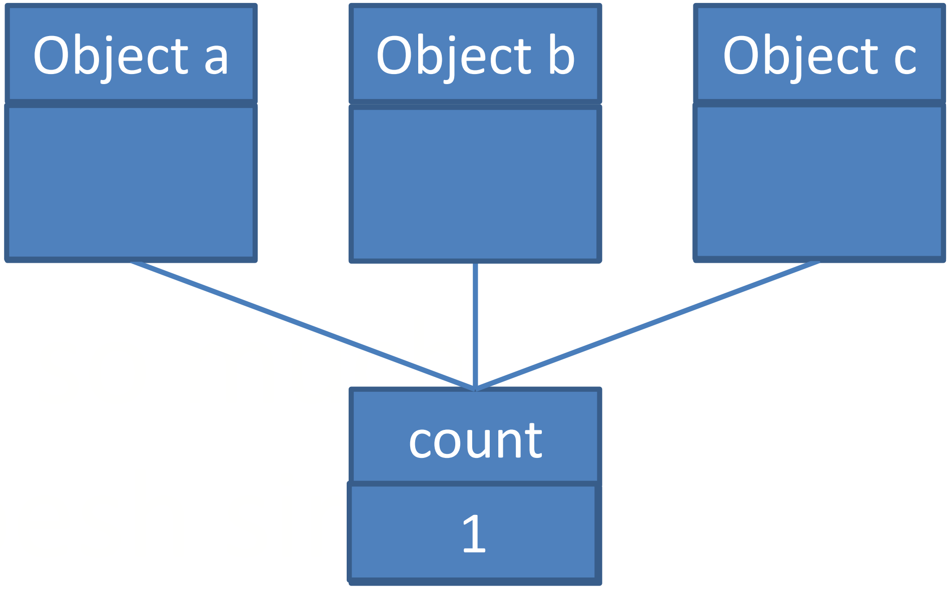
```
int main()
{
   item a,b,c;

   a.getdata(100);
   a.getcount();

   b.getdata(200);
   a.getcount();

   c.getdata(300);
   a.getcount();

   return 0;
}
```



Output:

```
value of count: 1
value of count: 2
value of count: 3
```

Program : Static data member

```
class shared {
   static int a;
   int b;
public:
   void set(int i, int j) {a=i; b=j;}
   void show();
} ;
int shared::a;
void shared::show()
{
   cout << "This is static a: " << a;
   cout << "\nThis is non-static b: " << b; cout << "\n";
}
```

```
int main() {
   shared x, y;
   x.set(1, 1);
   x.show();
   y.set(2, 2);
   y.show();
   x.show();
   return 0;
}
```

- static variable **a** declared inside the class but, storage is not allocated

- variable **a** declared outside the class using scope resolution operator.

- Storage for the variable will be allocated

Program : Static data member

```
class A
{
    int x;
public:
    A()
    {
     cout << "A's constructor called " << endl;
    }
};

class B
{
    static A a;
public:
    B()
    {
        cout << "B's constructor called " << endl;
    }
};

A B::a;  // definition of a

int main()
{
    B b1, b2, b3;
    return 0;
}
```

Output:

```
A's constructor called
B's constructor called
B's constructor called
B's constructor called
```

Static Member Functions

Static Member Functions

- **Static member functions** can access only static members of the class.
- **Static member functions** can be invoked using class name, not object.
- There cannot be static and non-static version of the same function.

- They cannot be **virtual**.
- They cannot be declared as **constant** or **volatile**.
- A static member function does not have **this pointer**.

Program: Static Member function

```
class item
{
    int number;
    static int count;// static variable declaration
    public:
    void getdata(int a){
      number = a;
      count++;
    }
    static void getcount(){
      cout<<”value of count: “<<count;
    }
};
int item :: count; // static variable definition
```

Program: Static Member function

```
int main()
{
   item a,b,c;

   a.getdata(100);
   item::getcount();

   b.getdata(200);
   item::getcount();

   c.getdata(300);
   item::getcount();
   return 0;
}
```

Output:

```
value of count: 1
value of count: 2
value of count: 3
```

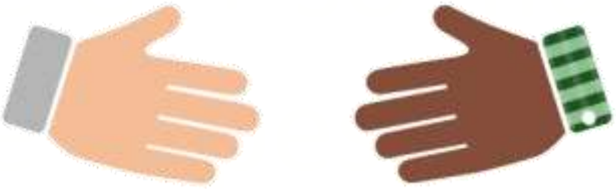
Friend Function

Friend Function

- In C++ a **Friend Function** that is a "friend" of a given class is allowed **access to private and protected data** in that class.
- A friend function is a function which is declared using **friend** keyword.

Class

```
class A
{
  private:
    int numA;
  public:
   void setA();
   friend void add();
 };
```



Friend Function

```
void add()
{       Access
        numA, numB
}
```

Class

```
class B
{
  private:
    int numB;
   public:
    void setB();
    friend void add();
};
```

Friend Function

- **Friend function** can be declared either in public or private part of the class.
- It is not a member of the class so it **cannot be called using the object**.
- Usually, it has the **objects as arguments**.

Syntax:

```
class ABC
{
   public:
      .............................................
      friend void xyz(argument/s); //declaration
      .............................................
};
```

Program: Friend Function

```
class numbers {
  int num1, num2;
  public:
   void setdata(int a, int b);
   friend int add(numbers N);
};
void numbers :: setdata(int a, int b){
  num1=a;
  num2=b;
}
int add(numbers N){
  return (N.num1+N.num2);
}
```

```
int main()
{
  numbers N1;
  N1.setdata(10,20);
  cout<<”Sum = ”<<add(N1);
  return 0;
}
```

Program: Friend Function

```
class Box {
   double width;
public:
   friend void printWidth( Box );
   void setWidth( double wid );
};
void Box::setWidth( double wid ) {
   width = wid;
}
void printWidth(Box b) {
  cout << "Width of box : " << b.width;
}
int main( ) {
Box box;
box.setWidth(10.0);
printWidth( box );
return 0;
}
```

Program: Friend Function

```
class  base
{
    int val1,val2;
   public:
    void get(){
       cout<<"Enter two values:";
       cin>>val1>>val2;
    }
    friend float mean(base ob);
};
float mean(base ob){
   return float(ob.val1+ob.val2)/2;
}
int main(){
    base obj;
    obj.get();
    cout<<"\n Mean value is : "<<mean(obj);
}
```

Member function, friend to another class

```
class X {
 ........................................
 int f();
};
class Y{
 ........................................
 friend int X :: f();
};
```

- Member functions of one class can be made **friend function** of another class.
- The function **f** is a member of **class X** and a friend of **class Y**.

Friend function to another class

Class

```
class A
{
  private:
    int numA;
  public:
   void setA();
   friend void add();
 };
```

Friend Function

```
void add()
{
     Access
     numA, numB
}
```

Class

```
class B
{
  private:
    int numB;
   public:
    void setB();
   friend void add();
};
```

Program: Friend function to another class

- Write a program to find out sum of two private data members numA and numB of two classes ABC and XYZ using a common friend function. Assume that the prototype for both the classes will be `int add(ABC, XYZ);`

Program: Friend to another class

```
class ABC {
 private:
  int numA;
 public:
  void setdata(){
   numA=10;
  }
 friend int add(ABC, XYZ);
};
```

```
class XYZ {
 private:
  int numB;
 public:
  void setdata(){
    numB=25;
  }
 friend int add(ABC , XYZ);
};
```

```
int add(ABC objA, XYZ objB){
  return (objA.numA + objB.numB);
}
int main(){
  ABC objA;   XYZ objB;
  objA.setdata();  objB.setdata();
  cout<<"Sum: "<< add(objA, objB);
}
```

Program: Friend to another class

```
class Square;  // forward declaration
class Rectangle
{
   int width=5, height=6;
   public:
     friend void display(Rectangle , Square );
};
class Square
{
   int side=9;
   public:
     friend void display(Rectangle , Square );
};
void display(Rectangle r, Square s)
{
   cout<<"Rectangle:"<< r.width * r.height;
   cout<<"Square:"<< s.side * s.side;
}
```

Program: Friend to another class

```
int main () {
  Rectangle rec;
  Square sq;
  display(rec,sq);
  return 0;
}
```

Use of friend function

- It is possible to grant a nonmember function access to the private members of a class by using a **friend function**.
- It can be used to **overload binary operators**.

Constructors

What is constructor ?

A **constructor** is a block of code which is,

similar to member function

has same name as class name

called automatically when object of class created

A **constructor** is used to initialize the objects of class as soon as the object is created.

Constructor

```
class car
{
  private:
    float mileage;
  public:
    void setdata()
    {
       cin>>mileage;
    }
};
```

Same name as class name

Similar to member function

```
class car
{
  private:
    float mileage;
  public:
   car()
   {
      cin>>mileage;
   }
};
```

```
int main()
{
    car c1,c2;
    c1.setdata();
    c2.setdata();
}
```

Called automatically on creation of object

```
int main()
{
 car c1,c2;


}
```

Properties of Constructor

- **Constructor** should be declared in public section because private constructor cannot be invoked outside the class so they are useless.
- Constructors **do not have return types** and they cannot return values, not even void.

```
class car
{
  private:
    float mileage;
  public:
    car()
    {
     cin>>mileage;
    }
};
```

- Constructors **cannot be inherited**, even though a derived class can call the base class constructor.
- Constructors **cannot be virtual**.
- They make implicit calls to the operators **new** and **delete** when memory allocation is required.

Constructor (Cont…)

```
class Rectangle
{
 int width,height;
 public:
 Rectangle(){
   width=5;
   height=6;
   cout<<”Constructor Called”;
 }
};
int main()
{
  Rectangle r1;
  return 0;
}
```

Types of Constructors

Types of Constructors

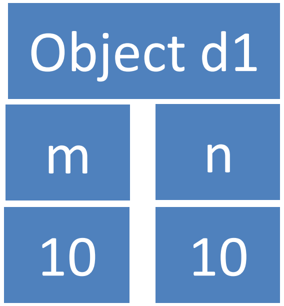
1) Default constructor
2) Parameterized constructor
3) Copy constructor

1) Default Constructor

- **Default constructor** is the one which invokes by default when object of the class is created.
- It is generally used to initialize the default value of the data members.
- It is also called **no argument constructor**.

```
class demo{
  int m,n;
 public:
  demo()
  {
    m=n=10;
  }
};
```

```
int main()
{
  demo d1;
}
```

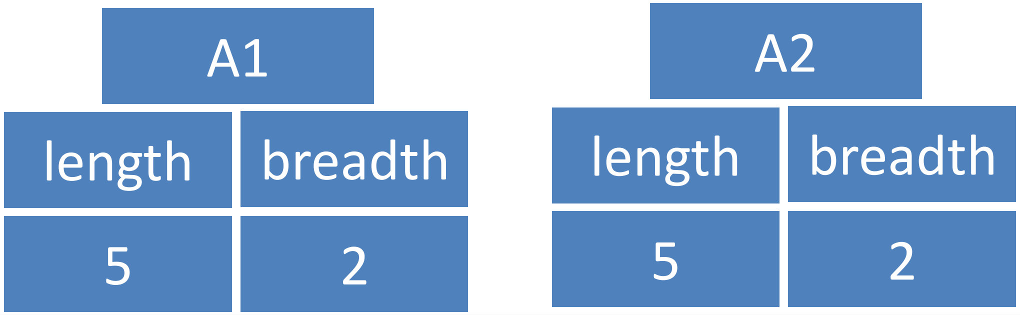


Object d1	
m	n
10	10

Program Constructor

```
class Area
{
  private:
   int length, breadth;
  public:
   Area(){
   length=5;
   breadth=2;
   }
   void Calculate(){
    cout<<"\narea="<<length * breadth;
   }
};
```

```
int main(){
 Area A1;
 A1.Calculate();
 Area A2;
 A2.Calculate();
 return 0;
}
```



A1	
length	breadth
5	2

A2	
length	breadth
5	2

2) Parameterized Constructor

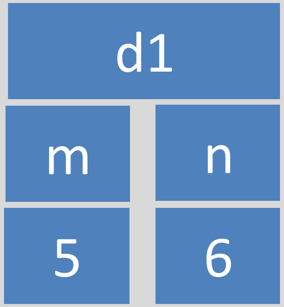
- Constructors that can take arguments are called **parameterized constructors.**
- Sometimes it is necessary to initialize the various data elements of different objects with different values when they are created.
- We can achieve this objective by passing arguments to the constructor function when the objects are created.

Parameterized Constructor

- Constructors that can take arguments are called **parameterized constructors**.

```
class demo
{
   int m,n;
   public:
   demo(int x,int y){ //Parameterized Constructor
     m=x;
     n=y;
     cout<<“Constructor Called“;
   }
};
int main()
{

}
```



d1

m n

5 6

Program Parameterized Constructor

- Create a class **Distance** having data members **feet** and **inch**. Create parameterized constructor to initialize members **feet** and **inch.**

3) Copy Constructor

- A **copy constructor** is used to declare and initialize an object from another object using an object as argument.
- For example:

```
demo(demo &d);    //declaration
demo d2(d1);            //copy object
```

OR

```
demo d2=d1;           //copy object
```

- Constructor which accepts a reference to its own class as a parameter is called **copy constructor**.

3) Copy Constructor

- A **copy constructor** is used to initialize an object

  from another object

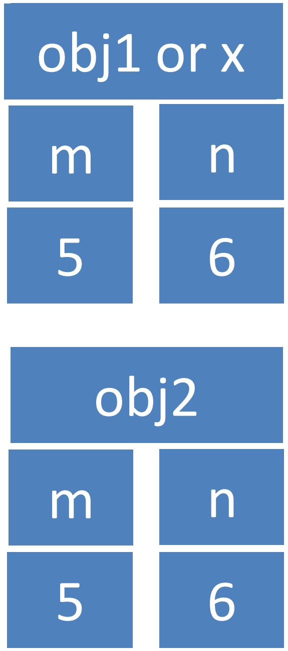
  using an object as argument.

- A Parameterized constructor which accepts a reference to its own class as a parameter is called **copy constructor**.

Copy Constructor

```
class demo
{
   int m, n;
   public:
   demo(int x,int y){
    m=x;
    n=y;
    cout<<"Parameterized Constructor";
   }
   demo(demo &x){
    m = x.m;
    n = x.n;
    cout<<"Copy Constructor";
   }
};
```

```
int main()
{
   demo obj1(5,6);
   demo obj2(obj1);
   demo obj2 = obj1;
}
```



obj1 or x

m	n
5	6

obj2

m	n
5	6

Program: Types of Constructor

- Create a class **`Rectangle`** having data members **`length`** and **`width`**. Demonstrate default, parameterized and copy constructor to initialize members.

Program: Types of Constructor

This is constructor overloading

```
class rectangle{
   int length, width;
   public:
   rectangle(){ // Default constructor
     length=0;
     width=0;
   }
   rectangle(int x, int y){// Parameterized
                                         constructor
     length = x;
     width = y;
   }
   rectangle(rectangle &_r){ // Copy constructor
     length = _r.length;
     width = _r.width;
   }
};
```

Program: Types of Constructor (Cont…)

```
int main()
{
  rectangle r1; // Invokes default constructor
  rectangle r2(10,20); // Invokes parameterized
                          constructor
  rectangle r3(r2); // Invokes copy constructor
}
```

Destructor

Destructor

- **Destructor** is used to destroy the objects that have been created by a constructor.
- The syntax for **destructor** is same as that for the constructor,
  - the class name is used for the name of destructor,
  - with a **tilde (~)** sign as prefix to it.

```
class car
{
    float mileage;
  public:
   car(){
     cin>>mileage;
   }

   ~car(){
cout<<" destructor";
   }

};
```

Destructor

- never takes any argument nor it returns any value nor it has return type.
- is invoked automatically by the complier upon exit from the program.
- should be declared in the public section.

Program: Destructor

```
class rectangle
{
  int length, width;
  public:
  rectangle(){ //Constructor
   length=0;
   width=0;
   cout<<”Constructor Called”;
  }
  ~rectangle() //Destructor
  {
   cout<<”Destructor Called”;
  }
// other functions for reading, writing and
processing can be written here
};
```

```
int main()
{
   rectangle x;
// default
constructor is
called
}
```

Program: Destructor

```
class Marks{
public:
   int maths;
   int science;
   //constructor
   Marks() {
      cout << "Inside Constructor"<<endl;
      cout << "C++ Object created"<<endl;
   }
  //Destructor
   ~Marks() {
      cout << "Inside Destructor"<<endl;
      cout << "C++ Object destructed"<<endl;
   }
};
```

```
int main( )
{
   Marks m1;
   Marks m2;
   return 0;
}
```

Operator Overloading

Operator Overloading

```
int a=5, b=10,c;
c = a + b;
```

Operator **+** performs **addition** of **integer operands** a, b

```
time t1,t2,t3;
t3 = t1 + t2;
```

Operator **+** performs **addition** of **objects** of type time

```
string str1=“Hello”
string str2=“Good Day”;
string str3;
str3 = str1 + str2;
```

Operator **+ concatenates** two strings str1,str2

Operator overloading

- **Function overloading** allow you to use same function name for different definition.
- **Operator overloading** extends the overloading concept to operators, letting you assign multiple meanings to C++ operators
- **Operator overloading** giving the normal C++ operators such as +, * and == additional meanings when they are applied with **user defined data types**.

Some of C++ Operators are already overloaded

Operator	Purpose
*	As pointer, As multiplication
<<	As insertion, As bitwise shift left
&	As reference, As bitwise AND

Operator Overloading

```
int a=5, b=10,c;
c = a + b;
```

Operator + performs addition of integer operands a, b

```
class time
{
  int hour, minute;
};
```

```
time t1,t2,t3;
t3 = t1 + t2;
```

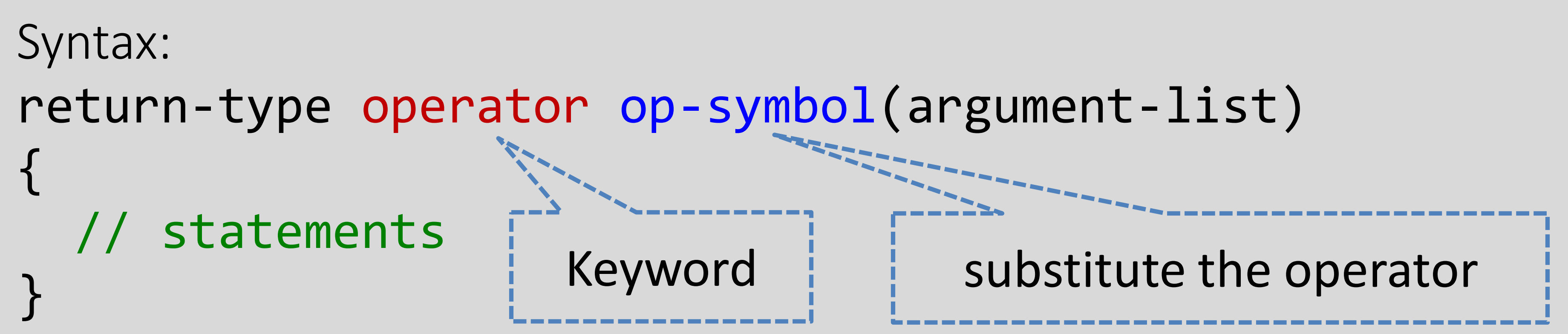
Operator + performs addition of objects of type time t1,t2

```
string str1=“Hello”,str2=“Good Day”;
str1 + str2;
```

Operator + concatenates two strings str1,str2

Operator Overloading

- Specifying more than one definition for an **operator** in the same scope, is called **operator overloading**.
- You can overload operators by creating ***“operator functions”***.



Syntax:

```
return-type operator op-symbol(argument-list)
{
   // statements
}
```

Keyword

substitute the operator

Example:

```
void operator + (arguments);
int operator - (arguments);
class-name operator / (arguments);
float operator * (arguments);
```

Overloading Binary operator +

```
class complex{
  int real,imag;
  public:
    complex(){
     real=0; imag=0;
    }
    complex(int x,int y){
     real=x; imag=y;
    }
    void disp(){
     cout<<"\nreal value="<<real<<endl;
     cout<<"imag value="<<imag<<endl;
    }
    complex operator + (complex);
};
complex complex::operator + (complex c){
  complex tmp;
  tmp.real = real + c.real;
  tmp.imag = imag + c.imag;
  return tmp;
}
```

```
int main()
{
  complex c1(4,6),c2(7,9);
  complex c3;
  c3 = c1 + c2;
  c1.disp();
  c2.disp();
  c3.disp();
  return 0;
}
```

Similar to function call

`c3=c1.operator +(c2);`

Binary Operator Arguments

```
result = obj1.operator symbol (obj2);//function notation
```

```
result = obj1 symbol obj2;           //operator notation
```

```
complex operator + (complex x)
{
   complex tmp;
   tmp.real = real + x.real;
   tmp.imag = imag + x.imag;
   return tmp;
}
```

```
result = obj1.display();
```

```
void display()
{
   cout<<"Real="<<real;
   cout<<"Imaginary="<<imag;
}
```

Operator Overloading

- **Operator overloading** is compile time polymorphism.
- You can overload most of the built-in operators available in C++.

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Operator Overloading using Friend Function

Invoke Friend Function in operator overloading

```
result = operator symbol (obj1,obj2);//function notation
```

```
result = obj1 symbol obj2;           //operator notation
```

```
friend complex operator +(complex c1,complex c2)
{
  complex tmp;
  tmp.r=c1.r+c2.r;
  tmp.i=c1.i+c2.i;
  return tmp;
}
```

```
int main()
{
  complex c1(4,7),c2(5,8);
  complex c3;
  c3 = c1 + c2;
  c3 = operator +(c1,c2);
}
```

Overloading Binary operator ==

```
class complex{
  int r,i;
  public:
  complex(){
    r=i=0;}
  complex(int x,int y){
    r=x;
    i=y;}
  void display(){
   cout<<"\nreal="<<r<<endl;
   cout<<"imag="<<i<<endl;}
   int operator==(complex);
};
int complex::operator ==(complex c){
  if(r==c.r && i==c.i)
    return 1;
  else
    return 0;}
```

```
int main()
{
  complex c1(5,3),c2(5,3);
  if(c1==c2)
    cout<<"objects are equal";
  else
    cout<<"objects are not equal";
    return 0;
     }
```

Overloading Unary Operator

Overloading Unary operator –

```
class space {
  int x,y,z;
  public:
  space(){
    x=y=z=0;}
  space(int a, int b,int c){
    x=a; y=b; z=c; }
  void display(){
   cout<<"\nx="<<x<<",y="<<y<<",z="<<z;
  }
  void operator-();
};
void space::operator-() {
  x=-x;
  y=-y;
  z=-z;
}
```

```
int main()
{
  space s1(5,4,3);
  s1.display();
  -s1;
  s1.display();
  return 0;
        }
```

Overloading Unary operator --

```
class space {
  int x,y,z;
  public:
  space(){
    x=y=z=0;}
  space(int a, int b,int c){
    x=a; y=b; z=c; }
  void display(){
   cout<<"\nx="<<x<<",y="<<y<<",z="<<z;
  }
  void operator--();
};
void space::operator--() {
  x--;
  y--;
  z--;
}
```

```
int main()
{
  space s1(5,4,3);
  s1.display();
  --s1;
  s1.display();
  return 0;
        }
```

Overloading Prefix and Postfix operator

```cpp
class demo
{
    int m;
    public:
     demo(){ m = 0;}
     demo(int x)
     {
       m = x;
     }
     void operator ++()
     {
        ++m;
        cout<<"Pre Increment="<<m;
     }
     void operator ++(int)
     {
        m++;
        cout<<"Post Increment="<<m;
     }
};
```

```cpp
int main()
{
    demo d1(5);
    ++d1;
    d1++;
}
```

Invoking Operator Function

- Binary operator

```
operand1 symbol operand2
```

- Unary operator

```
operand symbol
symbol operand
```

- Binary operator using friend function

```
operator symbol (operand1,operand2)
```

- Unary operator using friend function

```
operator symbol (operand)
```

Rules for operator overloading

- Only existing operator can be overloaded.
- The overloaded operator must have at least one operand that is user defined type.
- We cannot change the basic meaning and syntax of an operator.

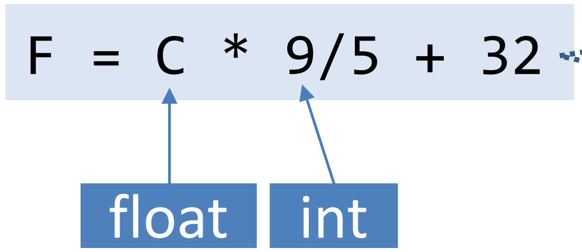
Rules for operator overloading (Cont…)

- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
- We cannot overload following operators.

Operator	Name
. and .*	Class member access operator
::	Scope Resolution Operator
sizeof()	Size Operator
?:	Conditional Operator

Type Conversion

Type Conversion

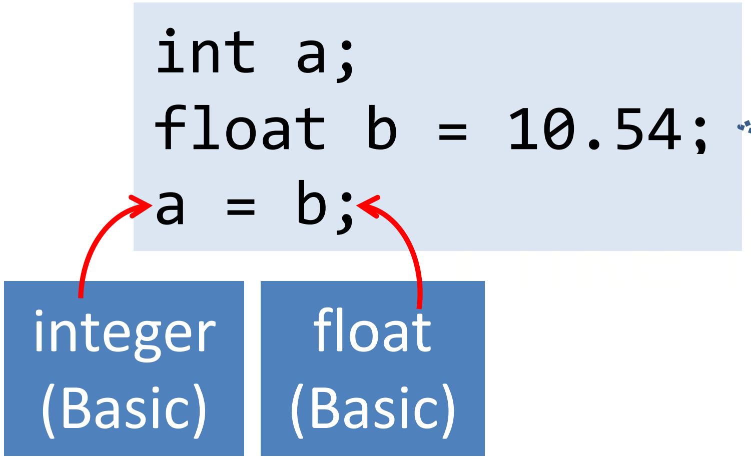


```
F = C * 9/5 + 32
```

float

int

If different data types are mixed in expression, C++ applies automatic type conversion as per certain rules.



```
int a;
float b = 10.54;
a = b;
```

integer (Basic)

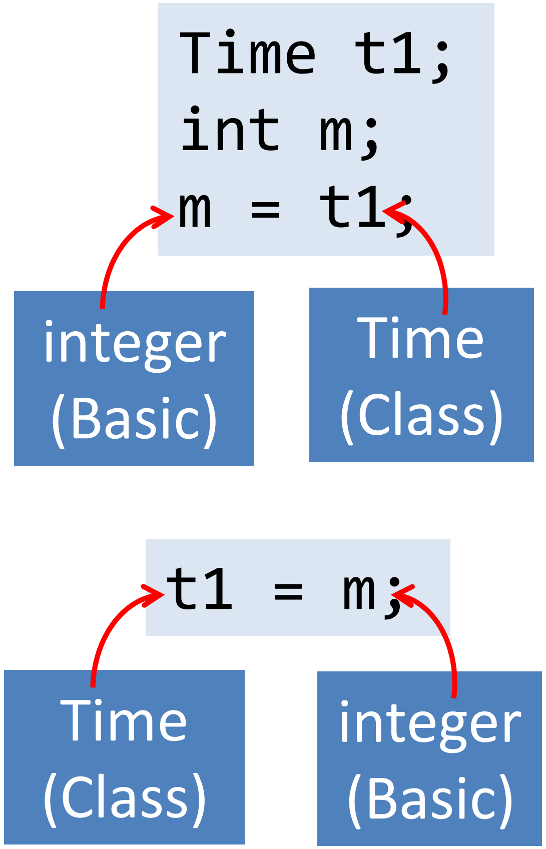
float (Basic)

a = 10;

- float is converted to integer automatically by complier.
- basic to basic type conversion.

- An assignment operator causes automatic type conversion.
- The data type to the right side of assignment operator is automatically converted data type of the variable on the left.

Type Conversion



```
Time t1;
int m;
m = t1;
```

integer (Basic)

Time (Class)

```
t1 = m;
```

Time (Class)

integer (Basic)

- class type will not be converted to basic type OR basic type will not be converted class type automatically.

Type Conversion

- C++ provides mechanism to perform automatic type conversion if all variable are of **basic type**.
- For user defined data type programmers have to convert it by using **constructor** or by using **casting operator**.

- Three type of situation arise in user defined data type conversion.
  1. Basic type to Class type (Using Constructors)
  2. Class type to Basic type (Using Casting Operator Function)
  3. Class type to Class type (Using Constructors & Casting Operator Functions)

(1) Basic to class type conversion

- Basic to class type can be achieved **using constructor**.

```
class sample
{
  int a;
  public:
  sample(){}
  sample(int x){
    a=x;
  }
  void disp(){
    cout<<"The value of a="<<a;
  }
};
```

```
int main()
{
  int m=10;
  sample s;
  s = m;
  s.disp();
  return 0;
}
```

(2) Class to basic type conversion

- The Class type to Basic type conversion is done **using casting operator function**.
- The casting operator function should satisfy the following conditions.
  1. It must be a class member.
  2. It must not mention a return type.
  3. It must not have any arguments.

Syntax:

```
operator destinationtype()
{
    ....
    return
}
```

Program: Class to basic type conversion

```
class sample
{
   float a;
   public:
   sample()
   {
     a=10.23;
   }
   operator int() //Casting operator
                    function
   {
     int x;
     x=a;
     return x;
   }
};
```

```
int main()
{
  sample S;
  int y= S;//Class to Basic
             conversion
  cout<<"The value of y="<<y;
  return 0;
}
```

Explicit type conversion

`y = int (S);`

Automatic type conversion

`y = S;`

Program: Class to basic type conversion

```
class vector{
   int a[5];
   public:
   vector(){
      for(int i=0;i<5;i++)
           a[i] = i*2;
   }
   operator int();
};
vector:: operator int() {
   int sum=0;
   for(int i=0;i<5;i++)
      sum = sum + a[i];
   return sum;}
```

```
 int main()
{
vector v;
int len;
len = v;
cout<<“Length of V="<<len;
return 0;
}
```

(3) Class type to Class type

- It can be achieved by two ways
  1. Using constructor
  2. Using casting operator function

Program: Class type to Class type

```
class alpha
{
  int commona;
  public:
    alpha(){}
    alpha(int x)
    {
      commona = x;
    }
    int getvalue()
    {
      return commona;
    }
};
```

```
int main()
{
  alpha obja(10);
  beta objb(obja);
  beta objb(20);
  obja = objb;
 }
```

```
class beta
{
   int commonb;
   public:
     beta(){}
     beta(int x)
     {
       commonb = x;
     }
     beta(alpha temp)  //Constructor
     {
       commonb = temp.getvalue();
     }
     operator beta()  //operator function
     {
       return beta(commonb);
     }
};
```

Thank You