Classes

```
class point {
    double x, y;
  public:
    void print(){ cout << "(" << x << "," << y << ")";}
    void init(double u, double v) { x = u; y = v; }
    void plus(point c);
};
```

default protection: **private**

```
struct point {
    void print(){ cout << "(" << x << "," << y << ")";}
    void init(double u, double v) { x = u; y = v; }
    void plus(point c);
  private:
    double x, y;
};
```

default protection: **public**

≡

```
class point {
  public:
    void print(){ cout << "(" << x << "," << y << ")";}
    void init(double u, double v){ x = u; y = v; }
    void plus(point c);
  private:
    double x, y;
};
```

```
struct point {
  public:
    void print(){ cout << "(" << x << "," << y << ")";}
    void init(double u, double v){ x = u; y = v; }
    void plus(point c);
  private:
    double x, y;
};
```

Use struct for classes containing only public data:

```
struct SimplePoint{ int x, y;};
SimplePoint p1={3,4};
SimplePoint p2= {p1.x +2, p1.y+3};
```


C++ Classes/ Java Classes

C++

```
/////File: vector2.h

class Vector2
{ public:
      Vector2(float a, float b);
      float inner(const Vector2& a);
  private:
      float x, y;
};
```

```
/////File vector2.cpp,

#include "vector2.h"

Vector2::Vector2(float a, float b)
{  x = a; y = b;}

float Vector2::inner(const Vector2& a)
  {return(x * a.x + y * a.y);}
```

```
///// File client.cpp
#include "vector2.h"

{ …
Vector2 V1(2.1f, 3.4f);
Vector2 V2(3.3f, 6.8f);

float IP= V1.inner(V2);


Vector2* vp1= new Vector2(2.1f, 3.4f);
Vector2* vp2= new Vector2(3.3f, 6.8f);

IP= vp1->inner(*vp2);

delete vp1; delete vp2;

}
```

Java:

```
//////File Vector2.java

public class Vector2
{
  public Vector2(float a, float b)
    {x=a; y=b;}
  public float inner(Vector2 a)
    { return (x * a.x + y * a.y);}
  private float x, y;
}
```

Objects are always allocated dynamically in Java.
Object variables are always (hidden) pointers to dynamically allocated objects

---

```
///// File client.java

{…


Vector2 vp1= new Vector2(2.1f, 3.4f);
Vector2 vp2= new Vector2(3.3f, 6.8f);

float IP= vp1.inner(vp2);
```

The objects referenced by vp1,vp2 deleted by garbage collector when no longer accessible. }

Classes, Initializing Objects

```
class Date
  {public:                  // No constructor
    unsigned month, day, year;
    void display(){cout << month << '/' << day << '/' << year; }
  };


int main()
  { Date arival, departure;
    Date birthday = {11, 26, 1985};
    Date vacation = birthday;
    Date absence[12];
  }
```

array-style initialisation:
only for objects of class without explicit constructor.
only public members can be initialised.

```
class Date
  {public:
    unsigned month=1, day=1;
    unsigned year=1900;
    void display()
      {cout << month << '/'
            << day << '/' << year;
      }
  };
```

Illegal!!,
Initialiser not allowed for class member

Classes, Constructors

```
class Date
  {public:
      Date(unsigned m, unsigned d, unsigned y);  // constructor
      Date(){};                 // default constructor
      void display()
        {cout << month << '/' << day << '/' << year; }

   private:
      unsigned month, day, year;
  };
```

A constructor never returns a value. Its exclusive mission is to perform initialisation and to set up for a freshly allocated object.

```
Date:: Date(unsigned m , unsigned d, unsigned y)
  { month= m;
    day= d;          // initialisation by assignment
    year= y;
  }

Date birthday(3, 23, 1977);      // calls constructor with three arguments
Date birthday = Date(3, 23, 1977);  // alternative form

Date appointment;                // calls default constructor
Date appointment= Date();        // alternative form

Date week[7]=
  { Date(3,18,1996), Date(3,19,1996), Date(3,20,1996),
    Date(3,21,1996), Date(3,22,1996)};    //calls def.constr 2 times !
```

CONST Declarations

```
const double pi= 3.14159;     // must be initialised
pi= 7.1234   // ILLEGAL
const int K;            // ILLEGAL, must be initialised
K= 20;                  // ILLEGAL
```

Reference Declarations

```
int a= 0, b;
int &ra = a;        // ra is an alias for a
ra= 2;              // now   a== 2
int &c;             // ILLEGAL, must be initialised
ra= b               // does not make ra an alias for b
                    // but assigns the value of b to a
```

CONST and Reference Members

```
class A
  {public:
   private:
      const int K= 20;  // ILLEGAL, Initialiser not allowed for class member
  }

class B
  {public:
     B(int kinit)   // constructor
   private:
      const int K ;
  }

B :: B(int kinit){K= kinit;}
      // ILLEGAL, cannot assign to constant
```

?

Classes, member-initializers

```
class ClassX
  {public:
     ClassX(args);
     ... a1 ;    // member dclarations
     ... a2 ;
   private:
     ... b1 ;    // member dclarations
  }

ClassX:: ClassX(args)
  : a1(initval), a2(initval), b1(initval)         // init-list
  {
     constructor-body
  }
```

```
Date:: Date(unsigned m , unsigned d, unsigned y)
  : month(m), day(d), year(y)
  { //empty body
  }
```

```
class B
  {public:
     B(int kinit)   // constructor
   private:
      const int K ;
  }

B :: B(int kinit) :K(kinit) {}      // OK

B h(10), k(20);
```

Classes, Initializing const and & members of Objects

```
#include <iostream>

class Xyz
  {public:
     Xyz(int& a, int b)              // constructor
          : abc(a), ijk(b) {}        // must initialise using mem-initialiser
     void display()
          { cout << abc << " " << ijk << endl; }
     void increment() { abc += ijk; }

   private:
     int& abc;              // reference member
     const int ijk;         // const member
};


int main()
  { int u = 1, v = 20, w = 99, z = 500;
     Xyz  r(u,v), s(w, z);
     r.display(); s.display();
     r.increment();  s.increment();
     r.display(); s.display();
     cout << "Now u = " << u
          << ", w = " << w << endl;
     return(0);
  }
```

result:
1 20
99 500
21 20
599 500
Now u = 21, w = 599

Read-Only Parameters and Methods

```
class Vector
  {  public:
       Vector( ) { }
       Vector(double a, double b){x= a; y= b;};
       Vector add(const Vector& V);
       Vector subtr(const Vector& V);
       .......
     private:
       double x, y;
  };

Vector Vector:: add(const Vector& V)
     { return Vector(x + V.x, y + V.y);}

Vector Vector:: subtr(const Vector& V)
     { return Vector(x - V.x, y - V.y);}
```

V : read-only
reference parameter

```
int main()
{
Vector a(1.1, 2.5),  b(3.0, 4.0),  c(7.33, 8.0);
a =  b.add(c);
```

c will not be modified
but what about b ?

```
}
```

Read-Only Parameters and Methods

```
class Vector
  {  public:
       Vector( ) { }
       Vector(double a, double b){x= a; y= b;};
       Vector add(const Vector& V)const;
       Vector subtr(const Vector& V)const;
       .....
     private:
       double x, y;
  };
```

read-only methods

read-only methods do not change the object.

Not in Java!
A **finale** method in Java cannot be changed by derivation!

```
Vector Vector:: add(const Vector& V)const
    { return Vector(x + V.x, y + V.y);}
Vector Vector:: subtr(const Vector& V)const
    { return Vector(x - V.x, y - V.y);}
```

```
a =  b.add(c);
```

c will not be modified

b will not be modified

Read-Only Parameters and Methods

```
class Set
  {public:
     Set();
     bool inset(int n) const;
     Set intersect(const Set&)const;
     void insert(int newelem);
     ...
   private:
     enum{MAX=100};
     int setrepr[MAX];
     int top;
  };
```

y must not be modified

```
Set Set::intersect(const Set& y)const
  {Set tmp; int n;
   for(int i=0; i< top; i++)

     {if(y.inset(n=setrepr[i])) tmp.insert(n);}

   return(tmp);
  }
```

y's inset method does not modify y

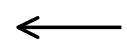
Constructors as Conversions

```
class Vector
{ public:
    Vector( ) { }
    Vector(double a, double b){x= a; y= b;};
    Vector(double a){x=a; y=0;}          // defines implicit conversion!!
    Vector add(const Vector& V)const;
    Vector subtr(const Vector& V)const;
    double getX()const {return x;};
    double getY()const {return y;};
  private:
    double x, y;
};

Vector Vector::add(const Vector& V)const
  { return Vector(x + V.x, y + V.y);}

Vector Vector:: subtr(const Vector& V)const
  { return Vector(x - V.x, y - V.y);}


int main()
{ Vector a(1.1, 2.5),  b(3.0, 4.0),  c(7.33, 8.0);
  a =  b.add(c);
  cout << a.getX() << ", " << a.getY()<< endl;
  a= b.add(2.0);   // ~ b.add(Vector(2.0))
  cout << a.getX() << ", " << a.getY()<< endl;
}
```



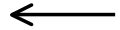
output:
10.33, 12
5, 4

Constructors as Conversions

explicit: suppress implicit conversion:

```
class Vector
{ public:
    Vector( ) { }
    Vector(double a, double b){x= a; y= b;};
    explicit Vector(double a){x=a; y=0;}    //  no conversion !
    Vector add(const Vector& V)const;
    Vector subtr(const Vector& V)const;
    double getX()const {return x;};
    double getY()const {return y;};
  private:
    double x, y;
};


int main()
{ Vector a(1.1, 2.5),  b(3.0, 4.0),  c(7.33, 8.0);
  a= b.add(2.0);      // illegal
  a= b.add(Vector(2.0))   // OK
}
```



Destructors

An objects destructor is called when the object is destroyed
(explicitly by **delete** or implicitly when getting out of scope)

```
#include <iostream>

class A
{ public:
    A(const char* s):name(s){cout<< name << " created\n";}
    ~A(){cout << name << " deleted\n";}
                                          destructor
  private:
    const char* name;
};


int main()
{
  A* a1= new A("A1");
  A* a2= new A("A2");

    { A b1("B1"); //local variable, automatic
      A b2("B2"); // local variable, automatic

      delete a1;

    }                    b1 and b2 deleted here
delete a2;
}
```

Output:
A1 created
A2 created
B1 created
B2 created
A1 deleted
B2 deleted
B1 deleted
A2 deleted

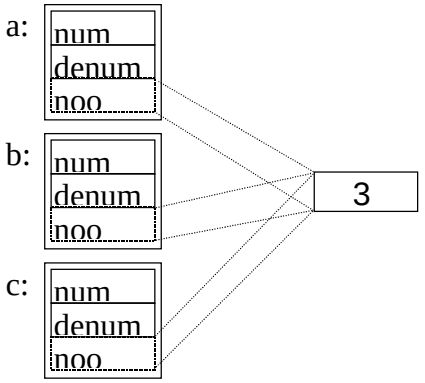
Static Members ~ Class Members

```
#include <iostream>

class F
{ public:
    F(){num=0; denum=1; noo++;}                  //default constructor
    F(int n, int d)  :num(n), denum(d){noo++;}  //overloaded constructor
    ~F(){noo--;}                                 //destructor
    void display(){cout << "(" << num << "/" << denum << ") ";}
    int nmb_of_objts(){return(noo);};
  private:
    int num, denum;
    static int noo;    // nmb of objects,  static member   <----
};

int F::noo = 0;    // file scope declaration of class F's static member

int main()
  {F a(2,3), b(3,4), c(5,6);
   cout << a.nmb_of_objts() << ", " << c.nmb_of_objts() << "\n";
    {F s(1,9), t(2,7);
     cout << a.nmb_of_objts() << ", " << c.nmb_of_objts() << "\n";
    }
   cout << a.nmb_of_objts() << ", " << c.nmb_of_objts() << "\n";
   return(0);
  }
```



Results:
3, 3
5, 5
3, 3

Static Members/Methods, Modules

```
#include <iostream>

class TableModul
{  public:
   static int lookup(int x)
     {  for(int j=0; j<LEN;++j)
          {if(Table[j].x==x)
                        return Table[j].y;
          }
        return 0;
     }

   static void change(int x,int y)
     {  for(int j=0; j<LEN;++j)
          {if(Table[j].x==x)Table[j].y=y;}
     }

private:
   enum{LEN=5};
   struct Entry{int x,y;};
   static Entry Table[LEN];
};

TableModul::Entry TableModul::Table[]=
                  {{1,2},{2,4},{3,9},{4,16},{5,25}};

int main()
{  cout<<TableModul::lookup(3)<< endl;
   TableModul::change(3,100);
   cout<<TableModul::lookup(3)<< endl;
}
```

The member functions contain no references to non static members ⇒

- functions can be declared **static**
- Static member functions has no host pointer (**this**)
- Static member functions can be invoked through an object or directly using the class scope operator
  Class-name::memfunc( )

File scope declaration and initialisation of static member of class TableModul

No object created.
Use TableModul as a module