# Inheritance

Feza BUZLUCA
Istanbul Technical University
Computer Engineering Department
http://faculty.itu.edu.tr/buzluca
http://www.buzluca.info





1

# Overview

- Introduction
- Reusability
- Inheritance: "Kind-Of" or "Is A" Relationship
- Generalization-Specialization
- Inheritance Syntax
- Access Control for Inheritance
- Redefining Members
- Overloading vs. Overriding
- Special Member Functions and Inheritance
- Constructors and Inheritance
- Destructors and Inheritance
- Assignment Operator and Inheritance
- Composition vs. Inheritance
- Multiple Inheritance
- Repeated Base Classes
- Virtual Base Classes
- Conclusion





2

# Inheritance

- Using inheritance, we can create more “specialized” classes from general classes
- In object-oriented programming, the concept of inheritance provides an important extension to the idea of reusability







# Reusability

- Reusability means taking an existing class and using it in a new program
- By reusing classes, you can reduce the time and effort needed to develop a program and make software more robust and reliable

# History of Reusability

- The earliest approach to reusability was simply rewriting existing code:
  - You have some code that works in an old program, but does not do quite what you want in a new project
  - You paste the old code into your new source file, make a few modifications to adapt it to the new environment
  - Now, you must debug the code all over again
  - Often, you are sorry you did not just write new code
- To reduce the bugs introduced by modification of code, programmers attempted to create self-sufficient reusable program elements in the form of functions
  - Function libraries were a step in the right direction, but functions do not model the real world very well because they do not include important data







# Reusability in Object-Oriented Programming

- A powerful new approach to reusability appears in object-oriented programming: the class library
- Since a class more closely models a real-world entity, it needs less modification than functions do to adapt it to a new situation

# Reusability in Object-Oriented Programming

Once a class has been created and tested, it can be used in different ways again:

1. (Simplest) Just using an object of that class directly. C++ standard library has many useful classes and objects.
   - For example, cin and cout are such built-in objects. Another useful class is string, which is used very often in C++ programs.
2. Placing an object (or a pointer) of that class inside a new class. We call this “creating a member object.” (has-a relation)
   - Your new class can be made up of any number and type of other objects, in any combination that you need to achieve the functionality desired in your new class.
   - Since you are composing a new class from existing classes, this concept is called composition (or more generally, aggregation). Composition is often referred to as a “has-a” relationship. See example e410.cpp.
3. Using inheritance, which is described next. Inheritance is referred to as a "is-a" or "a-kind-of" relationship.






# An Example for Using Classes of the Standard Library: Strings

- While a character array can be fairly useful, it is quite limited
  - It is simply a group of characters in memory, but if you want to do anything with it, you must manage all the little details (e.g., memory allocation)
- The Standard C++ string class is designed to take care of (and hide) all the low-level manipulations of character arrays that were previously required of the C programmer
- To use strings, you include the C++ header file <string>
- Because of operator overloading, the syntax for using strings is quite intuitive (natural)

# An Example for Using Classes of the Standard Library: Strings

```
#include <string>                    // standard C++ header file
#include <iostream>
using namespace std;
int main() {
   string s1, s2;                    // empty strings
   string s3 = "Hello, World.";      // initialize string
   string s4("I am");                // also initialize string
   s2 = "Today";                     // assign to a string
   s1 = s3 + " " + s4;               // combine strings
   s1 += " 20 ";                     // append to a string
   cout << s1 + s2 + "!" << endl;
   return 0;
}
```

**See Example e61.cpp**







# An Example for Using Classes of the Standard Library: Strings

- First two strings, s1 and s2, start out empty
- Strings s3 and s4 show two equivalent ways to initialize string objects from character arrays
  - You can just as easily initialize string objects from other string objects
- You can assign to any string object using the assignment operator '='
- This replaces the previous contents of the string with whatever is on the right-hand side, and you do not have to worry about what happens to the previous contents (that is handled automatically for you)

# An Example for Using Classes of the Standard Library: Strings

- To combine strings, you simply use the '+' operator, which also allows you to combine character arrays with strings
- If you want to append either a string or a character array to another string, you can use the operator '+='
- Note: cout already knows what to do with strings, so you can just send a string (or an expression that produces a string, which happens with `s1 + s2 + "!"`) directly to cout in order to print it





# Inheritance

- OOP provides a way to modify a class without changing its code
  - using inheritance to derive a new class from the old one
- The old class (called the base class) is not modified, but the new class (the derived class) can use all the features of the old one and additional features of its own

# Inheritance: a "Kind-Of" or “Is-A" Relationship

- We know that
  - PCs, Macintoshes, and Crays are kinds of computers
  - a worker, a section manager, and general manager are kinds of employees
- If there is a "kind-of" relationship between two objects, then we can derive one from the other using inheritance






# Generalization - Specialization

- Using inheritance, we can create more “specialized” classes from general classes
  - Employee → worker → manager (Manager is a worker, worker is an employee)
  - Vehicle → air vehicle → helicopter (Vehicle is general, helicopter is special)
- Specialized classes may have more members (data and methods) than general classes

# Inheritance Syntax

- Simplest example of inheritance requires two classes: a base class and a derived class
- Base class does not need any special syntax
- Derived class must indicate that it is derived from the base class






# Inheritance: Example

- Example: Modeling teachers and the principal (director) in a school
- Assume that we have a class to define teachers
- We can use this class to model principal because principal is a teacher

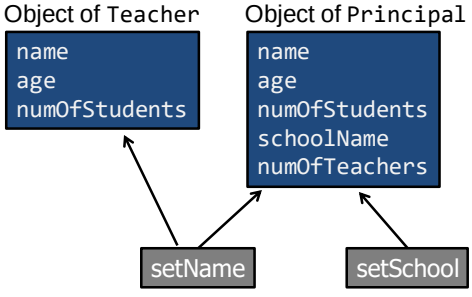
```
class Teacher {       // base class
 protected:           // means public for derived class members
   string name;
   int age, numOfStudents;
 public:
   void setName (const string & newName){ name = newName; }
};
class Principal : public Teacher {   // derived class
   string schoolName;                // additional members
   int numOfTeachers;
 public:
   void setSchool(const string & sName){ schoolName = sName; }
};
```

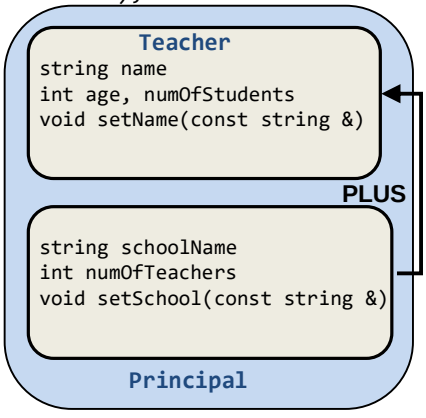
- `Principal` is a special type of `Teacher`
  - It has more members

# Inheritance: Example

```
int main()
{
   Teacher teacher1;
   Principal principal1;
   principal1.setName( "Principal 1" );
   teacher1.setName( "Teacher 1" );
   principal1.setSchool( "Elementary School" );
   return 0;
}
```

**Objects in Memory:**










# Access Control for Inheritance

- An object of a derived class inherits all the member data and functions of the base class
  - child (derived) object `principal1` contains not only data items `schoolName` and `numOfTeachers`, but data items `name`, `age`, `numOfStudents` as well
  - `principal1` object can also access, in addition to its own member function `setSchool()`, the member function that is inherited from parent (base) class, which is `setName()`

# Access Control for Inheritance: Private Members

- Private members of the base class are inherited by the derived class, but they are not visible in the derived class
- The members of the derived class cannot access private members of the base class directly
- The derived class may access them only through the public interface of the base class






# Redefining Members (Name Hiding)

- Some members (data or function) of the base class may not suitable for the derived class
  - These members should be redefined in the derived class
- Example: assume that the `Teacher` class has a print function that prints properties of teachers on the screen
  - However, this function is not sufficient for the class `Principal`, because principals have more properties to be printed
  - So, the print function must be redefined

# Redefining Members: Teacher Class Print Function

```
class Teacher {                        // base class
 protected:
   string name;
   int age, numOfStudents;
 public:
   void setName (const string & newName) { name = newName; }
   void print() const;
};

void Teacher::print() const          // print method of Teacher
class
{
   cout << "Name: " << name << "  Age: " << age << endl;
   cout << "Number of Students: " << numOfStudents << endl;
}
```







# Redefining Members: Principle Class Print Function

```
class Principal : public Teacher { // derived class
   string schoolName;
   int numOfTeachers;
 public:
   void setSchool( const string & sName ) { schoolName = sName; }
   void print() const;              // print method of Principal class
};

void Principal::print() const      // print method of Principal class
{
   cout << "Name: " << name << "  Age: " << age << endl;
   cout << "Number of Students: " << numOfStudents << endl;
   cout << "Name of the school: " <<  schoolName << endl;
}
```

**See Example e62.cpp**

- `print()` function of the `Principal` class overrides (hides) the `print()` function of the `Teacher` class
- Now, the `Principal` class has two `print()` functions

# Redefining Members: Principle Class Print Function

- Members of the base class can be accessed using scope operator (::)

```
void Principal::print() const  // print method of Principal class
{
    Teacher::print();          // invoke print func. of Teacher class
    cout << "Name of the school: " << schoolName << endl;
}
```

**See Example e62.cpp**





# Overloading vs. Overriding

- If you modify the signature and/or the return type of a member function from the base class, then the derived class has two member functions with the same name (see `print()` in the previous example)
- But, this is not overloading, it is overriding
- If the author of the derived class redefines a member function, it means he changes the interface of the base class
- In this case, the member function of the base class is hidden

# Overriding: Example

```
class A {
 public:
   int ia1, ia2;
   void fa1();
   int fa2(int);
};
class B: public A{
 public:
   float ia1;              // overrides ia1
   float fa1(float);       // overrides fa1
};
int main()
{
   B b;
   int j = b.fa2(1);       // A::fa2
   b.ia1 = 4;              // B::ia1
   b.ia2 = 3;              // A::ia2 if ia2 is public in A
   float y = b.fa1(3.14); // B::fa1
   b.fa1();                // ERROR! fa1 func. in B hides function of A
   b.A::fa1();             // OK
   b.A::ia1 = 1;           // OK if ia1 is public in A
   return 0;
}
```

**See Example e63.cpp**





# Access Control

- Remember, when inheritance is not involved,
  - class member functions have access to anything in the class, whether public or private
  - objects of that class have access only to public members
- Once inheritance enters the picture, other access possibilities arise for derived classes
  - Member functions of a derived class can access public and protected members of the base class, but not private members
  - Objects of a derived class can access only public members of the base class

| Access Specifier | Accessible from Own Class | Accessible from Derived Class | Accessible from Objects (Outside Class) |
|---|---|---|---|
| **public** | yes | yes | yes |
| **protected** | yes | yes | no |
| **private** | yes | no | no |

# Access Control: Example

```
class Teacher {                 // base class
  private:                      // only members of Teacher can access
    string name;
  protected:                    // also members of derived classes can
    int age, numOfStudents;
  public:                       // everyone can access
    void setName (const string & newName){ name = newName; }
    void print() const;
};

class Principal : public Teacher {      // derived class
  private:                              // default
    string  schoolName;
    int numOfTeachers;
  public:
    void setSchool( const string & sName ) { schoolName = sName; }
    void print() const;
    int getAge() const { return age; }       // works because age is protected
    const string & getName(){ return name;} // ERROR! because name is private
};
```

protected in Teacher

private in Teacher





# Access Control: Example

```
int main()
{
   Teacher teacher1;
   Principal principal1;
   teacher1.numOfStudents = 100;              // ERROR! (protected)
   teacher1.setName( "Ali Bilir" );           // OK (public)
   principal1.setSchool( "Istanbul Lisesi" ); // OK (public)
   return 0;
}
```

# Protected vs. Private Members

- In general, class data should be private
- Public data is open to modification by any function anywhere in the program and should almost always be avoided
- Protected data is open to modification by functions in any derived class
  - Anyone can derive one class from another and thus gain access to the base class's protected data
- It is safer and more reliable if derived classes cannot access base class data directly
- In real-time systems, where speed is important, function calls to access private members is a time-consuming process
  - In such systems, data may be defined as protected to give derived classes direct and faster access to data







# Protected vs. Private Members

**Private data: Slow and reliable**

```
class A {                        // base class
 private:
   int i;                        // safe
 public:
   void access( int new_i ) {  // public interface to access i
      if ( new_i > 0 && new_i <= 100)
         i = new_i;
   }
};

class B:public A {               // derived class
 private:
   int k;
 public:
   void set( new_i, new_k ) {
      A::access( new_i );          // reliable but slow
       :
   }
};
```

**Protected data: Fast, author of derived class responsible**

```
class A {     // base class
 protected:
   int i;         // derived class can access directly
 public:
    :
};

class B:public A {  // derived class
 private:
   int k;
 public:
   void set( new_i, new_k ) {
      i = new_i;                    // fast
       :
   }
};
```

# Public Inheritance

- In inheritance, you usually want to make the access specifier public

```
class Base
   {  };
class Derived : public Base {
```

- This is called public inheritance (or sometimes public derivation)
- The access rights of the members of the base class are not changed
- Objects of the derived class can access public members of the base class
- Public members of the base class are also public members of the derived class



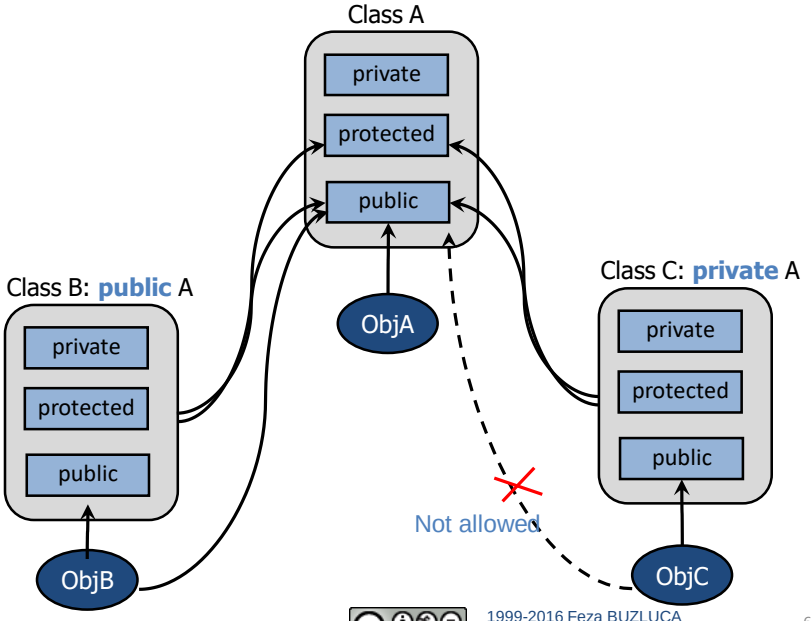


# Private Inheritance

```
class Base
   {  };
class Derived : private Base {
```

- This is called private inheritance
- Now, public members of the base class are private members of the derived class
- Objects of the derived class cannot access members of the base class
- Member functions of the derived class can still access public and protected members of the base class

# Public vs. Private Inheritance










# Redefining Access Specifications

- Access specifications of public members of the base class can be redefined in the derived class
- When you inherit privately, all the public members of the base class become private
- If you want any of them to be visible, just say their names (no arguments or return values) along with the using keyword in the public section of the derived class:

```
class Base {
 private:
   int k;
 public:
   int i;
   void f();
};

class Derived : private Base {   // all members of Base are private now
   int m;
 public:
   using Base::f;               // f() is public again, i is still private
   void fb1();
};
```

```
int main(){
   Base b;
   Derived d;
   b.i = 5;  // OK public in Base
   d.i = 0;  // ERROR private inherit.
   b.f();    // OK
   d.f();    // OK
   return 0;
};
```

# Special Member Functions and Inheritance

- Some functions will need to do different things in the base class and the derived class
  - overloaded assignment (=) operator
  - the destructor
  - all constructors







# Special Member Functions and Inheritance

- Consider a constructor
- The base class constructor must create the base class data, and the derived class constructor must create the derived class data
- Because the derived class and base class constructors create different data, one constructor cannot be used in place of another
- Constructor of the base class cannot be the constructor of the derived class

# Special Member Functions and Inheritance

- Similarly, the = operator in the derived class must assign values to derived class data, and the = operator in the base class must assign values to base class data
- These are different jobs, so assignment operator of the base class cannot be the assignment operator of the derived class






# Constructors and Inheritance

- When you define an object of a derived class, the base class constructor will be called before the derived class constructor
- This is because the base class object is a subobject (a part) of the derived class object, and you need to construct the parts before you can construct the whole
- If the base class has a constructor that needs arguments, this constructor must be called before the constructor of the derived class

```
class Teacher {                                        // base class
   string name;
   int age, numOfStudents;
 public:
   Teacher(const string & newName): name( newName )// constructor of base
   { }                                                  // empty body
};
class Principal : public Teacher {                    // derived class
   int numOfTeachers;
 public:
   Principal( const string &, int );     // constructor of derived class
};
```

**See Example e64.cpp**

# Constructors and Inheritance

```
// constructor of the derived class
// constructor of base called before body of constructor of derived class
Principal::Principal( const string & newName, int numOT ):Teacher( newName )
{
   numOfTeachers = numOT;
}
```

- Remember, the constructor initializer can also be used to initialize members

```
// Constructor of the derived class
Principal::Principal( const string & newName, int numOT )
                              :Teacher(newName), numOfTeachers(numOT)
 { }                                  // body of the constructor is empty
```

- If the base class has a constructor, which must take some arguments, then the derived class must also have a constructor that calls the constructor of the base with proper arguments

**See Example e65.cpp**

- The same rule also applies to the copy constructor
- The copy constructor of the derived class must call the proper constructor of the base class (see example e68.cpp later)







# Destructors and Inheritance

- Destructors are called automatically
- When an object of the derived class goes out of scope, the destructors are called in reverse order: The derived object is destroyed first, then the base class object

```
class B {                       // base class
 public:
   B() { cout << "B constructor" << endl; }
   ~B() { cout << "B destructor" << endl; }
};

class C : public B {            // derived class
 public:
   C() { cout << "C constructor" << endl; }
   ~C() { cout << "C destructor" << endl; }
};
int main()
{
   cout << "Start" << endl;
   C ch;                        // create an object of derived class
   cout << "End" << endl;
   return 0;
}
```

**Result:**

```
Start
B constructor
C constructor
End
C destructor
B destructor
```

**See Example e66.cpp**

## Constructors and Destructors in a Chain of Classes

```
class A
{
 private:
   int intA;
   float floA;
 public:
   A(int i, float f) : intA(i), floA(f)  //initialize A
      { cout << "Constructor A" << endl;  }
   void display() const
      { cout << intA << ", " << floA << "; "; }
   ~A() { cout << "Destructor A" << endl;  }
};
```

```
class B : public A
{
 private:
   int intB;
   float floB;
 public:
   B(int i1, float f1, int i2, float f2) :
      A(i1, f1),                    // initialize A
      intB(i2), floB(f2)            // initialize B
      { cout << "Constructor B" << endl; }
   void display() const
   {
      A::display();
      cout << intB << ", " << floB << "; ";
   }
   ~B() { cout << "Destructor B" << endl;  }
};
```

```
class C : public B
{
 private:
   int intC;
   float floC;
 public:
   C(int i1, float f1, int i2, float f2, int i3, float f3) :
      B(i1, f1, i2, f2),                // initialize B
      intC(i3), floC(f3)                // initialize C
      { cout << "Constructor C" << endl; }
   void display() const
   {
      B::display();
      cout << intC << ", " << floC;
   }
   ~C() { cout << "Destructor C" << endl;  }
};
```

```
int main()
{
  C c( 1, 1.1, 2, 2.2, 3, 3.3 );
  cout << endl << "Data in c = ";
  c.display();
  return 0;
}
```

**See Example e67.cpp**







## Constructors and Destructors in a Chain of Classes

- A C class is inherited from a B class, which is in turn inherited from an A class
- The constructor in each class takes enough arguments to initialize the data for the class and all ancestor classes
- This means two arguments for the A class constructor, four for B (which must initialize A, as well as itself), and six for C (which must initialize A and B, as well as itself). Each constructor calls the constructor of its base class.
- When a constructor starts to execute, it is guaranteed that all the subobjects are created and initialized

# Constructors and Destructors in a Chain of Classes

- Incidentally, you cannot skip a generation when you call an ancestor constructor in an initialization list
- In the following modification of the C constructor:

```
C(int i1, float f1, int i2, float f2, int i3, float f3) :
                    A(i1, f1),          // ERROR! cannot initialize A
                  intC(i3), floC(f3)    // initialize C
   {  }
```

- The call to A() is illegal because the A class is not the immediate base class of C
- You never need to make explicit destructor calls because there is only one destructor for any class, and it does not take any arguments
- The compiler ensures that all destructors are called, and that means all of the destructors in the entire hierarchy, starting with the most-derived destructor and working back to the root

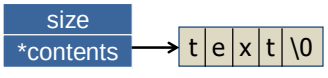





# Assignment Operator and Inheritance

- Assignment operator of the base class cannot be the assignment operator of the derived class
- Recall the String example:

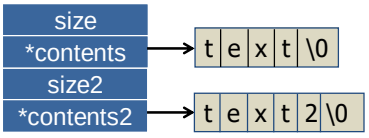
```
class String{
 protected:
   int size;
   char *contents;
 public:
   const String & operator=( const String & );    // assignment operator
     :                                             // other methods
};

const String & String::operator=( const String &inObject )
{
   size = inObject.size;
   delete[] contents;                              // delete old contents
   contents = new char[size+1];
   strcpy(contents, inObject.contents);
   return *this;
}
```



# Assignment Operator and Inheritance

- Example: Class String2 is derived from class String
- If an assignment operator is necessary, it must be written

```
class String2 : public String{                    // String2 is derived from String
   int size2;
   char  *contents2;
 public:
   const String2 & operator=(const String2 &);    // assignment operator for String2
   :                                              // other methods
};
// **** Assignment operator for String2 ****
const String2 & String2::operator=( const String2 &inObject )
{
    size = inObject.size;                  // inherited size
    delete[ ] contents;
    contents = new char[size + 1];         // inherited contents
    strcpy(contents, inObject.contents);
    size2 = inObject.size2;
    delete[ ] contents2;
    contents2 = new char[size2 + 1];
    strcpy(contents2, inObject.contents2);
    return *this;
}
```










# Assignment Operator and Inheritance

- In previous example, data members of String (base) class must be protected
  - Otherwise, methods of the String2 (derived) class cannot access them
- The better way to write the assignment operator of String2 is to call the assignment operator of the String (base) class
- Now, data members of String (base) class may be private

```
//** Assignment operator **
const String2 &  String2::operator=( const String2 & inObject )
{
   String::operator=(inObject);              // call the operator= of String (base)
   size2 = inObject.size2;
   delete[] contents2;
   contents2 = new char[size2 + 1];
   strcpy( contents2, inObject.contents2 );
   return *this;
}
```

**See Example e68.cpp**
**Also check the copy constructor**

- In this method, the assignment operator of the String is called with an argument of type (String2 &)
- Actually, the operator of String class expects a parameter of type (String &).
- This does not cause a compiler error, because as we will see in Lecture 7, a reference to base class can carry the address of an object of derived class

# Composition (*has-a* relation)

- Every time you place instance data in a class, you are creating a “has-a” relationship
- If there is a class Teacher and one of the data items in this class is the teacher’s name, we can say that a Teacher object has a name
- This sort of relationship is called composition because the Teacher object is composed of these other variables
- Remember the class ComplexFrac
  - This class is composed of two Fraction objects







# Composition (*has-a* relation) vs. Inheritance (*is-a* relation)

- Composition in OOP models the real-world situation in which objects are composed of other objects
- Inheritance in OOP mirrors the concept that we call generalization - specialization in the real world
- If we model workers, managers, and researchers in a factory, we can say that these are all specific types of a more general concept called an employee
  - Every kind of employee has certain features: name, age, ID num, and so on
  - But, a manager, in addition to these general features, has a department that he manages
  - In this example, the manager has not an employee. The manager is an employee

# Composition and Inheritance

- You can use composition and inheritance together
- The following example shows the creation of a more complex class using both

```
class A {
   int i;
public:
   A(int ii) : i(ii)
    {}
   ~A() {}
   void f() const
    {}
};
```

```
class B {
   int i;
public:
   B(int ii) : i(ii) {}
   ~B() {}
   void f() const {}
};
```

```
class C : public B {   // inheritance, C is B
   A a;                // composition, C has A
public:
   C(int ii) : B(ii), a(ii) {}
   ~C() {}             // calls ~A() and ~B()
   void f() const {    // redefinition
      a.f();
      B::f();
   }
};
```

**See Example e69.cpp**







# Composition and Inheritance

- C inherits from B and has a member object ("is composed of") of type A
- Constructor initializer list contains calls to both base-class constructor and member-object constructor
- Function C::f( ) redefines B::f( ), which it inherits, and also calls the base-class version. In addition, it calls a.f( )
- Notice that the only time you can talk about redefinition (overriding) of functions is during inheritance; with a member object, you can only manipulate the public interface of the object, not redefine it.
- In addition, calling f( ) for an object of class C would not call a.f( ) if C::f( ) had not been defined, whereas it would call B::f( )

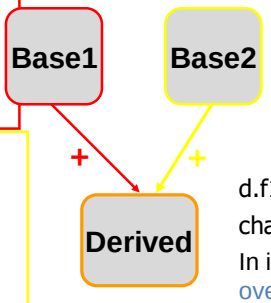
# Multiple Inheritance

- Multiple inheritance occurs when a class inherits from two or more base classes, like this:

```
class Base1 {        // Base 1
 public:
   void f1();
   char *f2(int);
};
```

```
class Base2 {       // Base 2
 public:
   char *f2(int, char);
   int f3();
   void f4();
};
```

```
class Derived : public Base1 , public Base2 {
 public:
   float f1(float);      // override Base1
   void f4();            // override Base2
   int f5(int);
};
```




```
int main()
{
  Derived d;
  float y = d.f1(3.14);// Derived::f1
  d.f3();              // Base2::f3
  d.f4();              // Derived::f4
  d.Base2::f4();       // Base2::f4
  return 0;
}
```

d.f1(); // ***ERROR !***

char *c = d.f2(1); // ***ERROR !***

In inheritance, functions are not overloaded. They are **overridden**.

You have to write

```
char *c = d.Base1::f2(1);        // Base1::f2
```

or

```
char *cc = d.Base2::f2(1,'A'); // Base2::f2
```

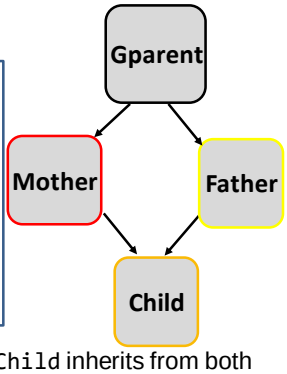
**See Example e610.cpp**




51

# Repeated Base Classes

```
class Gparent
   {   };
class Mother : public Gparent
   {   };
class Father : public Gparent
   {   };
class Child : public Mother, public Father
   {   };
```




- Both `Mother` and `Father` inherit from `Gparent`, and `Child` inherits from both `Mother` and Father
- Recall that each object created through inheritance contains a subobject of the base class
- A `Mother` object and a `Father` object will contain subobjects of `Gparent`, and a `Child` object will contain subobjects of `Mother` and `Father`, so a `Child` object will also contain two `Gparent` subobjects, one inherited via Mother and one inherited via `Father`
- This is a strange situation. There are two subobjects when really there should be only one.




52

# Repeated Base Classes

- Suppose there is a data item in Gparent:

```
class Gparent
{
   protected:
      int gdata;
};
```

and you try to access this item from Child:

```
class Child : public Mother, public Father
{
 public:
   void Cfunc()
   {
      int temp = gdata;       // ERROR: ambiguous
   }
};
```

- The compiler will complain that the reference to gdata is ambiguous
- It does not know which version of gdata to access: the one in the Gparent subobject in the Mother subobject or the one in the Gparent subobject in the Father subobject





# Virtual Base Classes

- You can fix this using a new keyword, virtual, when deriving `Mother` and `Father` from `Gparent`:

```
class Gparent
   {   };
class Mother : virtual public Gparent
   {   };
class Father : virtual public Gparent
   {   };
class Child : public Mother, public Father
   {   };
```

**See Example e611.cpp**

- The virtual keyword tells the compiler to inherit only one subobject from a class into subsequent derived classes
- That fixes the ambiguity problem, but other more complicated problems arise that are too complex to delve into here
- In general, you should avoid multiple inheritance
- If you have considerable experience in C++, you might find reasons to use it in some situations

# Conclusion

- We can create specialized types from general types
- We can reuse the base class without changing its code
- We can add new members, redefine existing members, and redefine access specifications without touching the base class