Object Oriented Programming in C++

Destructors

Based on materials from Brian Balazs (USNA)

Destructors

- Only one per class
- Class name with ~ in front
- Doesn't take any arguments
- Controls what happens when object destroyed
- Called automatically

Destructor Example

```
class Silly
{
  private:
    string name;
  public:
    Silly(){
      cout <<"A silly object is born!"<< endl;
    }
    ~Silly(){
      cout <<"Silly object "<<name<<" dies!"<< endl;
    }
};
```

```
int main()
{
  Silly *p;
  if (1>0){
    Silly first;
    first.name = “Tom";
    p = new Silly[2];
    p[0].name = "John";
    p[1].name = “Sara";
  }
  Silly last;
  last.name = “Tom Jr";
  delete [] p;
  return 0;
}
```

Use?

- When is destructor useful?
  - Executed when object destroyed
  - Can do anything, but interesting when deallocate memory
    - Want to delete items created using new to free up memory

Destructor Example

```
/** DEFINITION OF CLASS NODE **/
class Node {
  public:
    int data;
    Node *next;
    Node(int val, Node* p) {
      data = val;
      next = p;
    }
};
```

```
List::~List() {
  while(head != 0) {
    Node *p = head;
    head = head->next;
    delete p;
  }
}
```

Templates

Based on materials by Bjarne Stroustrup

Templates

- But we don't just want vector of double
- We want vectors with element types we specify
  - **vector<double>**
  - **vector<int>**
  - **vector<Month>**
  - **vector<Record*>** *// vector of pointers*
  - **vector< vector<Record> >** *// vector of vectors*
  - **vector<char>**
- We must make the element type a parameter to **vector**
- **vector** must be able to take both built-in types and user-defined types as element types
- This is not some magic reserved for the compiler, we can define our own parameterized types, called "templates"

Templates

- The basis for generic programming in C++
  - Sometimes called “parametric polymorphism”
    - Parameterization of types (and functions) by types (and integers)
  - Unsurpassed flexibility and performance
    - Used where performance is essential (*e.g.,* hard real time and numerics)
    - Used where flexibility is essential (*e.g.,* the C++ standard library)

- Template definitions

```
template<class T, int N> class Buffer { /* ... */ };
template<class T, int N> void fill(Buffer<T,N>& b) { /* ... */  }
```

- Template specializations (instantiations)

```
// for a class template, you specify the template arguments:
Buffer<char,1024> buf;   // for buf, T is char and N is 1024

// for a function template, the compiler deduces the template arguments:
fill(buf);        // for fill(), T is char and N is 1024; that's what buf has
```

Parameterize with element type

```
// an almost real vector of Ts:
template<class T> class vector {
    // …
};

vector<double> vd;              // T is double
vector<int> vi;                 // T is int
vector< vector<int> > vvi;      // T is vector<int>
                    //          in which T is int
vector<char> vc;                // T is char
vector<double*> vpd;            // T is double*
vector< vector<double>* > vvpd; // T is vector<double>*
                    //          in which T is double
```

vector<T> is

```
// an almost real vector of Ts:
template<class T> class vector {   // read "for all types T" (just like in math)
    int sz;             // the size
    T* elem;            // a pointer to the elements
    int space;          // size+free_space
public:
    vector() : sz(0), elem(0), space(0);          // default constructor
    explicit vector(int s) :sz(s), elem(new T[s]), space(s) { }   // constructor
    vector(const vector&);                        // copy constructor
    vector& operator=(const vector&);             // copy assignment
    ~vector() { delete[ ] elem; }                 // destructor

    T& operator[ ] (int n) { return elem[n]; }    // access: return reference
    int size() const { return sz; }               // the current size

    // …
};
```

Basically, **vector<double>** is

```
// an almost real vector of doubles:
class vector {
    int sz;           // the size
    double* elem;     // a pointer to the elements
    int space;        // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }            // default constructor
    explicit vector(int s) :sz(s), elem(new double[s]), space(s) { } // constructor
    vector(const vector&);                        // copy constructor
    vector& operator=(const vector&);             // copy assignment
    ~vector() { delete[ ] elem; }                 // destructor

    double& operator[ ] (int n) { return elem[n]; }  // access: return reference
    int size() const { return sz; }               // the current size

    // …
};
```

Basically, **vector<char>** is

```
// an almost real vector of chars:
class vector {
    int sz;              // the size
    char* elem;          // a pointer to the elements
    int space;           // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }              // default constructor
    explicit vector(int s) :sz(s), elem(new char[s]), space(s) { } // constructor
    vector(const vector&);                     // copy constructor
    vector& operator=(const vector&);          // copy assignment
    ~vector() { delete[ ] elem; }              // destructor

    char& operator[ ] (int n) { return elem[n]; }   // access: return reference
    int size() const { return sz; }            // the current size

    // …
};
```

Templates

- Problems (“there’ s no free lunch”)
  - Poor error diagnostics
    - Often spectacularly poor
  - Delayed error messages
    - Often at link time
  - All templates must be fully defined in each translation unit
    - (the facility for separate compilation of templates, called “export”, is not widely available)
    - So place template definitions in header files
- Recommendation
  - Use template-based libraries
    - Such as the C++ standard library
      - *E.g.,* **vector**, **sort()**
  - Initially, write only very simple templates yourself
    - Until you get more experience

Range checking

```
// an almost real vector of Ts:

struct out_of_range { /* … */ };

template<class T> class vector {
    // …
    T& operator[ ](int n);       // access
    // …
};

template<class T>  T& vector<T>::operator[ ](int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}
```

Range checking

```
void fill_vec(vector<int>& v, int n)     // initialize v with factorials
{
    for (int i=0; i<n; ++i) v.push_back(factorial(i));
}

int main()
{
    vector<int> v;
    try {
      fill_vec(v,10);
      for (int i=0; i<=v.size(); ++i)
            cout << "v[" << i << "]==" << v[i] << '\n';
    }
    catch (out_of_range) {             // we'll get here (why?)
      cout << "out of range error";
      return 1;
    }
}
```

Exception handling (primitive)

```
// sometimes we cannot do a complete cleanup

vector<int>* some_function()      // make a filled vector
{
    vector<int>* p = new vector<int>; // we allocate on free store,
                                      // someone must deallocate
    try {
      fill_vec(*p,10);
      // …
      return p;     // all's well; return the filled vector
    }
    catch (…) {
      delete p;     // do our local cleanup
      throw;        // re-throw to allow our caller to deal
    }
}
```

Exception handling
(simpler and more structured)

```
// When we use scoped variables cleanup is automatic

vector<int> glob;

void some_other_function()   // make a filled vector
{
    vector<int> v;     // note: vector handles the deallocation of elements

    fill_vec(v,10);
    // use v
    fill_vec(glob,10);
    // …
}
```

- if you feel that you need a try-block: think.
  - You might be able to do without it

RAII (Resource Acquisition Is Initialization)

- Vector
  - acquires memory for elements in its constructor
  - Manage it (changing size, controlling access, etc.)
  - Gives back (releases) the memory in the destructor
- This is a special case of the general resource management strategy called RAII
  - Also called "scoped resource management"
  - Use it wherever you can
  - It is simpler and cheaper than anything else
  - It interacts beautifully with error handling using exceptions
  - Examples of "resources":
    - Memory, file handles, sockets, I/O connections (iostreams handle those using RAII), locks, widgets, threads.

What the standard guarantees

```
// the standard library vector doesn't guarantee a range check in operator[ ]:
template<class T> class vector {
    // …
    T& at(int n);              // checked access
    T& operator[ ](int n);     // unchecked access
};

template<class T>  T& vector<T>::at (int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}

template<class T>  T& vector<T>::operator[ ](int n)
{
    return elem[n];
}
```

What the standard guarantees

- Why doesn't the standard guarantee checking?
  - Checking cost in speed and code size
    - Not much; don't worry
      - No student project needs to worry
      - Few real-world projects need to worry
  - Some projects need optimal performance
    - Think huge (e.g., Google) and tiny (e.g., cell phone)
  - The standard must serve everybody
    - You can build checked on top of optimal
    - You can't build optimal on top of checked
  - Some projects are not allowed to use exceptions
    - Old projects with pre-exception parts
    - High reliability, hard-real-time code (think airplanes)

C++ Inheritance

One Class “inherits”
Properties of Another

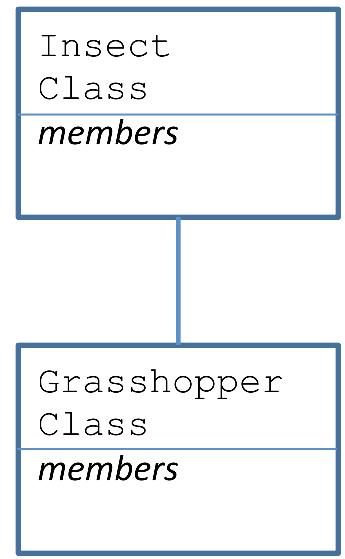
Tony Gaddis, Starting out with C++
Herbert Schildt, Teach Yourself C++

C++ Inheritance

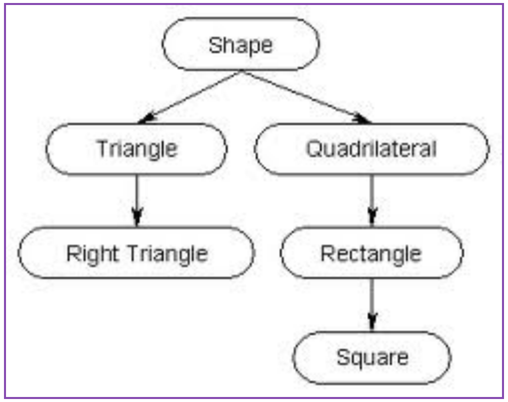
- Inheritance allows a hierarchy of classes to be built.
- Move from the most general
  to the most specific
- The class that is inherited is the **base class**.
- The inheriting class is called the **derived class**.
- A derived class inherits traits of the base class
  &
  adds properties that are specific to that class.

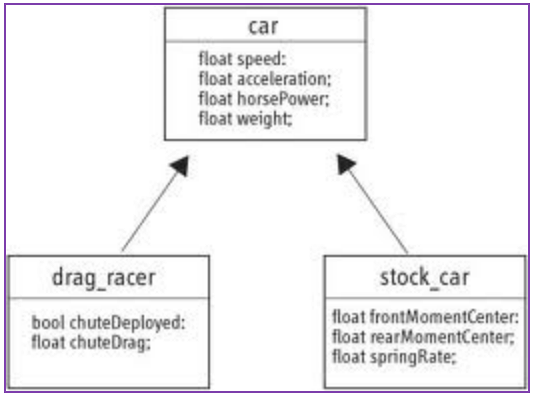
C++ Inheritance

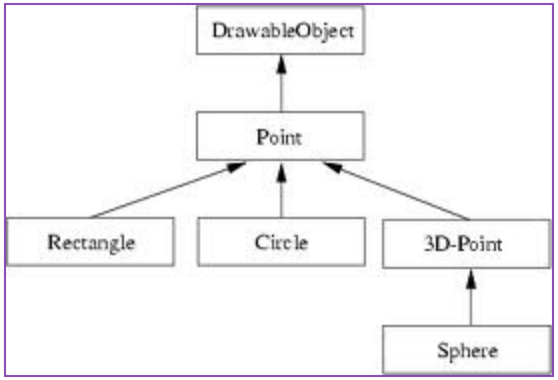
- Inheritance = the "**Is a**" Relationship
- A poodle **is a** dog
- A car **is a** vehicle
- A tree **is a** plant
- A rectangle **is a** shape
- A football player **is a** an athlete
- Base Class is the ***General Class***
- Derived Class is the ***Specialized Class***

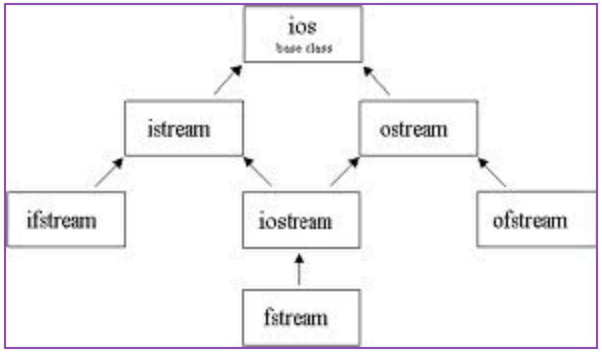


C++ Inheritance









C++ Inheritance

- **Syntax**

```
class B {
   int I;
public:
   void Set_I(int X){I=X;}
   int  Get_I() {return I;}
};
```

Access Specification: Public

- Public members of Base are public members of Derived
- Private members of Base remain private members, but are inherited by the Derived class.

i.e. "They are invisible to the Derived class"

Base Class Access Specification

```
class D : public B {
   int J;
public:
   void Set_J(int X)
          {J = X;}
   int  Mul()
          {return J * Get_I();}
          // J * I → Compile error!
};
```

B Class //Base

members

D Class //Derived

members

```
int main() {
   D ob;
   ob.Set_J(10);
   ob.Set_I(4);
   // ob.I = 8; Compile error!
   cout << ob.Mul() << endl;
   return 0;
} // end main
```

C++ Inheritance

- A base class in not exclusively “owned” by a derived class. A base class can be inherited by any number of different classes.

- There may be times when you want to keep a member of a base class private but still permit a derived class access to it.
  SOLUTION: Designate the data as **protected.**

C++ Inheritance

- **Protected Data Inherited as Public**

Private members of the base class are always private to the derived class regardless of the access specifier.

```
class Base {
  protected:
      int a, b;
  public:
      void Setab(int n, int m)
           { a = n; b = m;}
};
```

```
class Derived: public Base {
     int c;
  public:
     void Setc(int x) { c = x;}
     void Showabc() {
        cout << a << " " << b << " " << c << endl;
      }
};
```

```
int main() {
   Derived ob;

   ob.Setab(1,2);
   ob.Setc(3);
   ob.Showabc();
   //ob.a = 5 NO! NO!

   return 0;

} // end main
```

C++ Inheritance

- **Public Access Specifier**
  - *Private members of Base remain private members and are inaccessible to the derived class.*
  - *Public members of Base are public members of Derived*

BUT

  - ***Protected members of a base class are accessible to members of any class derived from that base.***

    Protected members, like private members, are not accessible outside the base or derived classes.

Private members of the base class are always private to the derived class regardless of the access specifier

C++ Inheritance

- But when a base class is inherited as **protected**, ***public and protected*** members of the base class become protected members of the derived class.

```
class Base {
  protected:
     int a, b;
  public:
      void Setab(int n, int m)
           { a = n; b = m;}
};
```

```
class Derived: protected Base {
     int c;
  public:
     void Setc(int x) { c = x;}
     void Showabc() {
        cout << a << " " << b << " " << c << endl;
      }
};
```

```
int main() {
   Derived ob;

   //ob.Setab(1,2); ERROR
   //ob.a = 5;     NO! NO!

   ob.Setc(3);
   ob.Showabc();

   return 0;
} // end main
```

Private members of the base class are always private to the derived class regardless of the access specifier

C++ Inheritance

- **Protected Access Specifier**
  - Private members of the base class are inaccessible to the derived class.
  - Public members of the base class become protected members of the derived class.
  - Protected members of the base class become protected members of the derived class.

i.e. only the public members of the derived class are accessible by the user application.

C++ Inheritance

- **Constructors & Destructors**
  - When a base class and a derived class both have constructor and destructor functions
    - Constructor functions are executed in order of derivation – base class before derived class.
    - Destructor functions are executed in reverse order – the derived class's destructor is executed before the base class's destructor.
  - A derived class does not inherit the constructors of its base class.

C++ Inheritance

```
class Base {
   public:
      Base() { cout << "Constructor Base Class\n";}
      ~Base() {cout << "Destructing Base Class\n";}
};
class Derived : public Base {
   public:
      Derived() { cout << Constructor Derived Class\n";}
      ~Derived(){ cout << Destructing Derived Class\n";}
};
```

```
int main() {
   Derived ob;
   return o;
}
```

```
---- OUTPUT ----
Constructor Base Class
Constructor Derived Class
Destructing Derived Class
Destructing Base Class
```

C++ Inheritance

- **Passing an argument to a derived class's constructor**

```
Class Base {
   public:
      Base()  {cout << "Constructor Base Class\n";}
      ~Base(){cout << "Destructing Base Class\n";}
};
Class Derived : public Base {
      int J;
   public:
      Derived(int X) {
          cout << Constructor Derived Class\n";
          J = X;
      }
      ~Derived(){ cout << Destructing Derived Class\n";}
      void ShowJ() { cout << "J: " << J << "\n"; }
};
```

```
int main() {
   Derived Ob(10);
   Ob.ShowJ();
   return 0;
} // end main
```

C++ Inheritance

- **Arguments to both Derived and Base Constructors**

```
Class Base {
   int I;
   public:
      Base(int Y) {
         cout << "Constructor Base Class\n";
         I = Y;}
      ~Base(){cout << "Destructing Base Class\n";}
      void ShowI() { cout << "I: " << I << endl; }
};
Class Derived : public Base {
      int J;
   public:
      Derived(int X) : Base (X) {
         cout << Constructor Derived Class\n";
         J = X;
      }
      ~Derived(){ cout << Destructing Derived Class\n";}
      void ShowJ() { cout << << "J:" << J << "\n"; }
};
```

```
int main() {
   Derived Ob(10);

   Ob.ShowI();
   Ob.ShowJ();
   return 0;
} // end main
```

C++ Inheritance

- **Different arguments to the Base – All arguments to the Derived.**

```
Class Base {
   int I;
   public:
      Base(int Y) {
         cout << "Constructor Base Class\n";
         I = Y;}
      ~Base(){cout << "Destructing Base Class\n";}
      void ShowI() { cout << "I: " << I << endl; }
};
Class Derived : public Base {
      int J;
   public:
      Derived(int X, int Y) : Base (Y) {
         cout << Constructor Derived Class\n";
         J = X;
      }
      ~Derived(){ cout << Destructing Derived Class\n";}
      void ShowJ() { cout << << "J:" << J << "\n"; }
};
```

```
int main() {
   Derived Ob(5,8);

   Ob.ShowI();
   Ob.ShowJ();
   return 0;
} // end main
```

C++ Inheritance

- **OK – If Only Base has Argument**

```
Class Base {
   int I;
   public:
      Base(int Y)  {
         cout << "Constructor Base Class\n";
         I = Y;}
      ~Base(){cout << "Destructing Base Class\n";}
      void ShowI() { cout << "I: " << I << endl; }
};
Class Derived : public Base {
      int J;
   public:
      Derived(int X) : Base (X)  {
         cout << Constructor Derived Class\n";
         J = 0;            // X not used here
      }
      ~Derived(){ cout << Destructing Derived Class\n";}
      void ShowJ() { cout << << "J:" << J << "\n"; }
};
```

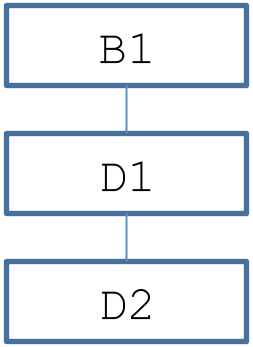
```
int main()  {
   Derived Ob(10);

   Ob.ShowI();
   Ob.ShowJ();
   return 0;
} // end main
```

C++ Inheritance

- **Multiple Inheritance – Inheriting more than one base class**

1. Derived class can be used as a base class for another derived class (***multilevel class hierarchy***)
2. A derived class can directly inherit more than one base class. 2 or more base classes are combined to help create the derived class

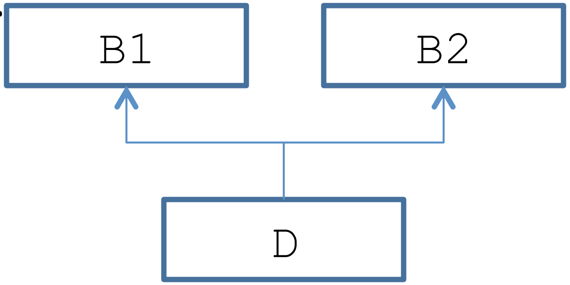
C++ Inheritance



- **Multiple Inheritance**

1. Multilevel Class Hierarchy
   - Constructor functions of all classes are called in order of derivation: B1, D1, D2
   - Destructor functions are called in reverse order

2. When a derived class directly inherits multiple base classes…
   - Access_Specifiers { public, private, protected} can be different
   - Constructors are executed in the order left to right, that the base classes are specified.
   - Destructors are executed in the opposite order.

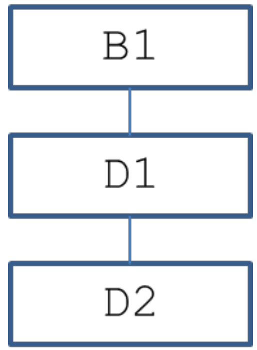



```
class Derived_Class_Name: access Base1,
           access Base2,… access BaseN
{
   //.. body of class
} end Derived_Class_Name
```

C++ Inheritance

- **Derived class inherits a class derived from another class.**

```
class B1 {
   int A;
   public:
      B1(int Z) { A = Z;}
      int GetA() { return A; }
};
class D1 : public B1 {
      int B;
   public:
      D1(int Y, int Z) : B1 (Z) { B = Y; }
      void GetB() { return B; }
};
class D2 : public D1 {
      int C;
    public:
      D2 (int X, int Y, int Z) : D1 ( Y, Z)) { C = X; }
      void ShowAll () {
         cout << GetA() << " " << GetB() << " " << C << endl; }
};
```



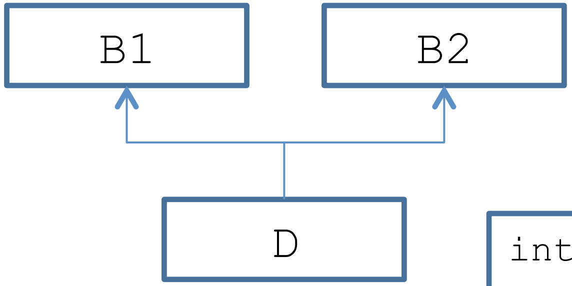

```
int main() {
   D2 Ob(5,7,9);

   Ob.ShowAll();

   // GetA & GetB are still public here
   cout << Ob.GetA() << " "
        << Ob.GetB() << endl;

   return 0;
} // end main
```

Because bases are inherited as public,
D2 has access to public elements of both B1 and D1

C++ Inheritance

Derived Class Inherits Two Base Classes

```
class B1 {
   int A;
   public:
      B1(int Z) { A = Z;}
      int GetA() { return A; }
};
class B2 {
      int B;
   public:
      B2 (int Y) { B = Y; }
      void GetB() { return B; }
};
class D : public B1, public B2 {
      int C;
    public:
      D (int X, int Y, int Z) : B1(Z), B2 (Y) { C = X; }
      void ShowAll () {
         cout << GetA() << " " << GetB() << " " << C << endl; }
};
```




```
int main() {
   D Ob(5,7,9);

   Ob.ShowAll();

   return 0;
} // end main
```

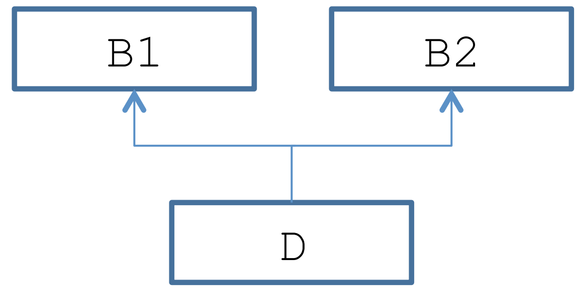
C++ Inheritance

- **Inheritance Multiple Base Classes (constructor and destructor)**

```
class B1 {
   public:
       B1() {cout << "Constructing B1\n"; }
      ~B1() {cout << "Destructing  B1\n";  }
};
class B2 {
   public:
       B2() {cout << "Constructing B2\n"; }
      ~B2() {cout << "Destructing  B2\n";  }
};
```

3 1 2

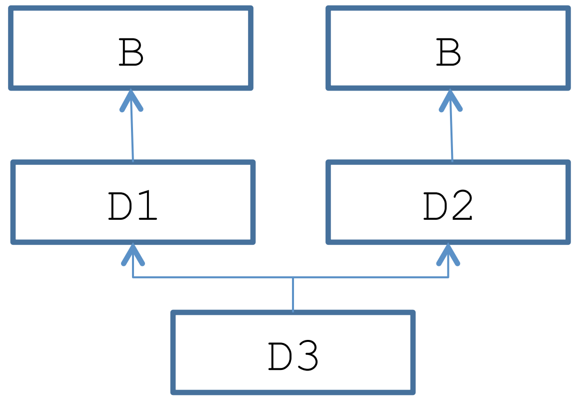
```
class D : public B1, public B2 {
   public:
       D() {cout << "Constructing D\n"; }
      ~D() {cout << "Destructing  D\n";  }
};
```




```
int main () {
   D ob;
   return 0;
} // end main

----OUTPUT----
Constructing B1
Constructing B2
Constructing D
Destructing D
Destructing B2
Destructing B1
```

C++ Inheritance

- **Virtual Base Class**
  - **PROBLEM:** The Base B is inherited twice by D3.



  - There is ambiguity!
  - Solution: mechanism by which only one copy of B will be included in D3.

C++ Inheritance

```
class B {
   public:
      int I;
};
class D1 : virtual public B {
   public:
      int J;
};
class D2 : virtual public B {
   public:
      int K;
};
class D3 : public D1, public D2 {
   int product {return I * J * K; }
};
```

```
int main() {
   D3 ob;

   ob.I = 15; //must be virtual
              //   else compile
              //      time error
   ob.J = 21;
   ob.K = 26;

   cout << "Product: "
        << ob.product() << endl;
   return 0;
} // end main
```

C++ Inheritance

- A Derived class does not inherit the constructors of its base class.
- Good Advice: You can and should include a call to one of the base class constructors when you define a constructor for a derived class.
- If you do not include a call to a base class constructor, then the default (zero argument) constructor of the base class is called automatically.
- If there is no default constructor for the base class, an error occurs.

C++ Inheritance

- If the programmer does not define a ***copy constructor*** in a derived class (or any class), C++ will auto-generate a copy constructor for you. (Bit-wise copy)
- Overloaded assignment operators are not inherited, but can be used.
- When the destructor for the derived class is invoked, it auto-invokes the destructor of the base class. No need to explicitly call the base class destructor.

C++ Inheritance

- A derived class inherits all the member functions (and member variables) that belong to the base class – except for the constructor.
- If a derived class requires a different implementation for an inherited member function, the function may be ***redefined*** in the derived class. (not the same *overloading*)
  - List its declaration in the definition of the derived class (even though it is the same as the base class).
  - Redefined function will have the same number and types of parameters. I.e. signature is the same.
  - Ok to use both (must use the base class qualifier to distinguish between the 2)

C++ Inheritance

- **Virtual Functions**
  - Background:
    - A pointer declared as a pointer to a base class can also be used to point to any class derived from that base.
    - We can use a base pointer to point to a derived object, but you can access only those members of the derived object that were inherited from the base. The base pointer has knowledge only of the base class; it knows nothing about the members added by the derived class.
    - A pointer of the derived type cannot (should not) be used to access an object of the base class.

C++ Inheritance

- **Virtual Functions-Background**

```
class Base {
   int X;
public:
   void SetX(int I) { X = I;}
   int  GetX()      { return X:}
};
class Derived : public Base {
   int Y;
public:
   void SetY(int I) { Y = I;}
   int  GetY()      { return Y;}
};
```

```
int main() {
   Base    *ptr;
   Base    BaseOb;
   Derived DerivedOb;

   ptr = &BaseOb;
   ptr→SetX(15);
   cout <<"Base X: "
        << ptr→GetX() << endl;

   ptr = &DerivedOb;
   ptr→SetX(29);

   DerivedOb.SetY(42); // cannot use ptr
   cout << "Derived Object X: "
        << ptr→GetX()      << endl;
   cout << "Derived Object Y: "
        << DerivedOb.GetY() << endl;

   return 0;
} // end main
```

C++ Inheritance

- **Virtual Functions**
  - When the programmer codes “virtual” for a function, the programmer is saying, “ I do not know how this function is implemented”.
  - Technique of waiting *until runtime* to determine the implementation of a procedure is called **late binding** or **dynamic binding**.
  - A virtual function is a member function that is declared within a base class and redefined by a derived class.
  - Demonstrates “One interface, multiple methods” philosophy that is polymorphism.
  - “Run-time polymorphism”- when a virtual function is called through a pointer.
  - When a virtual function is redefined by a derived class,
    the keyword virtual is not needed.
  - “A base pointer points to a derived object that contains a virtual function and that virtual function is called through that pointer, C++ determines which version of that function will be executed based upon the type of object being pointed to by the pointer.” Schildt

C++ Inheritance

- **Virtual Functions**
  - Exact same **prototype** (***Override*** not Overload)
    Signature + return type
  - Can only be class members
  - Destructors can be virtual; constructors cannot.
  - Done at runtime!
  - ***Late Binding:*** refers to events that must occur at run time.
  - ***Early Binding:*** refers to those events that can be known at compile time.

C++ Inheritance

- **Virtual Functions**

```
class Base {
   public:
      int I;
      base(int X) { I = X;}
      virtual void func() {
         cout << "Using Base version of func(): ";
         cout << I << endl;
      }
};
class D1 ; public Base {
   public:
      D1(int X) : base(X) {}
      void func() {
         cout << Using D1's version of func(): ";
         cout << I*I << endl;
      }
};
class D2 : public Base {
   public:
      D2(int X) : base(X) {}
      void func() {
         cout << Using D2's version of func(): ";
         cout << I+I << endl;
      }
};
```

Polymorphic class contains a virtual function.

```
int maint() {
   Base *ptr;
   Base BaseOb(10);
   D1   D1Ob(10);
   D2   D2Ob(10);

   ptr = &BaseOb;
   ptr→func(); // use Base's func()

   ptr = &D1Ob;
   ptr→func(); // use D1's func()

   ptr = &D2Ob;
   ptr→func()' // use D2's func()

   return 0;
}
```

```
----OUTPUT----
Using Base version of func(): 10
Using D1's version of func(): 100
Using D2's version of func(): 20
```

If the derived class does not override a virtual function, the function defined within its base class is used.

C++ Inheritance

```
class Area {
      double dim1, dim2;
   public :
      void SetArea(double d1, double d2) {
         dim1 = d1;
         dim2 = d2;
      }
      void GetDim (double &d1, double &d2) {
         d1 = dim1;
         d2 = dim2;
      }
      virtual double GetArea() {
         cout << "DUMMY DUMMY OVERRIDE function";
         return 0.0;
      }
class Rectangle : public Area {
   public :
      double GetArea() {
         double temp1, temp2
         GetDim (temp1, temp2);
         return temp1 * temp2;
      }
};
class Triangle : public Area {
   public :
      double GetArea() {
         double temp1, temp2
         GetDim (temp1, temp2);
         return 0.5 temp1 * temp2;
      }
};
```

```
int main () {
   Area *ptr;
   Rectangle R;
   Triangle  T;

   R.SetArea(3.3, 4.5);
   T.SetArea(4.0, 5.0);

   ptr = &R;
   cout << "RECTANGLE_AREA: "
        << ptr→GetArea() << endl;

   ptr = &T;
   cout << "TRIANGLE_AREA: "
        << ptr→GetArea() << endl;

   return 0;
} // end main
```

When there is no meaningful action for a base class virtual function to perform, the implication is that any derived class MUST override this function. C++ supports ***pure virtual functions*** *to do this.*

```
Virtual double GetArea() = 0; // pure virtual
```

C++ Inheritance

- **Virtual Functions**
  - When a class contains at least one ***pure*** virtual function, it is referred to as an ***abstract class***.
  - An abstract class contains at least one function for which no body exists,
    so an abstract class exists mainly to be inherited.
  - Abstract classes do not stand alone.
  - If Class B has a virtual function called f(), and D1 inherits B and D2 inherits D1, both D1 and D2 can override f() relative to their respective classes.