Class and Objects

Recall

- Things having physical or logical existence are objects
- A class is a description of a number of similar objects.
- An object is said to be an instance of a class
  - Alto 800 is an instance of a car
  - There can not be an object called **Student** or **Professor**, rather there may be instances of Student and Professor.

Defining class

class ***ClassName*** {

access_specifier1:

field11_definition;

field12_definition;

…

access_specifier2:

field21_definition;

field22_definition;

…

};

Class definition

```
class FirstClass {
    private:
        int dataone;
        int datatwo;
    public:
        void setdata(int o, int t) { dataone = o; datatwo = t }
        void showdata()
            { cout << “\nData are “ << dataone << datatwo; }
};
```

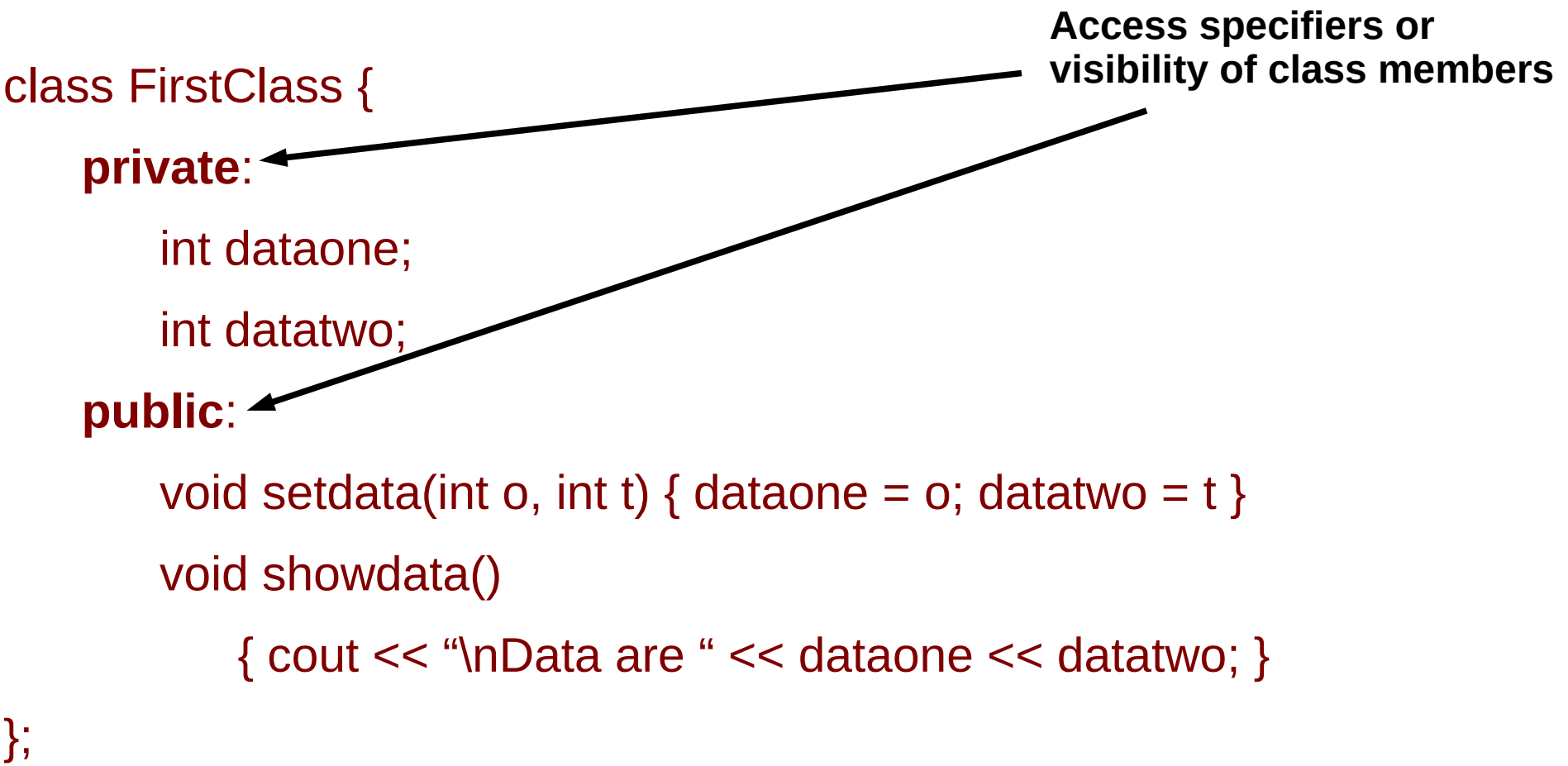
Class definition(2)

Class name

```
class FirstClass {
    private:
        int dataone;
        int datatwo;
    public:
        void setdata(int o, int t) { dataone = o; datatwo = t }
        void showdata()
            { cout << "\nData are " << dataone << datatwo; }
};
```

Definition ends with semicolon just like structure

Class definition(3)

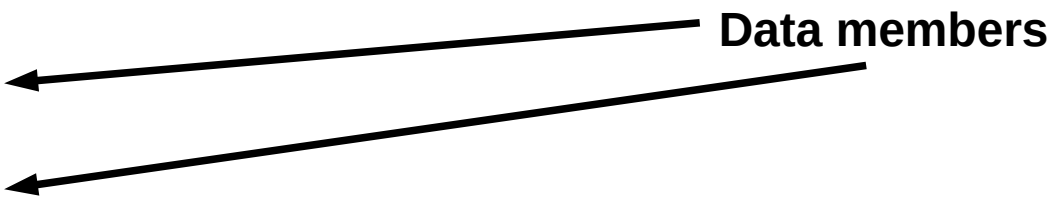


access specifier

- **public**
- **private**
- **protected**
- Fields marked as **private** can only be accessed by functions that are member of that class (there is an exception)
- In the FirstClass class, dataone, and datatwo fields are private fields
- Fields marked as **public** can be accessed by anyone
- The setdata() and showdata() are public these functions can be called by anyone

Class definition(4)

```
class FirstClass {
    private:
        int dataone;
        int datatwo;
    public:
        void setdata(int o, int t) { dataone = o; datatwo = t }
        void showdata()
            { cout << "\nData are " << dataone << datatwo; }
};
```

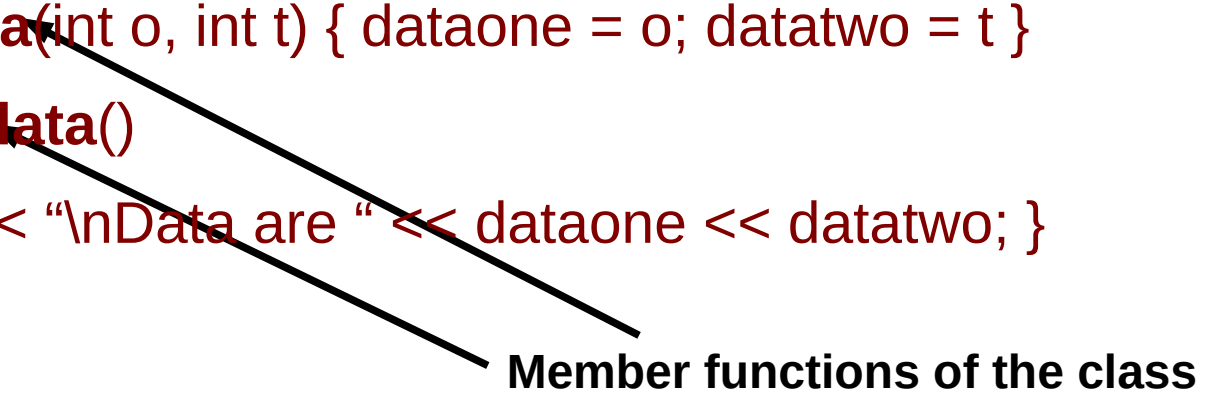


Restriction on data members

- A non-static member variable cannot have an initializer.
- No member can be an object of the class that is being declared. (a member can be a pointer to the class that is being declared.)
- No member can be declared as auto, extern, or register.
- In general, all data members of a class should be private.

Class definition(5)

```
class FirstClass {
    private:
        int dataone;
        int datatwo;
    public:
        void setdata(int o, int t) { dataone = o; datatwo = t }
        void showdata()
            { cout << "\nData are " << dataone << datatwo; }
};
```



Functions Are Public, Data Is Private

...

Why??

Answer

- Data are private to protect them accessing from any scope outside of its own class
- Member functions are public to allow invocation from a scope outside of the class
- Member fuctions are the (public) medium for performing operations on (private) data members of the class from outside.

Member functions

- Note that the member functions setdata() and showdata() are definitions contained within the class definition.
- Member functions defined inside a class this way (in a single line) are created as inline functions by default.
- It is also possible to declare a function within a class but define it elsewhere.
- Functions defined outside the class are not normally inline.

Defining objects

- **FirstClass s1, s2;**
  - Defines two objects, s1 and s2, of class **FirstClass**.
  - Instantiating object s1, s2 of class FirstClass
  - Creating objects s1, s2 of class FirstClass
- Objects participate in program operations.
- Defining an object is similar to defining a variable of any data type
- Objects are sometimes called instance variables of the class.

```
class FirstClass {
      private:
            int dataone;
            int datatwo;
      public:
            void setdata(int o, int t)
                  { dataone = o; datatwo = t }
            void showdata()  {
               cout << "\nData are "
                 << dataone << datatwo;  }
};
int main()
{
      FirstClass s1, s2;
      return 0;
}
```

Calling Member Functions

- s1.setdata(10,66);
- s2.setdata(17,76);
- Member functions can be accessed only by an object of that class.
- The dot operator (period) connects the object name and the member function.
  - Also called the class **member access operator**.
- Member function calls are also called as **messages**.
- **s1.showdata();** can be thought of sending a message to s1 telling it to show its data

```
class FirstClass {
      private:
            int dataone;
            int datatwo;
      public:
            void setdata(int o, int t)
                  { dataone = o; datatwo = t ;}
            void showdata()  {
            cout << "\nData are "
              << dataone  << datatwo; }
};
int main() {
      FirstClass s1, s2;
      s1.setdata(10,66);
      s2.setdata(17,76);
      s1.showdata();
      s2.showdata();
      return 0;
}
```

struct vs. class in C++

- by default all members are **public** in a struct
- Whereas, by default all members are **private** in a class.
- In all other respects, structures and classes are similar to each other.

Defining member functions outside the class

- Member functions that are declared inside a class need to be defined separately outside class.
- General form :
  - *return_type* class_name **: :** function_name(argument declaration) {

    function_ body;

    }
- :: indicates that the scope of the function is restricted to the class_name.
- Various classes may define members with same name. Their scope can be resolved using their membership label
- Member functions can access private data of the class.
- A member function can call another member function directly without .(period) operator.

An example

```
#include <iostream>
using namespace std;
class Box {
      double length;
      double breadth;
      double height;

   public:
      // Member functions declaration
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
      double getVolume(void);
};
```

```
void Box::setLength( double len ) {
    length = len;
}
void Box::setBreadth( double bre ) {
    breadth = bre;
}
void Box::setHeight( double hei ) {
    height = hei;
}
double Box::getVolume(void) {
    return length * breadth * height;
}
```

An example(2)

```
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   double volume = 0.0;

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);
   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);
   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;
   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

Runtime objects

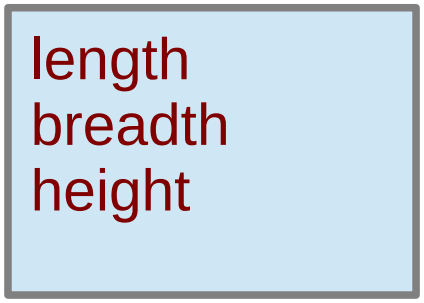
```
class Box {
        double length;
        double breadth;
        double height;
};
```

Creating objects of Box class

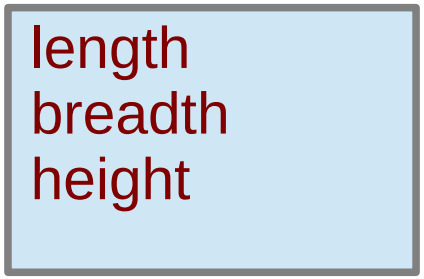
Box box1, box2, box3;

box1

box2



box3



Referring to the fields by a member function

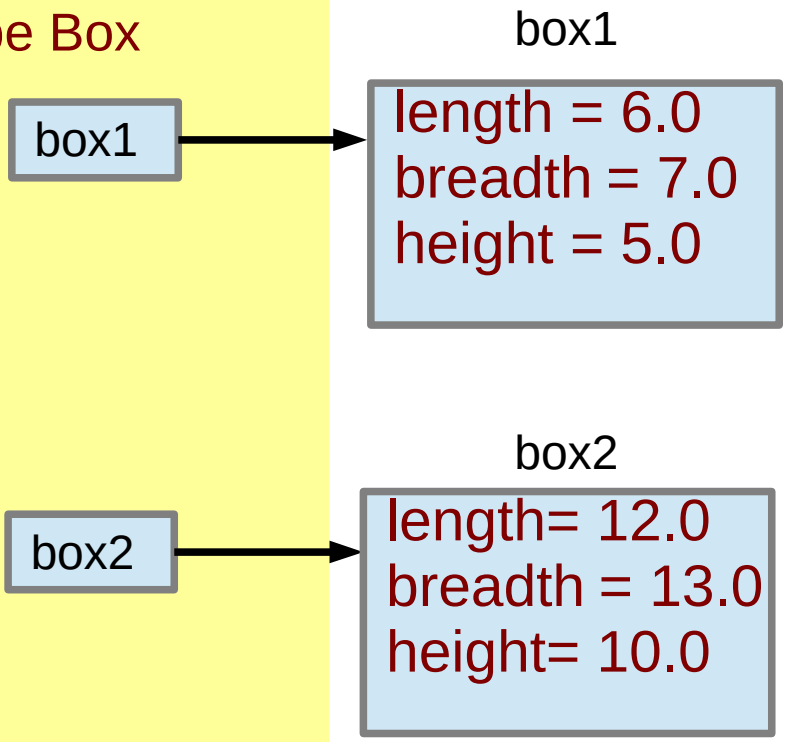
```
#include <iostream>
using namespace std;
class Box {
      double length;
      double breadth;
      double height;

   public:
      // Member functions declaration
      double getVolume(void);
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
};

double Box::getVolume(void) {
    return length * breadth * height;
}
void Box::setLength( double len ) {
    length = len;
}
void Box::setBreadth( double bre ) {
    breadth = bre;
}
void Box::setHeight( double hei ) {
    height = hei;
}
```

```
int main() {
        ...
        ...

        ...
        volume = Box1.getVolume();

        volume = Box2.getVolume();

    return 0;
}
```

- When getVolume() is called with reference to object Box1, then it refers to the member instances of Box1
- When called with reference to object Box2, these fields referes to the member copies of Box2

Accessing data members by a member function

```
int main() {
   Box Box1;              // Declare Box1 of type Box
   Box Box2;              // Declare Box2 of type Box
   double volume = 0.0;

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);
   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);
   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;
   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```



Types of class member functions

- Generally we group class methods into three broad categories:
- ***accessors*** - allow us to access the fields of a class instance (examples: getLength, getBreadth, getHeight), accessors do not change the fields of a class instance
- ***mutators*** - allow us to change the fields of a class instance (examples: setLength, setBreadth), mutators do change the fields of a class instance
- ***manager functions*** – special kind of functions (constructors, destructors) that deal with initializing and destroying class instances

Types of class member functions

- Why do we bother with accessors and mutators?
- Why provide showdata()?? Why not just make the member field publicly available?
- By restricting access using accessors and mutators, we make it possible to change the underlying details regarding a class without changing how people interact with the class
- If users interact using only accessors and mutators, then we can change things inside a class without affecting user's code

Array of objects

- It refers to array of variables that are of the type of some defined class
- Similar to other arrays, we can use index operator and dot operator to access individual elements of array objects
- Example
  - Box box[10]; //array of objects of type Box
  - cin >> box[5].width;
  - box[2].length = 44.5;
  - cout << box[1].length << box[2].length;

```
#include <iostream>
using namespace std;
class Distance {
    private:
      int feet;
      float inches;
    public:
      ... // implement getdist(), and setdist() member
function
      void add_dist( Distance, Distance );
};

void Distance::add_dist(Distance d2, Distance d3) {
      inches = d2.inches + d3.inches;
      feet = 0;
      if(inches >= 12.0) {
            inches -= 12.0;
            feet++;
      }
      feet += d2.feet + d3.feet;
}
```

```
int main() {
    Distance dist1, dist2, dist3;
    ...
    ...
    dist1.getdist();
    dist2.getdist();
    dist3.add_dist(dist1, dist2);
    dist3.getdist();
    ...
    return 0;
}
```

Objects as function arguments

Manager functions

- Sometimes, however, it is convenient if an object can **initialize itself** when it's first created, **without requiring** a separate call to a member function.
- It would also be better if some **predefined tasks** that we need to perform every time we create an object are **done automatically** when **an object is created**.
- How about triggering some predefined task when an object is destroyed automatically?
- The solution is provided by C++ through manager functions such as **constructors** and **destructors**

constructors

- Automatic initialization is carried out using a special member function called a **constructor**.
- A constructor is a member function that is executed automatically (or called automatically) whenever an object is created.
- Therefore, it substitutes the task of defining member function for initialization and explicitly calling them.
- The term constructor is sometimes abbreviated as **ctor**

Constructor features

- Constructor bears the **same name** as of class
- Constructors with no arguments is known as the **default** constructor
- Constructor can’t have a return type
- If no constructor is provided by the programmer then compiler provides one default constructor
- They cannot be inherited, though a derived class can call the base class constructor
- Cannot be made virtual
- It should always be declared in public part of the class structure. (why??)

Destructors

- It is a special member function named same as the class and preceded by a ~ (tilde)
- Example:

```
class Time{
    … public:
      ~Time();
}
```

- if your class allocates space on the heap, it is useful to deallocate that space before the object is destroyed
- It is used to free the state of an object
- These are called implicitly when an automatic object goes out of scope. But if an object is initialized using new than it has to be deleted
- In very unusual situations does the user need to call a destructor explicitly!

Types of constructor

- Constructors can be overloaded i.e. there may be more than one constructors for the same class with several ways of initialization
- Types
  - Default constructor
  - Parameterized constuctor
  - Copy constructor

Default constructor

```
// object represents a counter variable
#include <iostream>
using namespace std;

class Counter {
    private:
      unsigned int count;
    public:
      Counter() : count(0) {  }
      void inc_count() { count++; }
      void dec_count() { count--; }
      int get_count() { return count; }
};
```

```
int main() {
      Counter c1, c2;
      cout << "\nc1=" << c1.get_count();
      cout << "\nc2=" << c2.get_count();

      c1.inc_count(); //increment c1
      c2.inc_count(); //increment c2
      c2.inc_count(); //increment c2

      cout << "\nc1=" << c1.get_count();
      cout << "\nc2=" << c2.get_count();
      cout << endl;
      return 0;
}
```

```
//constructors, adds objects using member function
#include <iostream>
using namespace std;
class Distance {
    private:
       int feet;
       float inches;
    Public:
       Distance() : feet(0), inches(0.0) { }
       Distance(int ft, float in) : feet(ft), inches(in) { }
       void getdist() {
              cout << “\nEnter feet: “; cin >> feet;
              cout << “Enter inches: “; cin >> inches;
       }
       void showdist() { cout << feet << “\’-” << inches << ‘\”’; }
              void add_dist( Distance, Distance );
};

void Distance::add_dist(Distance d2, Distance d3) {
       inches = d2.inches + d3.inches;
       feet = 0;
       if(inches >= 12.0) {
              inches -= 12.0;
              feet++;
       }
       feet += d2.feet + d3.feet;
}
```

```
int main() {
       Distance dist1, dist3;
       Distance dist2(11, 6.25);
       dist1.getdist();
       dist3.add_dist(dist1, dist2);
       cout << "\ndist1 = ";
       dist1.showdist();
       cout << "\ndist2 = ";
       dist2.showdist();
       cout << "\ndist3 = ";
       dist3.showdist();
       cout << endl;
       return 0;
}
```

Parameterized constructor

Parameterized constructor : Special case

- If a constructor only has one parameter, there is a third way to pass an initial value to that constructor.

```
#include <iostream>
using namespace std;
class X {
      int a;
   public:
      X(int j) { a = j; }
      int geta() { return a; }
};
int main(){
      X ob = 99; // passes 99 to j
      X ob(99);
      cout << ob.geta(); // outputs 99
      return 0;
}
```

A combination constructor

- Can combine a ctor that requires arguments with the default ctor using default values:
- class Box {

```
        ...
    public:
            Box(double w = 0.0, double h = 0.0, double d = 0.0) {
                    width = w; height = h; depth = d;
            }
};
```

calling:

```
Box box1;   // ctor called with default args
Box box2();; // ctor called with default args
Box box3(5.0); // ???
Box box4(10.0,10.0,10.0);
```

Example: Destructor

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

class Distance{

    int feet;
    float inches;
    public:
    Distance(void){ cout << "Object created" <<endl; }
    ~Distance(void){ cout << "Object destroyed" << endl; }
    void getdist() {
        cout << "\nEnter feet: "; cin >> feet;
        cout << "\nEnter inches: "; cin >> inches;
    }
    void showdist() { cout << feet << "\'-" << inches << '\"'<<endl; }
};

int main(){
    Distance *ob1= new Distance();
    ob1->getdist();
    ob1->showdist();
    delete ob1;
    Distance *ob2= new Distance();
    ob2->getdist();
    ob2->showdist();
    delete ob2;
    return 0;
}
```

Array of objects initialization

```
#include <iostream>
using namespace std;
class Number {
        int i;
    public:
        Number(int j) { i=j; }
        int get_i() { return i; }
};
int main() {
    Number ob[3] = {1, 2, 3}; //short version of initialization
    Number ob[3] = { Number(1), Number(2), Number(3) };
    //longer form of initialization
...
}
```

Array of objects initialization

- If an object's constructor requires two or more arguments, you will have to use the longer initialization form
- Example :

```
class Coordinate2d {
        int x;
        int y;
    public:
    Coordinate2d(int j, int k) { x=j; y=k; } // constructor with 2 parameters
    ...
};
int main() {
    Coordinate2d  ob[3] = { Coordinate2d (1, 2), Coordinate2d (3, 4),
    Coordinate2d (5, 6) };
    ...
}
```

What is the problem with this?

```
#include <iostream>
using namespace std;
class Number {
        int i;
    public:
        Number(int j) { i=j; }
        int get_i() { return i; }
};
int main() {
    Number ob[3];
...
}
```

Pointers to objects

```
#include <iostream>
using namespace std;
class Number {
        int i;
    public:
        Number(int j) { i=j; }
        int get_i() { return i; }
};
int main() {
    Number ob(88), *p;
    p = &ob; // get address of ob
    cout << p->get_i();
    ...
    ...
    Number ob2[3] = {1, 2, 3};
    p = ob2; // get start of array
    for(i=0; i < 3; i++) {
        cout << p->get_i() << "\n";
        p++;
    }
return 0;
}
```

this pointer

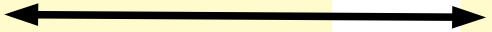
- When a non-static member function is called, automatically a pointer to the invoking object is passed as an implicit argument.
- This pointer is called **this**
- Every object has access to its own address through **this** pointer
- The **this** pointer is implicitly used to refer both the data and function members of an object
- It can also be used explicitly;
  - Example: `(*this).x=5;` or `this->x=5;`

'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).

For a class X, the type of this pointer is 'X* '. Also, if a member function of X is declared as const, then the type of this pointer is 'const X *'

Example : this pointer

```
#include <iostream>
using namespace std;
class pwr {
      double b;
      int e;
      double val;
    public:
      pwr(double base, int exp);
      double get_pwr() { return val; }
};
pwr::pwr(double base, int exp) {
      b = base;
      e = exp;
      val = 1;
      for( ; exp>0; exp--) val = val * b;
}
int main(){
      pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
      cout << x.get_pwr() << " ";
      cout << y.get_pwr() << " ";
      cout << z.get_pwr() << "\n";
      return 0;
}
```



```
pwr::pwr(double base, int exp) {
      this->b = base;
      this->e = exp;
      this->val = 1;
      for( ; exp>0; exp--)
            this->val = this->val * this->b;
}
```

Pointers to class members

```
#include <iostream>
using namespace std;
class Number {
    public:
       Number (int i) { val=i; }
       int val;
       int double_val() { return val+val; }
};
int main() {
       int Number::*data; // data member pointer
       int (Number::*func)(); // function member pointer
       Number ob1(1), ob2(2);
       data = &Number::val; // get offset of val
       func = &Number::double_val; // get offset of double_val()
       cout << "Original values: ";
       cout << ob1.*data << " " << ob2.*data << "\n";
       cout << "Doubled values: ";
       cout << (ob1.*func)() << " ";
       cout << (ob2.*func)() << "\n";
       return 0;
}
```

Copy constructor

- One of the most important forms of an overloaded constructor
- A copy constructor can help to prevent problems occurs when one object is used to initialize another.
- By default, when one object is used to initialize another, C++ performs a bitwise copy.
- That is, an identical copy of the initializing object is created in the target object.
- A common case is when an object allocates memory dynamically when it is created.

Copy constructor

- For example,
  - assume a class called MyClass that allocates memory dynamically for each object when it is created, and an object A of that class.
  - If a bitwise copy is performed, then B will be an exact copy of A.
  - This means that B will be using the same piece of allocated memory that A is using, instead of allocating its own.
  - Clearly, this is not the desired outcome.
  - if MyClass includes a destructor that frees the memory, then the same piece of memory will be freed twice when A and B are destroyed!

Copy constructor

- When a copy constructor exists, the default copy constructor (bitwise copy) is bypassed.
- The most common general form of a copy constructor is

```
classname (const classname &o) {
    // body of constructor
}
```

- It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them.
- C++ defines three distinct types of initialization in which the value of one object is given to another.
  - When one object explicitly initializes another, such as in a declaration (myclass x = y;)
  - When a copy of an object is made to be passed to a function (func(y))
  - When a temporary object is generated; most commonly, as a return value. (y = func();)
- The copy constructor applies to these initializations.

Example: Copy constructor

```
class Distance {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0) { }
      Distance(int ft, float in) : feet(ft), inches(in) { }
      Distance(const Distance& ob) {
            feet = ob.feet; inches = ob.inches;
      }
      void showdist() {
            cout << feet << "\'-" << inches << '\"';
      }
};
int main() {
      Distance dist1(11, 6.25);
      Distance dist2(dist1);
      Distance dist3 = dist1;
      cout << " \n dist1 = "; dist1.showdist();
      cout << " \n dist2 = "; dist2.showdist();
      cout << " \n dist3 = "; dist3.showdist();
      cout << endl;
      return 0;
}
```

Example : Copy constructor

```
class Table{
    char *name;
    float size;
    ...
    Table(float s=15){
          name=new char[size=s];
     }
    Table (const Table&);
};
Table::Table(const Table& t){
       name=new char[size=t.size];
       strcpy(name,t.name);
}
```

```
int main(){
     Table t1;
     Table t2=t1;
     Table t3;
     t3=t2; //???
     ...
     ...
}
```

- Why copy constructor argument is passed as const reference??

Answer

- Reference argument to the copy constuctor avoids recursive call to it, since when an object is passed by value the same copy constructor is invoked to create a copied object.
- Foremost reason of const specification is to avoid accidental modification of the object. Moreover, we can not have non-const reference to any temporary objects.

const objects

- Objects can be made const by using the const keyword.
- All const variables must be initialized at time of creation using constructors
- Once a const class object has been initialized via constructor, any attempt to modify the member variables of the object is disallowed
- Example
  - const Distance obj3;

const member function

- const class objects can only call const member functions
- A **const member function** is a member function that guarantees it will not change any class variables or call any non-const member functions.
- To make a function a const member function, we simply append the const keyword to the function prototype
- Any const member function that attempts to change a member variable or call a non-const member function will cause a compiler error to occur

```
class Distance {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0) { }
      Distance(int ft, float in) : feet(ft), inches(in) { }
      ...
      void showdist() const {
            cout << feet << "\'-" << inches << '\"';
      }
};
```

Static member of a class

- Static data member is an attribute that is a part of class, yet is not a part of an object
- In otherwords, there is exactly one copy of a static member instead of one copy per object
- A Function that needs access to members of a class, yet does not need to be invoked for a particular object is called static member function
- Often used when declaring class constants (since you generally only need one copy of a constant)
- To make a field static, add the **`static`** keyword in front of the field
  - can refer to the field like any other field
  - static variables are also considered to be global, you can refer to them without an instance static fields can be initialized

Static class members

- Static data members are accessible by both static and/or non-static member functions
- Static member functions can only access the static class members
- To access a public static class member, simply prefix the class name and scope resolution operator
- To access a private static class member, when no object exists take help of a static member function otherwise non-static could be used.

Example

```
class Employee{
        char* Name;
        static int count;
    public:
        char* getName();
        static int getCount();
        static void incCount();
};
int Employee::count=0;
int Employee::getCount() {return count;}
void Employee::incCount() { count++; }
char* Employee::getName() {    return Name; }

main(){
      cout<<“no of Employees: “<< Employee::getCount()<<endl;
      Employee* e1=new Employee (“Bob”);
      e1->incCount();
      Employee* e2=new Employee (“John”);
      e2->incCount();
      cout<<“no of employees: “ <<Employee::getCount();
}
```

A complete example

```
class Employee{
      char* Name;
      static int count;
      public:
      Employee(char*);
      char* getName();
      static int getCount();
      ~Employee();
};

int Employee::count=0;
int Employee::getCount()
        {return Count;}
Employee::Employee(char* N){
        Name=new char[strlen(N)+1];
        strcpy(Name,N);
        ++count;
}
Employee::~Employee(){
                    delete [] Name;
                    --count;
}
char* Employee::getName(){
        return Name;
}
```

```
main(){
        cout<<"no of Employees: "<<
             Employee::getCount()<<endl;

    Employee* e1= new Employee("Bob");
      Employee* e2=new Employee ("John");

      cout<<"no of employees: " <<
            e1->getCount();
     cout<<"Emp1: "<<e1->getName();
     cout<<"Emp2: "<<e2->getName();
     delete e1;
     cout<<"no of employees: "
            <<e2->getCount();
     delete e2;
     cout<<"no of employees: " <<
     Employee::getCount();
}
```

Friend function

- There could be a situation where we would like two classes to share a particular fuction
- Example
  - findarea() can be shared by rectangle and triangle class
  - income_tax() function by manager and clerks class
- C++ allows the common function to be made friendly with both the classes.
- The friendly function is allowed to access to the private data of these classes

Friend function

- To make an outside function friend of a class, the declaration of the function included in the class with **friend** keyword

```
class XYZ {

      ...
      public:
            friend void frndfun();

      ...
};
```

Features of a friend function

- It is not in the scope of the class to which it has been declared as friend
- It can not be called using the object reference
- It access the members of an object using dot operator (like A.x)
- Usually has objects as argument
- Often used in operator overloading

Member function as friend

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

class Point2D;
class Circle{
      int centerX;
      int centerY;
      int rad;
      public:
      void show(){
            cout << "Center : "
      << centerX << centerY << endl;
            cout << "Radius : "
      << rad << endl;
      }
      void setValues(Point2D,int);
};
```

```
class Point2D{

      int xco;
      int yco;
      public:
      int getX(void) { return xco; }
      int getY(void) { return xco; }
      void setPoint(int x,int y)
            { xco =x; yco =y; }
      friend void Circle::setValues(Point2D,int);
};

void Circle::setValues(Point2D p, int r){
            centerX = p.xco;
            centerY = p.yco;
            rad = r;
}
```

An example of friend function

```
// friend functions
#include <iostream>
using namespace std;
class Rectangle {
      int width, height;
  public:
      void set_values (int, int);
      int area () {
            return (width * height);
      }
      friend Rectangle duplicate (Rectangle);
};
void Rectangle::set_values (int a, int b)
{ width = a; height = b; }
```

```
Rectangle duplicate (Rectangle rectparam) {
      Rectangle rectres;
      rectres.width = rectparam.width*2;
      rectres.height = rectparam.height*2;
      return (rectres);
 }

 int main() {
      Rectangle rect, rectb;
      rect.set_values (2,3);
      rectb = duplicate (rect);
      cout << rectb.area();
      return 0;
 }
```

Friend class

- We can also declare all the member functions of one class as the friend function of another class.
- Here entire class is a friend class.

```
class X {
        ...
        friend class Y;
        ...
};
```

An Example of Friend class

```
// friend class
#include <iostream>
using namespace std;
class Square;
class Rectangle {
  int width, height;
 public:
  int area() {
   return (width * height);
  }
  void convert (Square a);
 };
 class Square {
  private:
    int side;
  public:
   void set_side (int a){
    side=a;
  }
  friend class Rectangle;
 };
```

```
void Rectangle::convert(Square a){
      width = a.side;
      height = a.side;
}
int main(){
      Square sqr;
      Rectangle rect;
      sqr.set_side(4);
      rect.convert(sqr);
      cout << rect.area();
      return 0;
}
```

End of Class and Object Slides