

Inheritance and Polymorphism

Danilo Ardagna

Politecnico di Milano
danilo.ardagna@polimi.it



Danilo Ardagna - Inheritance 2

Content

- PIE properties
- Basics
- Protected Members and Class Access
- Polymorphism and Virtual Member Functions
- Derived-to-Base Conversion
- Virtual Functions
- Abstract Classes
- Constructors and Destructors
- Containers and Inheritance

object oriented programming
OOP PIE properties

- Polymorphism
- Inheritance
- Encapsulation

Danilo Ardagna - Inheritance 4

Encapsulation

- Encapsulation: binds the data & function in one form known as Class. The data & function may be private or public
 - By thinking the system as composed of independent objects, we keep sub-parts really independent
 - They communicate only through well-defined method invocation
 - Different groups of programmers can work on different parts of the project, just making sure they comply with an interface
 - It is possible to build larger systems with less effort
- Building the system as a group of interacting objects:
 - Allows extreme modularity between pieces of the system
 - May better match the way we (humans) think about the problem
 - Avoids recoding, increases code-reuse

Danilo Ardagna - Inheritance 5

What is Inheritance?

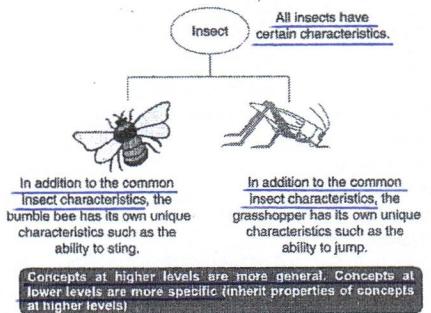
- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class
- Motivations: **code reuse and evolution**

Advantages of Inheritance

- When a class inherits from another class, there are three benefits. You can:
 - reuse** the methods and data of the existing class
 - extend** the existing class by adding new data and new methods
 - modify** the existing class by overloading/overriding its methods with your own implementations

linked with
polymorphism

Example: Insect Taxonomy



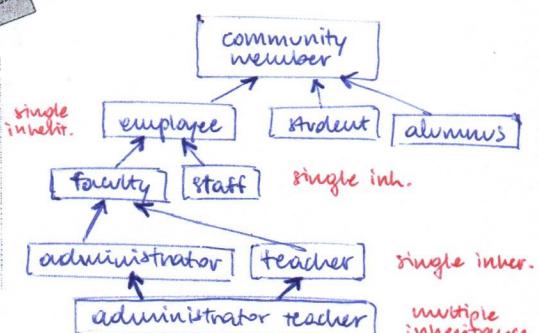
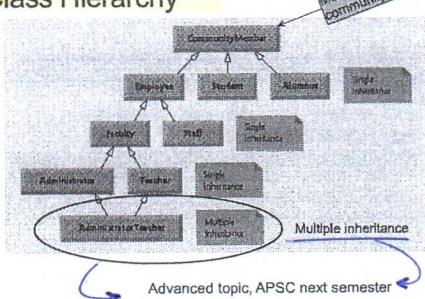
The "is a" Relationship

- Inheritance establishes an "is a" relationship between classes:
 - A dachshund is a dog
 - A car is a vehicle
 - A flower is a plant
 - A football player is an athlete



A "specific" is a "general".

Class Hierarchy



Basics

- Inheritance is a mean of specifying hierarchical relationships between types
- C++ classes inherit both data and function members from other (parent) classes
- Terminology: "the child (derived or subclass) type **inherits** (or is *derived from*) the parent (base or superclass) type"

The derived type is just the base type plus:

- Added specializations
 - Change implementation without changing the base class interface
- Added Generalizations/Extensions
 - New operations and/or data

What a derived class inherits

- Every data member defined in the parent class (although such members may not always be accessible in the derived class!)
- Every ordinary member function of the parent class (although such members may not always be accessible in the derived class!)

What a derived class doesn't inherit

- The base class constructors and destructor
 - The base class assignment operator
 - The base class friends
- Since all these functions are class-specific

if the child has more members
for more it'll need an other constructor

What a derived class can add

- New data members
- New member functions (also overwrite existing ones)
- New constructors and destructor

Inheritance – C++ Syntax

- Notation:

```
class Student // base class
{
    ...
};

class UnderGrad : public Student // derived class
{
    ...
};
```

Define a Class Hierarchy

Syntax:

```
class DerivedClassName : access-level BaseClassName  
(Inheritance type)
```

Where

- access-level specifies the type of derivation
 - private by default, or
 - public/protected
- we will use always and only public inheritance

Note that any class can serve as a base class

- Thus a derived class can also be a base class

it can be public, private, protected.
We'll use only "public"

Back to the 'is a' Relationship

An object of a derived class 'is a(n)' object of the base class

Example:

- an UnderGrad is a Student
- a Mammal is an Animal

A derived object has all of the characteristics of the base class...

... and possibly something more

What Does a Child Have?

An object (an instance!) of the derived class has:

- all members declared in parent class
- all members defined in child class

An object of the derived class can use:

- all public members defined in parent class
- all public members defined in child class

Objects of the derived class can use/access only PUBLIC MEMBERS - both DATA and FUNCTIONS members.

Members of class

private: are accessible only in the class itself

public: are accessible anywhere (outside the class)

protected: are accessible in subclasses of the class and inside the class

something in between public and private

Inherited objects cannot access, only the subclasses can access
(INHERITED OBJECTS CAN ACCESS ONLY TO PUBLIC MEMBERS)

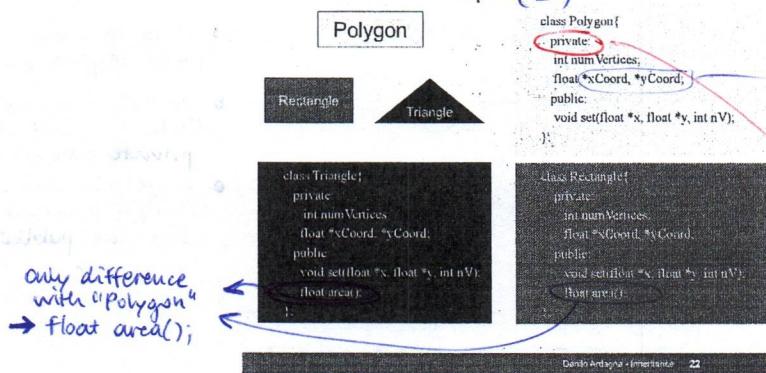
Members of class

private: are accessible only in the class itself

public: are accessible anywhere (outside the class)

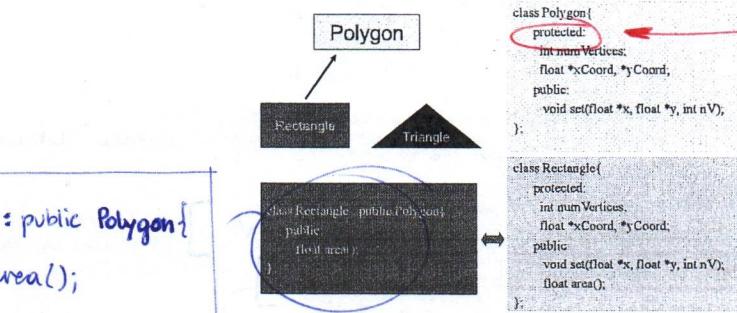
protected: are accessible in subclasses of the class and inside the class

Inheritance Concept (1)



pointers because xCoord and yCoord are always (with coordinates)

Inheritance Concept



to be able to pass these to subclasses we have to switch from private to protected

Note: if something is private it'll be still included in the subclass, however it'll not be possible to access to that (unless)

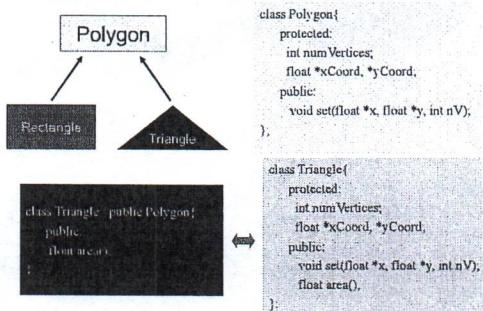
```

class Rectangle : public Polygon{
public:
    float area();
}

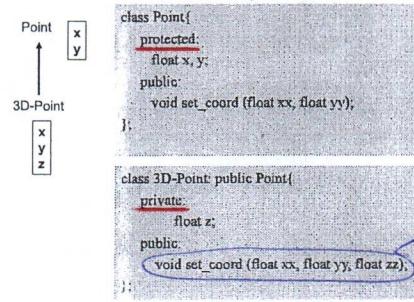
```

in this way we just have to specify the new things

Inheritance Concept



Inheritance Concept (2)



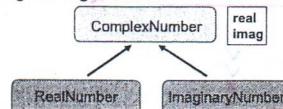
we're changing the setter (3 parameters instead of 2) (overloading)

Inheritance Concept

- Augmenting the original class



- Specializing the original class



protected Members

- Like private, protected members are inaccessible to users of the class (objects!)
- Like public, protected members are accessible to members (and friends) of classes derived from this class
- In addition, protected has another important property:
 - A derived class member may access the protected members of the base class only through a derived object. The derived class has no special access to the protected members of base-class objects

we have to make a distinction between objects and classes

- protected members from the OBJECTS point of view are like private (we cannot use)
- protected members from the DERIVED CLASSES point of view are like public



protected Members

! (*) from the point of view of "Base" the "Sneaky" functions are external functions! we cannot apply "Sneaky" functions on "Base" objects

```
class Base {
protected:
    int prot_mem; // protected member
};

class Sneaky : public Base {
    void clobber1(Sneaky&); // can access Sneaky::prot_mem
    void clobber2(Base&); // can't access Base::prot_mem
    int j; // j is private by default
};

// ok: clobber1 can access the private and protected members in Sneaky objects
void Sneaky::clobber1(Sneaky &s) { s.j = s.prot_mem = 0; } ✓

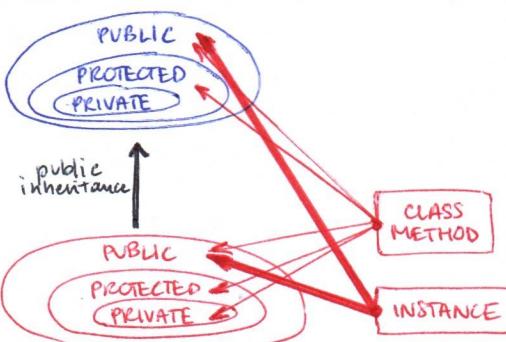
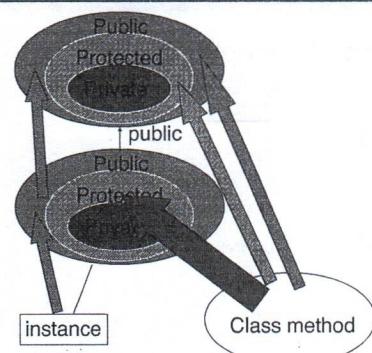
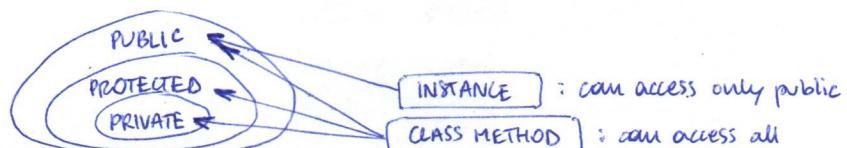
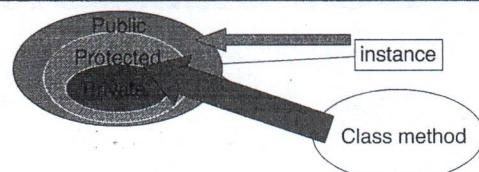
// error: clobber2 can't access the protected members in Base
void Sneaky::clobber2(Base &b) { b.prot_mem = 0; } ✗
```

DEMO

} all private (default)

} a "Sneaky" function trying to access a private member of a "Sneaky" object ✓

} a "Sneaky" function trying to access a private member of a "Base" object ✗ (*)



Example

```
class Base {
public:
    int pub_mem(); // public member
protected:
    int prot_mem; // protected member
private:
    char priv_mem; // private member
};

class Pub_Derv : public Base {
public:
    int f() { return prot_mem; } ✓
private:
    char g() { return priv_mem; } ✗
};

Pub_Derv d1;
int i1 = d1.pub_mem(); ✓
int ii = d1.f(); ✓
char c = d1.g(); ✗
int iii = d1.prot_mem(); ✗
```

Example

```

class Base {
public:
    int pub_mem(); // public member
protected:
    int prot_mem; // protected member
private:
    char priv_mem; // private member
};

class Pub_Derv : public Base {
public:
    // ok: derived classes can access protected members
    int f() { return prot_mem; } // return this->prot_mem !
private:
    // error: private members are inaccessible to derived classes
    char g() { return priv_mem; }
};

Pub_Derv d1;
int i = d1.pub_mem(); // members inherited from Base are public
int ii = d1.f(); // ok
char c = d1.g(); // error g is private !
int iii = d1.prot_mem; // error protected member cannot be accessed by objects !

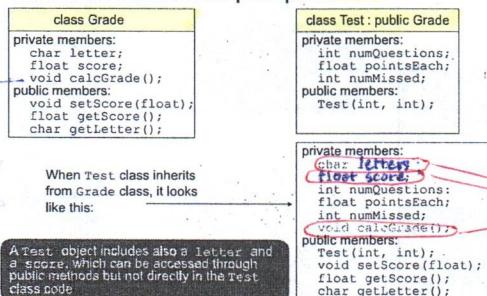
```

Note: also:

Base b;

b.prot_mem = 1;

error!

because protected
are like private**Inheritance vs. Access****sub-class methods perspective**given the score it
calculates the letter

they're part of the class
but we cannot access to them
(directly at least, because we
always have the getters/setters)

When a derived-class object is created & destroyed

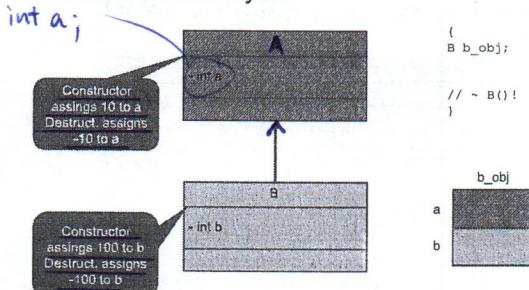
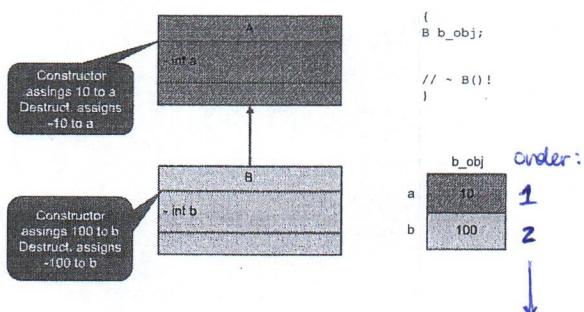
- Space is allocated (on the stack or the free store) for the full object (that is, enough space to store the data members inherited from the base class plus the data members defined in the derived class itself)
- The base class constructor is called to initialize the data members inherited from the base class
- The derived class constructor is then called to initialize the data members added in the derived class
- The derived-class object is then usable
- When the object is destroyed (goes out of scope or is deleted) the derived class destructor is called on the object first
- Then the base class destructor is called on the object
- Finally, the allocated space for the full object is reclaimed

Inheritance always brings everything from the ancestors: we cannot drop things, we can just add things. It may happen that some members cannot be used directly.

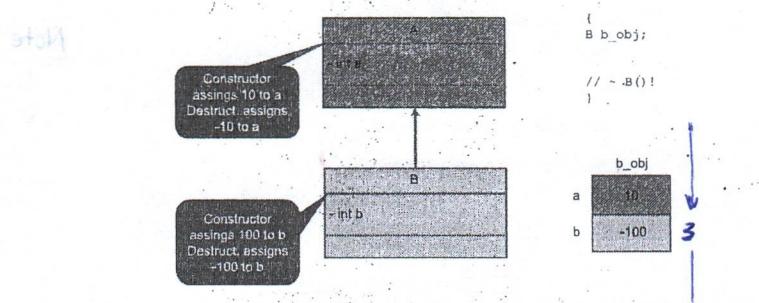
This is the order (IMPORTANT)

When a derived-class object is created & destroyedbecause of the
(possible) private
members

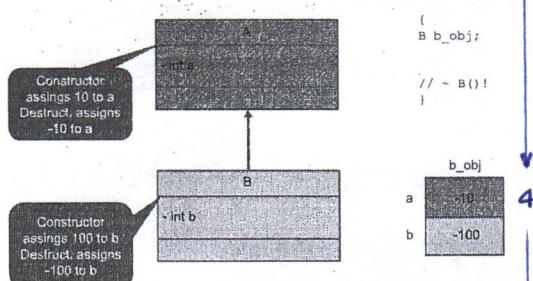
Example:

**When a derived-class object is created & destroyed**

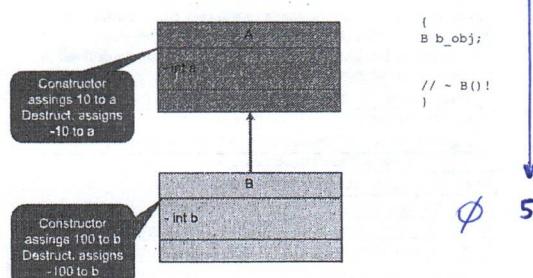
When a derived-class object is created & destroyed



When a derived-class object is created & destroyed



When a derived-class object is created & destroyed



Constructors and Destructors in Base and Derived Classes

Derived Classes
(basically the same thing as above)

```
1 // This program demonstrates the order in which base and
2 // derived class constructors and destructors are called.
3 #include <iostream>
4 using namespace std;
5
6 //*****//*****//*****//*****//*****//*****//*****//
7 // Baseclass declaration           *
8 //*****//*****//*****//*****//*****//*****//*****//
```

Constructors and Destructors in Base and Derived Classes

```
16 class BaseClass
17 {
18 public:
19     BaseClass() // Constructor
20     { cout << "This is the BaseClass constructor.\n"; }
21     ~BaseClass() // Destructor
22     { cout << "This is the BaseClass destructor.\n"; }
23 };
24
25 //*****+*****+*****+*****+*****+*****+*****+*****+
26 // DerivedClass declaration
27 //*****+*****+*****+*****+*****+*****+*****+*****+
28
29 class DerivedClass : public BaseClass
30 {
31 public:
32     DerivedClass() // Constructor
33     { cout << "This is the DerivedClass constructor.\n"; }
34     ~DerivedClass() // Destructor
35     { cout << "This is the DerivedClass destructor.\n"; }
36 };
37 }
```

Constructors and Destructors in Base and Derived Classes

```

34 //*****
35 // main function
36 //*****
37
38 int main()
39 {
40     cout << "We will now define a DerivedClass object.\n";
41     DerivedClass object;
42
43     cout << "The program is now going to end.\n";
44
45     return 0;
46 }

```

Program Output:

```

We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.

```

Program output:

We will now define a DerivedClass object.
 This is the BaseClass constructor.
 This is the DerivedClass constructor.
 The program is now going to end.
 This is the DerivedClass destructor.
 This is the BaseClass destructor.

Class design principle

- In the absence of inheritance, we can think of a class as having two different kinds of users: ordinary users (which are **programmers!**) and implementors
- Ordinary users:
 - write code that uses objects of the class type
 - such code can access only the **public** (interface) members of the class
- Implementors:
 - write the code contained in the members (and friends) of the class
 - the members (and friends) of the class can access both the **public** and **private** implementation sections

Class design principle

- Under inheritance, there is a **third kind of user**, namely, **derived classes**
- A base class makes protected those parts of its implementation that it is willing to let its derived classes use
- The protected members remain inaccessible to ordinary user code; private members remain inaccessible to derived classes
- General approach: **be the strictest as possible!**

Class design principle

- Like any other class, a class that is used as a base class makes its **interface members public**
- An **implementation member** should be **protected**:
 - if it **provides** an operation or data that a **derived class** will need to use in its own implementation
 - should be **private** otherwise

Polymorphism and Virtual Member Functions

Polymorphism

- **Poly = Many, Morphism = forms**
- The ability of objects to respond differently to the same message or function call
- An object has "multiple identities", based on its class inheritance tree
 - It can be used in different ways
- Two types:
 - Compile-time polymorphism
 - Run-time polymorphism

Polymorphism

- **Poly = Many, Morphism = forms**
- The ability of objects to respond differently to the same message or function call
- An object has "multiple identities", based on its class inheritance tree
 - It can be used in different ways
- Two types:
 - Compile-time polymorphism → **Templates**
 - Run-time polymorphism

} for example: vector template behaves differently in different situations (we can define vector of integers, vector of Books, vector of doubles). Vectors behave differently according to the type.
This happens at compile time.

Overwriting methods

- A subclass can overwrite, i.e., change, a base class method behaviour
- Three mechanisms:
 - Overloading
 - Redefinition ←
 - Overriding ←
- Overriding provides polymorphism and is the most powerful mechanism for changing a method behavior

Redefining Base Class Functions

- **Redefining function:** function in a derived class has the same name and parameter list as a function in the base class
- Typically used to replace a function in base class with different actions in derived class
- Not the same as overloading – with overloading, parameter lists must be different
- **Redefinition:** Objects of base class use the base class version of the function; objects of derived class use the derived class version of the function

Redefining Base Class Functions

- In C++, a base class distinguishes functions that are type dependent from those that it expects its derived classes to inherit without change
 - The base class defines as virtual those functions it expects its derived classes to define for themselves
- Derived classes frequently, but not always, override the virtual functions that they inherit
 - If a derived class does not (redefine or) override a virtual from its base, then, like any other member, the derived class inherits the version defined in its base class

Polymorphism and Virtual Member Functions

- **Virtual member function:** function in base class that expects to be overridden in derived class
- Function defined with key word `virtual`:
`virtual void y() {...}`
- Supports **dynamic binding**: functions bound at run time to function that they call
- **Without virtual member functions, C++ uses static (compile time) binding** and it is only function redefinition
 - However, this is not the only pre-requisite for dynamic binding

If we don't use "virtual" then we're **REDEFINING**. To have dynamic binding we need to use "virtual" (even if it's not enough, it's required)

Base Class

```
class Quote {
public:
    Quote() = default;
    Quote(const string &book, double sales_price):
        bookNo(book), price(sales_price) {}
    string isbn() const { return bookNo; } (getter)
    // returns the total sales price for the specified number of items
    // derived classes will override and apply different discount
    // algorithms
    virtual double net_price(size_t cnt) const
    { return cnt * price; }
    // dynamic binding for the destructor
    virtual ~Quote() = default;
private:
    string bookNo; // ISBN number of this item
protected:
    double price = 0.0; // normal, undiscounted price
};
```

will initialize the price to zero if called

(whenever we use inheritance)
 whenever we use something with "virtual" we have to define a **virtual destructor**, otherwise we may have troubles with the children of this class

generally, just to be sure
 (we never know if a class will be used in inheritance) add a **virtual destructor**

Derived Class

```
class Bulk_quote : public Quote // Bulk_quote inherits
                           // from Quote
public:
    Bulk_quote() = default;
    Bulk_quote(const string &book, double sales_price,
               size_t min_qty, double disc_rate);
    // overrides the base version in order to implement the bulk
    // purchase discount policy
    double net_price(size_t cnt) const override;
private:
    size_t min_qty = 0; // minimum purchase for the discount
    // to apply
    double discount = 0.0; // fractional discount to apply
};
```

these are from the other class (father)

we redefine the method and we're adding "override"
 (we're not repeating that it's virtual)
 (it's not overloading: the parameters are the same)

Derived Class

```
// if the specified number of items are purchased, use the discounted
// price
double Bulk_quote::net_price(size_t cnt) const
{
    if (cnt <= min_qty)
        return cnt * price;
    else
        return cnt * (1 - discount) * price;
}
```

Dynamic Binding

- Through **dynamic binding**, we can use the same code to process objects of either type `Quote` or `Bulk_quote` interchangeably:

```
// calculate and print the price for the given number of copies,
// applying any discounts
double print_total(const Quote &item, size_t n)
{
    // depending on the type of the object bound to the item parameter
    // calls either Quote::net_price or Bulk_quote::net_price
    double ret = item.net_price(n);
    cout << "ISBN: " << item.isbn() // calls Quote::isbn()
    << "# sold: " << n << " total due: " << ret << endl;
    return ret;
}
```

DYNAMIC BINDING *

The dynamic binding consists in: here we're passing an object of `Quote` class (the father) and we rely on the implementation of `net_price` of `Quote`, however we want to be able to pass a `Bulk_quote` object (subclass, child) and rely on `net_price` of `Bulk_quote`. If we pass an object of the base class we use the base class implementation. Otherwise if we pass an object of a subclass we'll rely on the subclass implementation.

* Opportunity to use one implementation or another at runtime according to the type of the object that we're passing to the function

Polymorphism Requires References or Pointers

- Polymorphic behavior is only possible when an object is referenced by a **reference** variable or a **pointer**, as demonstrated in the `print_total` function

DYNAMIC BINDING:
 1. virtual
 2. override
 3. reference/pointers

Dynamic Binding

```
// basic has type Quote; bulk has type Bulk_quote
print_total(basic, 20); // calls Quote version of net_price
print_total(bulk, 20); // calls Bulk_quote version of net_price
```

DEMO

Wrapping up...

- A base class specifies that a member function should be **dynamically bound** by preceding its declaration with the keyword **virtual**
- Any non-static member function other than a constructor, may be **virtual** (**and the destructor should be!**)
- The **virtual** keyword appears **only on the declaration inside** the class and may not be used on a function definition that appears outside the class body
- A **function that is declared as virtual in the base class is implicitly virtual in the derived classes** as well
- Member functions that are not declared as **virtual** are resolved at compile time, not at run time
 - For the `isbn` member, this is exactly the behavior we want

Redefining vs. Overriding

- In C++, **redefined functions are statically bound** and **overridden functions are dynamically bound**
- So, a **virtual function is overridden**, and a **non-virtual function is redefined**

Dynamic Binding summary

- The member function is declared as **virtual** in the **base class**
- The member function is declared with the **override** specifier in the **child class**
- The member function is **run through a pointer or a reference to the base class object**

Overwriting methods

- A subclass can overwrite, i.e., change, a base class method behaviour
- Three mechanisms:
 - Overloading (same method name, different parameters)
 - Redefinition (same method name, same parameters)
 - no virtual in the base class or missing any of the three conditions for overriding
 - Overriding (same method name, same parameters)
 - base class method declared as virtual
 - method overridden in the sub-class
 - method invocation through a pointer or reference of a base class object
- Overriding result:
 - base or sub class methods used interchangeably through the same code
 - this happens at runtime

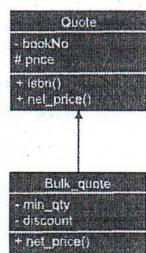
!! d

Derived-to-base Conversion

Derived-Class Objects and the Derived-to-Base Conversion

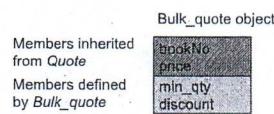
- A derived object contains multiple parts:
 - a subobject containing the (nonstatic) members defined in the derived class itself
 - subobjects corresponding to each base class from which the derived class inherits

Derived-Class Objects and the Derived-to-Base Conversion



Derived-Class Objects and the Derived-to-Base Conversion

- A Bulk_quote object will contain four data elements:
 - the bookNo and price data members that it inherits from Quote
 - the min_qty and discount members, which are defined by Bulk_quote
- Although C++ 11 does not specify how derived objects are laid out in memory, we can think of a Bulk_quote object as consisting of two parts



Derived-Class Objects and the Derived-to-Base Conversion

- Because a derived object contains subparts corresponding to its base class(es), we can use an object of a derived type as if it were an object of its base type(s)
- In particular, we can bind a base-class reference or pointer to the base-class part of a derived object

```
Quote item; // object of base type
Bulk_quote bulk; // object of derived type
Quote *p = &item; // p points to a quote object
p = &bulk; // p points to the Quote part of bulk
Quote &r = bulk; // r bounds to the Quote part of bulk
```

Conversions and Inheritance

- Ordinarily, we can bind a reference or a pointer only to an object that has the same type as the corresponding reference or pointer or to a type that involves an acceptable const conversion
- Classes related by inheritance are an important exception:
 - We can bind a pointer or reference to a base-class type to an object of a type derived from that base class
 - For example, we can use a `Quote&` to refer to a `Bulk_quote` object, and we can assign the address of a `Bulk_quote` object to a `Quote*`

Conversions and Inheritance

- The fact that we can bind a reference (or pointer) to a base-class type to a derived object has an important implication:
 - When we use a reference (or pointer) to a base-class type, we don't know the actual type of the object to which the pointer or reference is bound
 - That object can be an object of the base class or it can be an object of a derived class

Static Type and Dynamic Type

- When we use types related by inheritance, we often need to distinguish between the **static type** of a variable or other expression and the **dynamic type** of the object that expression represents
- The static type of an expression is always known at compile time
 - It is the type with which a variable is declared or that an expression yields
- The dynamic type is the type of the object in memory that the variable or expression represents. The dynamic type may not be known until run time

Static Type and Dynamic Type

- In `print_total()` we have:

```
double ret = item.net_price(n);
```

- We know that the static type of `item` is `Quote&`
- The dynamic type depends on the type of the argument to which `item` is bound
 - That type cannot be known until a call is executed at run time
 - If we pass a `Bulk_quote` object to `print_total`, then the static type of `item` will differ from its dynamic type
 - The static type of `item` is `Quote&`, but in this case the dynamic type is `Bulk_quote`

Static Type and Dynamic Type

- The dynamic type of an expression that is neither a reference nor a pointer is always the same as that expression static type
- For example:
 - A variable of type `Quote` is always a `Quote` object
 - There is nothing we can do that will change the type of the object to which that variable corresponds

Derived-Class Constructors

- A derived object contains members that it inherits from its base but it cannot directly initialize those members
- A derived class must use a **base-class constructor** to initialize its base-class part

```
Bulk_quote(const string& book, double p,
           size_t qty, double disc) :
    Quote(book, p), min_qty(qty), discount(disc) { }
// as before
};
```

- Unless we say otherwise, the base part of a derived object is default initialized
- To use a different base-class constructor, we provide a constructor initializer using the name of the base class, followed by a parenthesized list of arguments
- Those arguments are used to select which base-class constructor to use to initialize the base-class part of the derived object

Inheritance and static Members

- If a base class defines a static member, there is only one such member defined for the entire hierarchy
- Regardless of the number of classes derived from a base class, there exists a single instance of each static member

```
class Base {
public:
    static void statmem();
};

class Derived : public Base {
    void f(const Derived&);
};
```

Inheritance and static Members

- Static members obey normal access control
- If the member is private in the base class, then derived classes have no access to it
- Assuming the member is accessible, we can use a static member through either the base or derived

```
void Derived::f(const Derived &derived_obj)
{
    Base::statmem(); // ok: Base defines statmem
    Derived::statmem(); // ok: Derived inherits statmem
    // ok: derived objects can be used to access static from
    // base
    derived_obj.statmem(); // accessed through a Derived
    // object
    statmem(); // accessed through this object
}
```

Abstract Classes

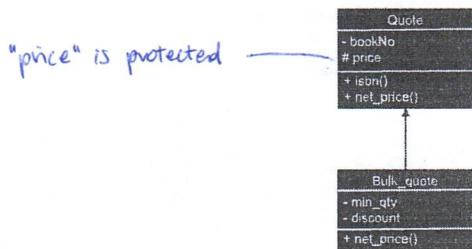
Abstract Base Classes and Pure Virtual Functions

- Pure virtual function:**
 - a virtual member function that must be overridden in a derived class that has objects
 - must have no function definition in the base class
- Abstract base class contains at least one pure virtual function:
`virtual void f() = 0;`
- The `= 0` indicates a pure virtual function

Abstract Base Classes and Pure Virtual Functions

- A class becomes an **abstract base class** when one or more of its member functions is a pure virtual function
- Abstract base classes:
 - Cannot have any objects
 - Serve as a basis for derived classes that may/will have objects

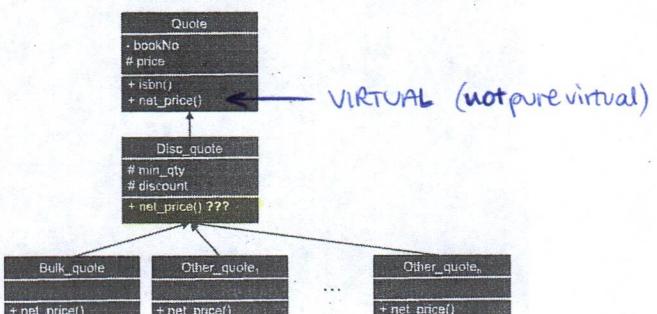
Abstract Base Classes and Pure Virtual Functions



Abstract Base Classes and Pure Virtual Functions

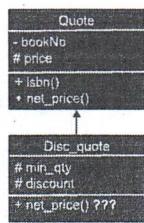
- Imagine that we want to extend our bookstore classes to support several discount strategies
 - Bulk_quote: discount for purchases above a certain limit but not for purchases up to that limit
 - In addition to a bulk discount, we might offer a discount for purchases up to a certain quantity and then charge the full price thereafter
 - Each of these discount strategies is the same in that it requires a quantity and a discount amount
 - We might support these differing strategies by defining a new class named Disc_quote to store the quantity and the discount amount
 - Classes, such as Bulk_quote, that represent a specific discount strategy will inherit from Disc_quote

Abstract Base Classes and Pure Virtual Functions



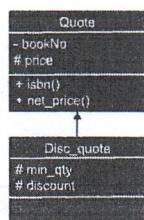
Abstract Base Classes and Pure Virtual Functions

- Each of the derived classes will implement its discount strategy by defining its own version of `net_price`
- Before we can define our `Disc_Quote` class, we have to decide what to do about `net_price`
- Our `Disc_quote` class doesn't correspond to any particular discount strategy



Abstract Base Classes and Pure Virtual Functions

- Each of the derived classes will implement its discount strategy by defining its own version of `net_price`
- Before we can define our `Disc_Quote` class, we have to decide what to do about `net_price`
- Our `Disc_quote` class doesn't correspond to any particular discount strategy
- We could define `Disc_quote` without its own version of `net_price`
- In this case, `Disc_quote` would inherit `net_price` from `Quote`



but we don't want it!



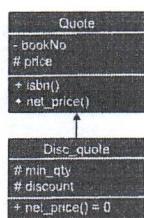
Abstract Base Classes and Pure Virtual Functions



- This design would make it possible for our users to write nonsensical code
- A user could create an object of type `Disc_quote` by supplying a quantity and a discount rate
- Passing that `Disc_quote` object to a function such as `print_total` would use the `Quote` version of `net_price`
- The calculated price would not include the discount that was supplied when the object was created

Abstract Base Classes and Pure Virtual Functions

- Our problem is not just that, we don't know how to define `net_price`
- In practice, we'd like to prevent users from creating `Disc_quote` objects at all
- This class represents the general concept of a discounted book, not a concrete discount strategy
- We can enforce this design intent—and make it clear that there is no meaning for `net_price`—by defining `net_price` as a pure virtual function



Abstract Base Classes and Pure Virtual Functions

*// class to hold the discount rate and quantity
// derived classes will implement pricing strategies using these data*

```

class Disc_quote : public Quote {
public:
    Disc_quote() = default;
    Disc_quote(const string& book, double price,
               size_t qty, double disc):
        < Quote(book, price), quantity(qty),
        discount(disc) {}
    virtual double net_price(std::size_t) const = 0;
protected:
    size_t quantity = 0; // purchase size for the discount to
                         // apply
    double discount = 0.0; // fractional discount to apply
}
  
```

Disc_quote relies on quote constructor!



Abstract Base Classes and Pure Virtual Functions

```
// class to hold the discount rate and quantity
// derived classes will implement pricing strategies using these data
class Disc_quote : public Quote {
public:
    Disc_quote() = default;
    Disc_quote(const string& part_of_inheriting_objects,
               size_t qty);
    Quote(book, price), quantity(qty),
    discount(disc) {}
    virtual double net_price(std::size_t) const = 0;
protected:
    size_t quantity = 0; // purchase size for the discount to
                         // apply
    double discount = 0.0; // fractional discount to apply
};
```

Disc_quote is an abstract class

- Because Disc_quote defines net_price as a pure virtual, we cannot define objects of type Disc_quote
- We can define objects of classes that inherit from Disc_quote, only if those classes override net_price:

```
// Disc_quote declares pure virtual functions, which Bulk_quote will
// override
Disc_quote discounted; // error: can't define a Disc_quote object
Bulk_quote bulk; // ok: Bulk_quote has no pure virtual functions
```

DEMO

Refactoring

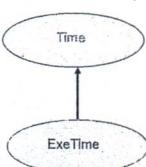
- Adding Disc_quote to the Quote hierarchy is an example of refactoring
- Refactoring involves redesigning a class hierarchy to move operations and/or data from one class to another
- Refactoring is common in object-oriented applications
- Even though we changed the inheritance hierarchy, code that uses Bulk_quote or Quote would not need to change
- However, when classes are refactored (or changed in any other way) we must recompile any code that uses those classes

Take Home Message

- Inheritance is a mechanism for defining new class types to be a specialization or an augmentation of existing types
- In principle, every member of a base class is inherited by a derived class with different access permissions, except for the constructors

Putting Them Together

Putting Them Together



- Time is the base class
- ExeTime is the derived class with public inheritance
- The derived class can
 - inherit all members from the base class, except the constructor
 - access all public and protected members of the base class
 - define its private data member
 - provide its own constructor
 - define its public member functions
 - override functions inherited from the base class

class Time Specification

```

// SPECIFICATION FILE          ( time.h )
class Time{
public:
void Set ( int h, int m, int s );
void Increment ();
virtual void Write () const;
Time ( int initH, int initM, int initS ); // constructor
Time () = default;                      // default constructor
protected:
int     hrs = 0;
int     mins = 0;
int     secs = 0;
};
  
```

Derived Class ExeTime

```

// SPECIFICATION FILE          ( exetime.h )
#include "time.h"
enum ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT};
class ExeTime : public Time
// Time is the base class and use public inheritance
{
public:
void Set ( int h, int m, int s, ZoneType timeZone );
void Write () const override; // overridden
ExeTime ( int initH, int initM, int initS, ZoneType initZone );
ExeTime(); // default constructor
private:
ZoneType zone; // added data member
};
  
```

enum is basically an integer. We enumerate the values that we can use to assign (EST=0, CST=1, MST=2, ...) Once we do this, in the code we can write:

*ZoneType var;
var = MST;*

equivalent to:

*int var;
var = 2;*

Implementation of ExeTime

Default Constructor

```

ExeTime :: ExeTime ( )
{
    zone = EST;
}
  
```

The default constructor of base class, Time(), is automatically called, when an ExeTime object is created.

ExeTime et1;

et1

hrs = 0
mins = 0
secs = 0
zone = EST

Implementation of ExeTime

Another Constructor

```

ExeTime :: ExeTime ( int initH, int initM, int initS, ZoneType initZone )
: Time ( initH, initM, initS ) // constructor initializer
{
    zone = initZone;
}
  
```

```

ExeTime *et2 =
new ExeTime(8,30,0,EST);
et2
  
```

5000 (that's the memory address)

et2 stores a memory address

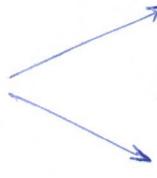
hrs = 8
mins = 30
secs = 0
zone = EST

Implementation of ExeTime

```
void ExeTime :: Set (int h, int m, int s, ZoneType timeZone)
{
    Time :: Set (h, m, s); // same name function call
    zone = timeZone;
}

void ExeTime :: Write () const // function overriding
{
    string zoneString[8] =
        {"EST", "CST", "MST", "PST", "EDT", "CDT", "MDT", "PDT"};
    Time :: Write ();
    cout << zoneString[zone] << endl;
}
```

we rely on functions
of the father
(superclass)



Working with ExeTime

```
#include "exetime.h"
...
int main()
{
    ExeTime thisTime (0, 35, 0, PST);
    ExeTime thatTime; // default constructor called
    thatTime.Write(); // outputs 00:35:00 EST
    thatTime.Set (16, 49, 23, CDT);
    thatTime.Write(); // outputs 16:49:23 CDT
    thisTime.Increment();
    thisTime.Increment();
    thisTime.Write(); // outputs 08:35:02 PST
}
```

References

- Lippman Chapter 15

Credits

- Bjarne Stroustrup. www.stroustrup.com/Programming
- UTDallas CS1 slides- Inheritance, Polymorphism, and Virtual Functions
- Sam Vanthath. Inheritance
- Gordon College CPS212 slides. C++ Inheritance
- NJIT CIS 601 slides. Inheritance in C++
- WPI CS-2303. Derived Classes in C++