deep se

dependable evolvable pervasive software engineering group

Inheritance and Polymorphism

Danilo Ardagna

Politecnico di Milano

danilo.ardagna@polimi.it

21/10/2024

POLITECNICO DI MILANO

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

OOP PIE properties

- Polymorphism
- Inheritance
- Encapsulation

without these 3 properties, a language is not considered object-oriented.

object oriented.

Encapsulation

WE HAVE ALREADY SEEN THAT.

- 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**

What is Inheritance?

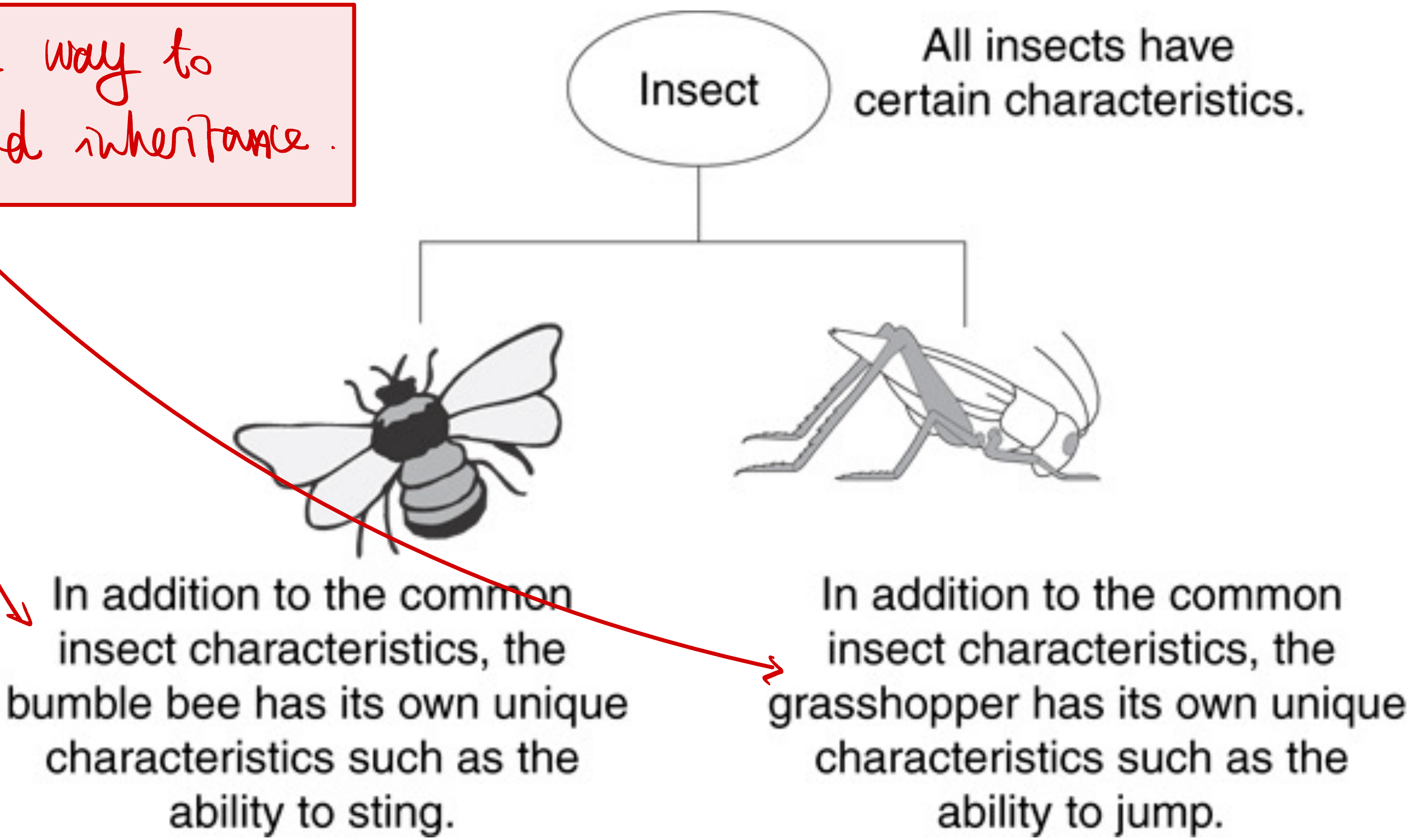
- Provides a way to create a **new** class **from** an **existing** class
- The new class is a specialized version of the existing class
- **Motivations: code reuse and evolution**

Advantages of Inheritance

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

Example: Insect Taxonomy

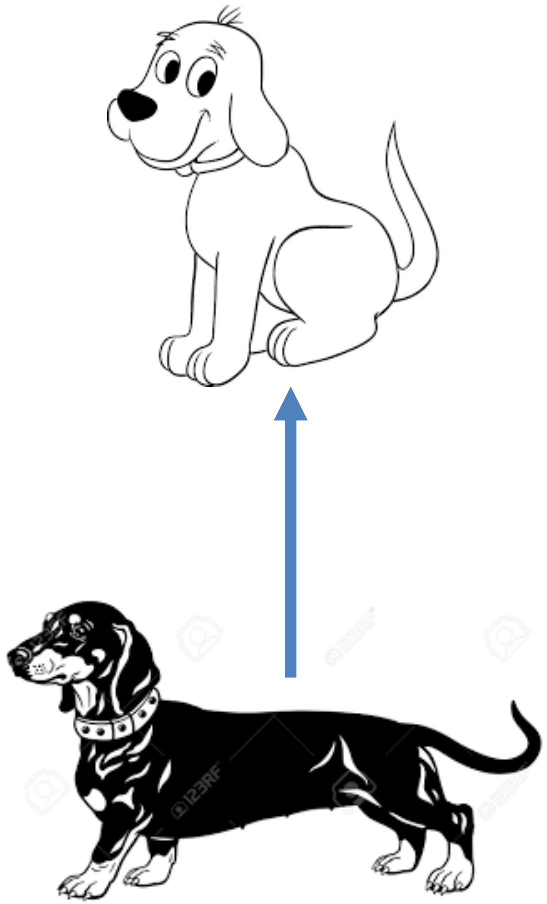
Intuitive way to understand inheritance.



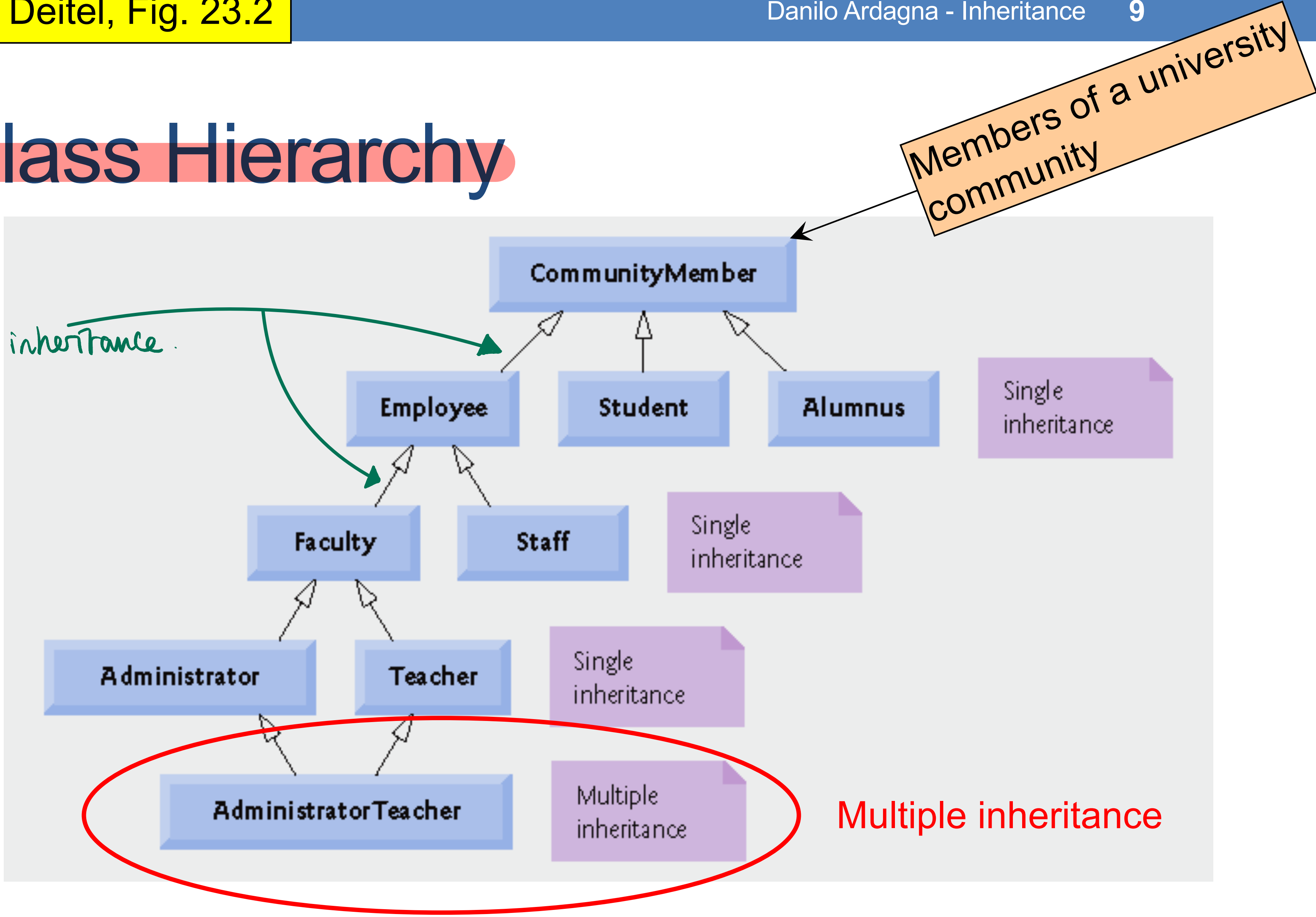
Concepts at higher levels are more general. Concepts at lower levels are more specific (inherit properties of concepts at higher levels)

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



Class Hierarchy



Advanced topic, APSC next semester

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**

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
      . . .
};
```

In this class we will only use "public" here.

Define a Class Hierarchy

we will always use "public" in APC.

- Syntax:

class DerivedClassName : **access-level** BaseClassName
(inheritance type)

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

- Note that any class can serve as a base class
  - Thus a derived class can also be a base class

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 Object 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

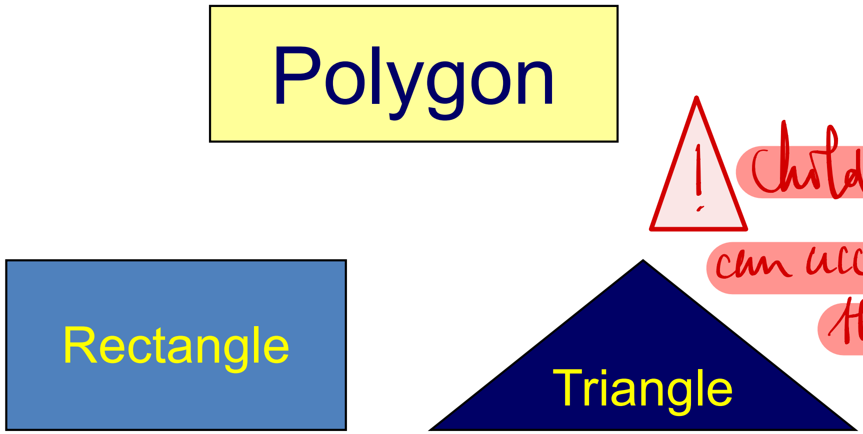
Members of class

- **private**: are accessible only in the class itself
- **public**: are accessible anywhere (outside the class and then from objects)
- **protected**: are accessible in subclasses of the class and inside the class

(children)

NEW!

Inheritance Concept



! Children can access this!

since, in general, we don't know how to compute it!

```
class Polygon{
  private:
   int numVertices;
   float *xCoord, *yCoord;
  public:
   void set(float x[], float y[], int nV);
};
```

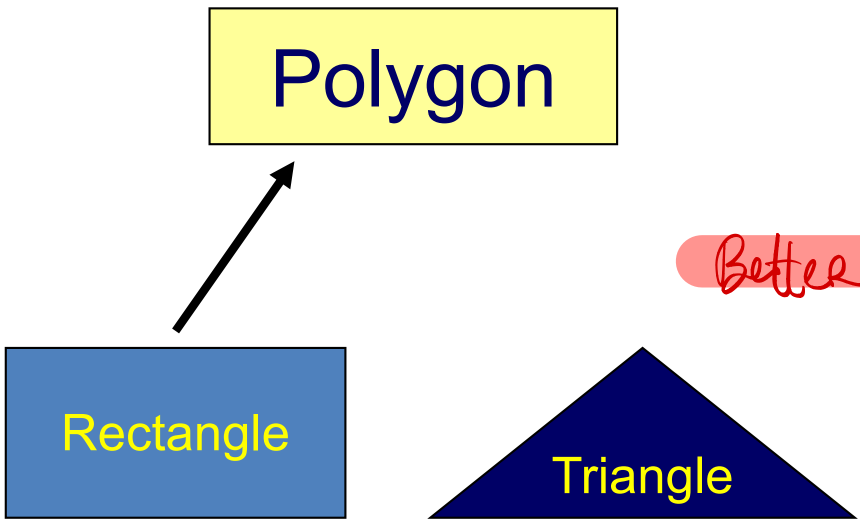
Here I can't have a method area()

```
class Triangle{
  private:
    int numVertices;
   float *xCoord, *yCoord;
  public:
   void set(float x[], float y[], int nV);
   float area();
};
```

```
class Rectangle{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float x[], float y[], int nV);
    float area();
};
```

Here we can have an area() method since we know how to compute it for rectangles & triangles.

Inheritance Concept



Better!

```
class Polygon{
    protected:
      int numVertices;
      float *xCoord, *yCoord;
    public:
       void set(float x[], float y[], int nV);
};
```

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

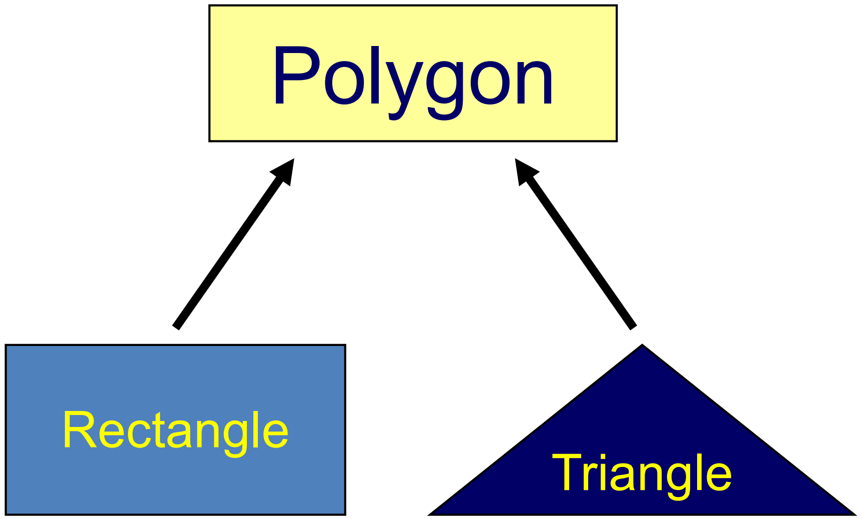
```
class Rectangle{
    protected:
      int numVertices;
      float *xCoord, *yCoord;
    public:
       void set(float x[], float y[], int nV);
       float area();
};
```

Just specify the differences!

Now we can just write this → I have all from the protected part of Polygon.

Inheritance Concept

Same as in the previous slide.

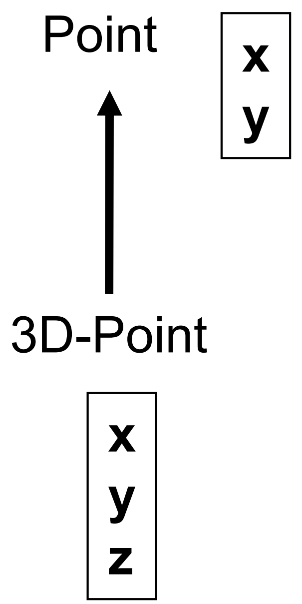



```
class Polygon{
    protected:
      int numVertices;
      float *xCoord, *yCoord;
    public:
       void set(float x[], float y[], int nV);
};
```

```
class Triangle : public Polygon{
    public:
     float area();
};
```

```
class Triangle{
    protected:
      int numVertices;
      float *xCoord, *yCoord;
    public:
       void set(float x[], float y[], int nV);
       float area();
};
```

Inheritance Concept



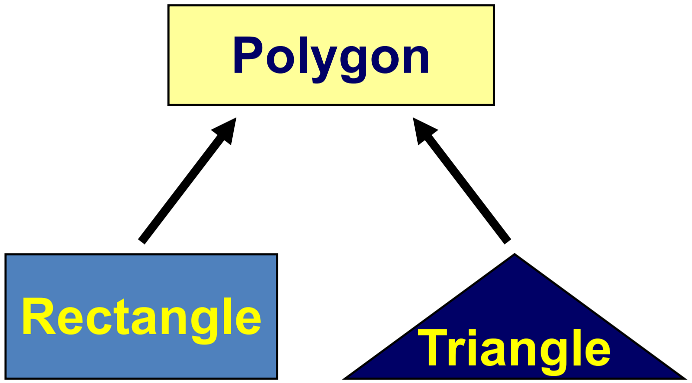

```
class Point{
    protected:
        float x, y;
    public:
        void set_coord (float xx, float yy);
};
```

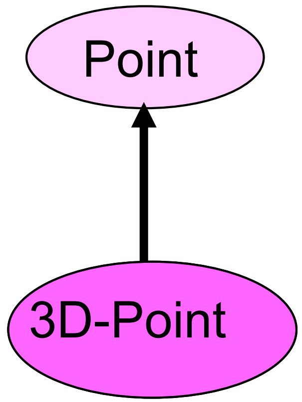
```
class 3D-Point: public Point{
    private:
            float z;
    public:
        void set_coord (float xx, float yy, float zz);
};
```

here we only add the new, OR ≠, things!

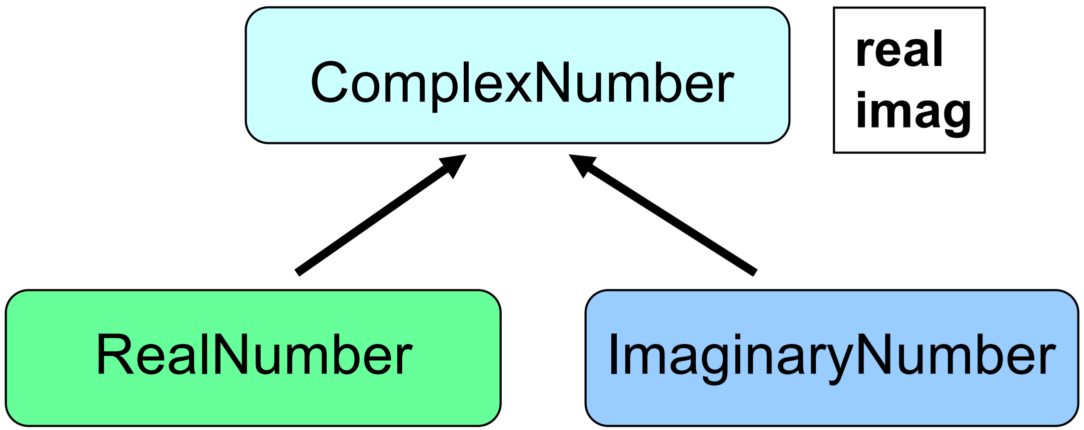
Inheritance Concept

- Augmenting the original class





- Specializing the original class



`protected` Members

- Like private, **protected members are inaccessible** to users of the class **(objects!)** initial class
- Like public, **protected members** of the initial class **are accessible** to **methods (**and friends) of **classes derived** from a class

- In addition, protected has another important property:
  - A derived class method 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

cf "sneaky" code example

on GitHub.

`protected` Members

cf "sneaky".

DEMO

```
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

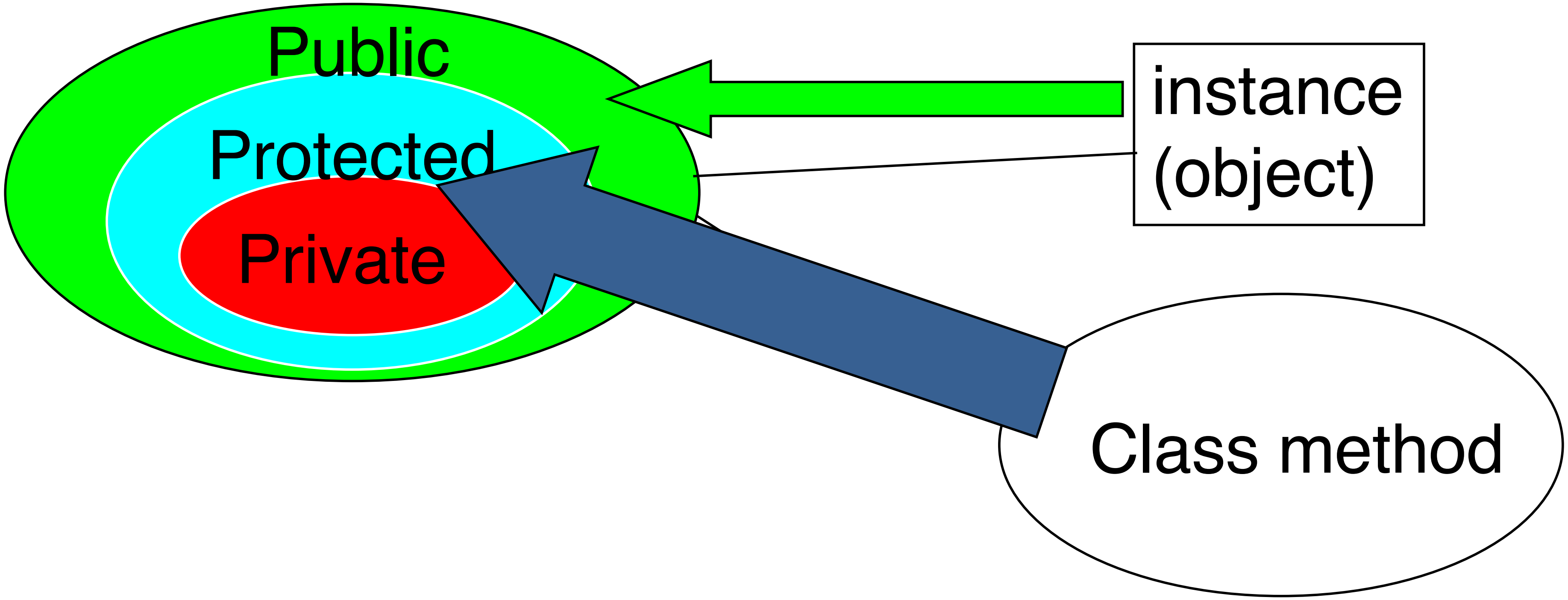
```
void Sneaky::clobber1(Sneaky &s) { s.j = s.prot_mem = 0; }
```

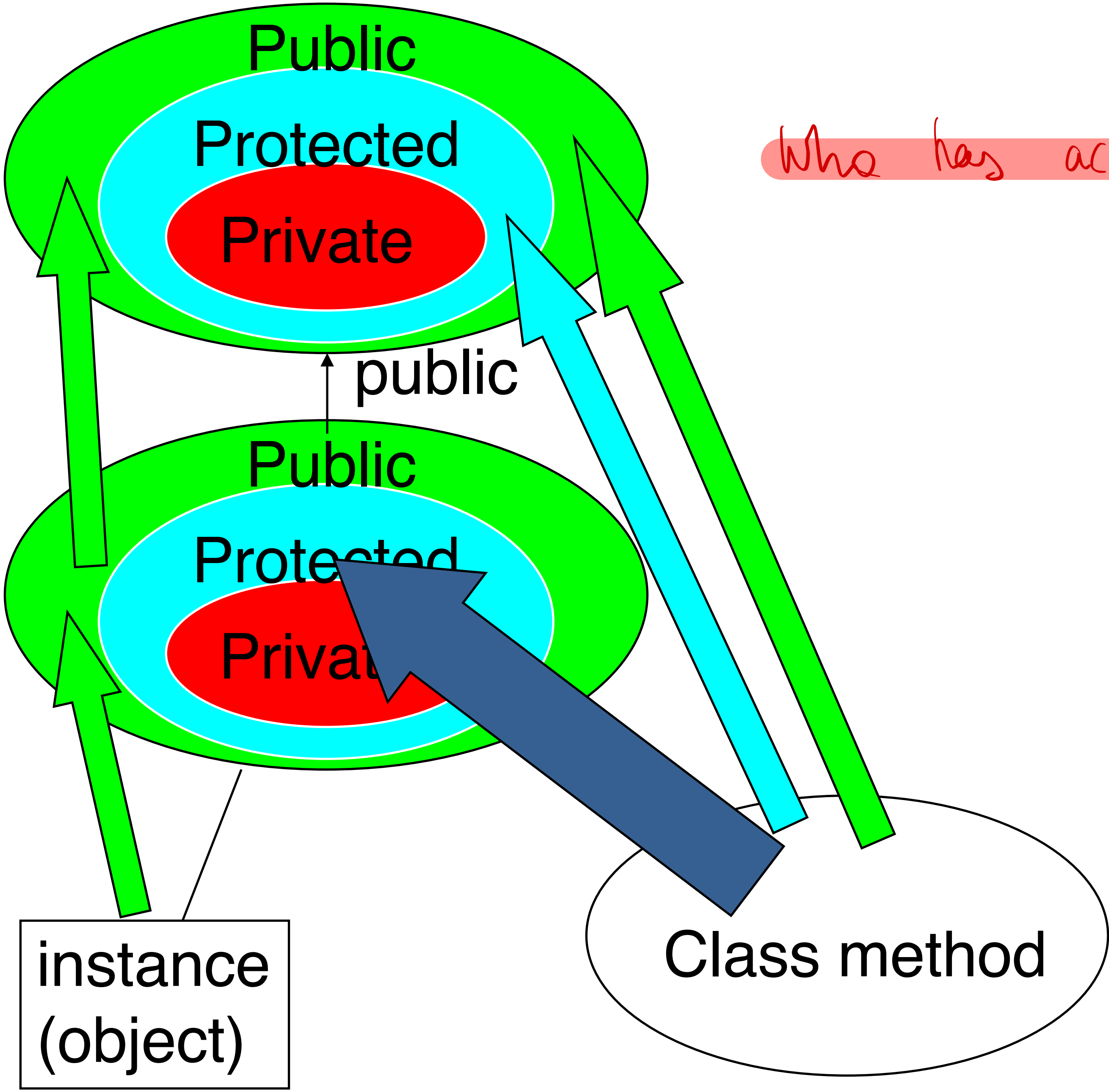
// error: clobber2 can't access the protected members in Base

```
void Sneaky::clobber2(Base &b) { b.prot_mem = 0; }
```

Members of class

- **private**: are accessible only in the class itself
- **public**: are accessible anywhere (outside the class and then from objects)
- **protected**: are accessible in subclasses of the class and inside the class


Public
Protected
Private
instance
(object)
Class method



Who has access to what?

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 i= d1.pub_mem();
int ii = d1.f();
char c = d1.g();
int iii = d1.prot_mem;
```

Inheritance vs. Access
sub-class methods perspective

class Grade

private members:
```
char letter;
float score;
void calcGrade();
```
public members:
```
void setScore(float);
float getScore();
char getLetter();
```

class Test : public Grade

private members:
```
int numQuestions;
float pointsEach;
int numMissed;
```
public members:
```
Test(int, int);
```

When `Test` class inherits from `Grade` class, it looks like this:

private members:
```
char letter;
float score;
int numQuestions:
float pointsEach;
int numMissed;
void calcGrade();
```
public members:
```
Test(int, int);
void setScore(float);
float getScore();
char getLetter();
```

A `Test` object includes also a `letter` and a `score`, which can be accessed through public methods but not directly in the `Test` class code

When a derived-class object is created & destroyed

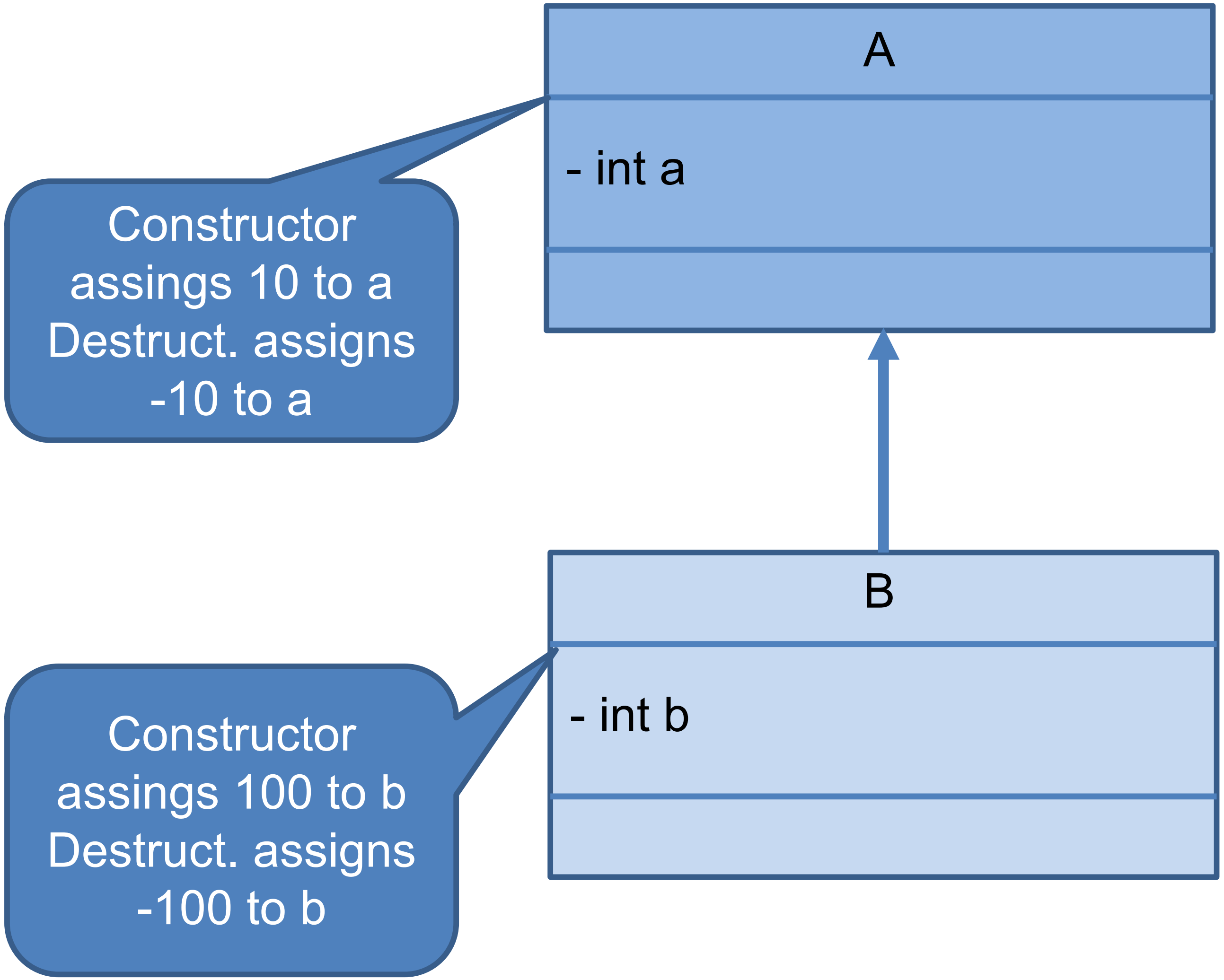
creation

- 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

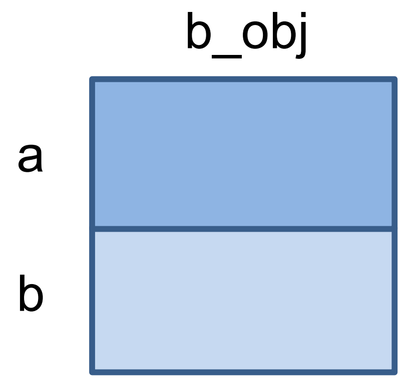
destruction

- 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

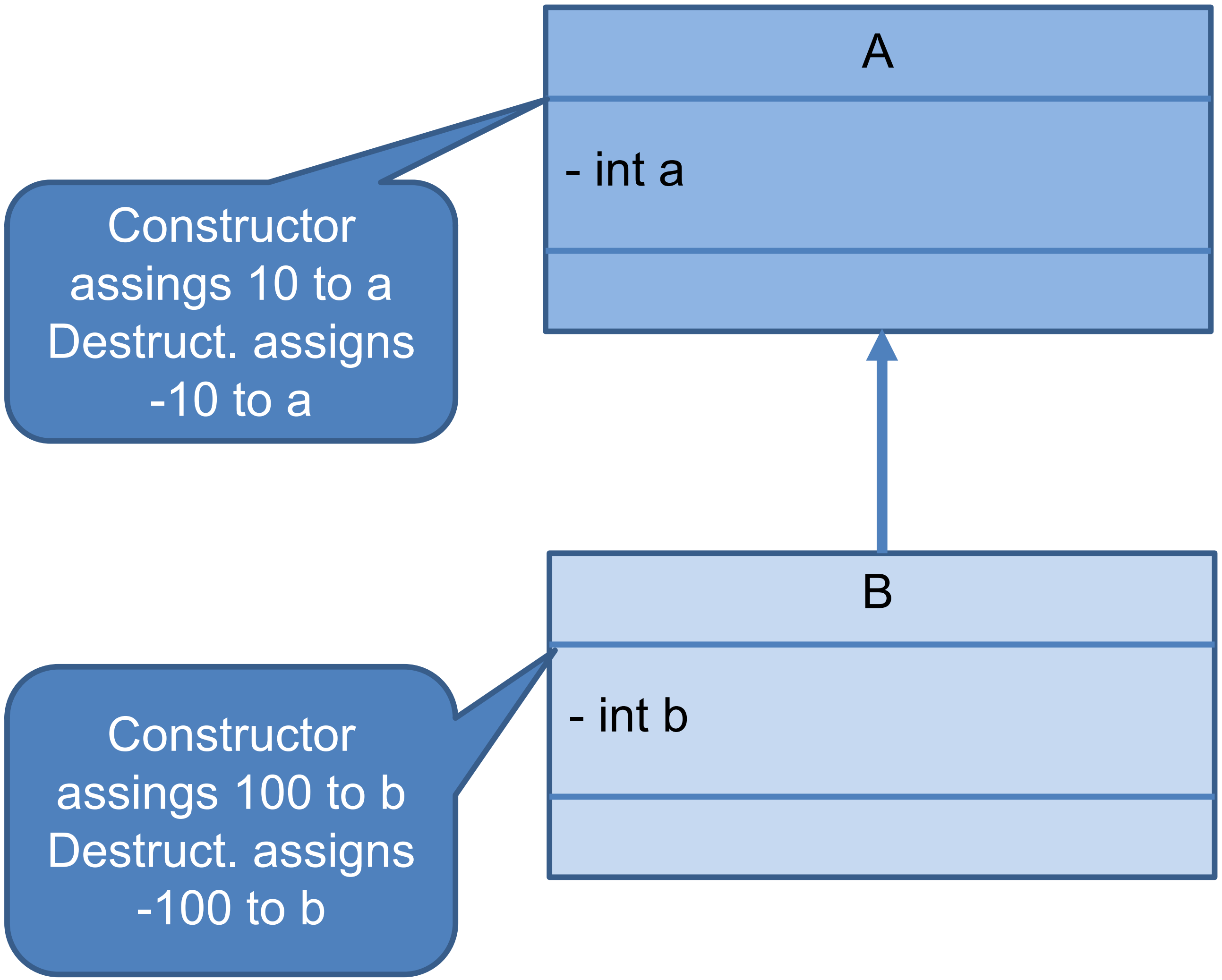
When a derived-class object is created & destroyed



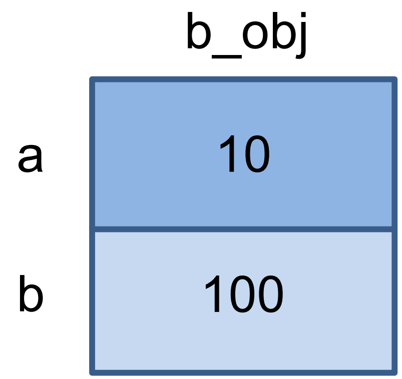

```
{
B b_obj;


// ~ B()!
}
```



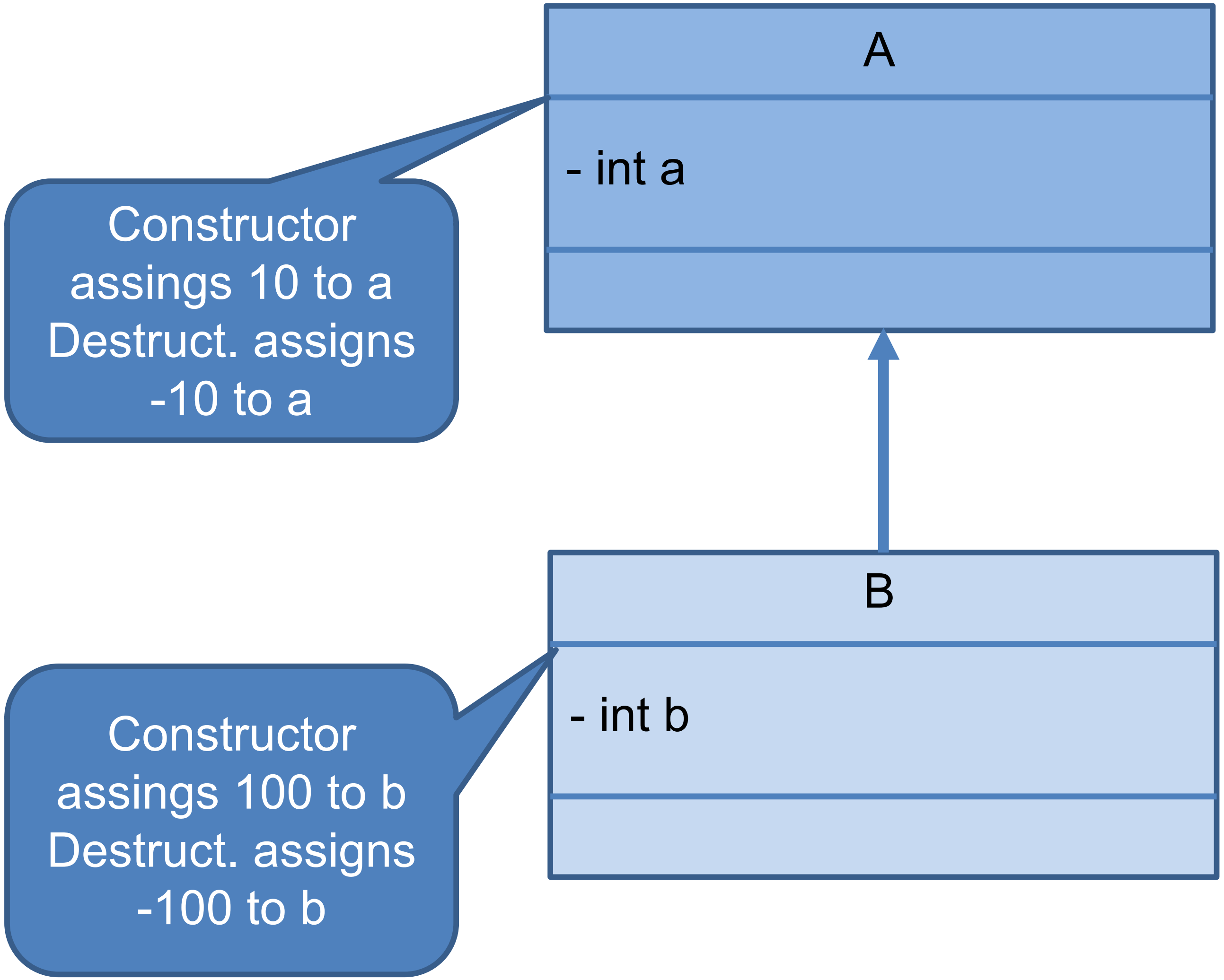
When a derived-class object is created & destroyed



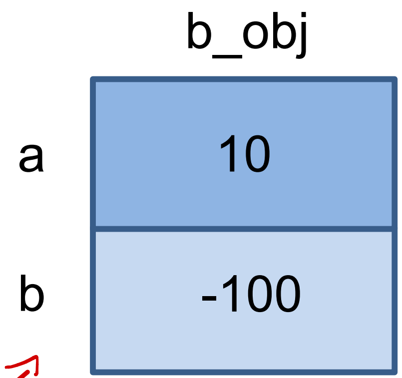

```
{
B b_obj;


// ~ B()!
}
```



When a derived-class object is created & destroyed

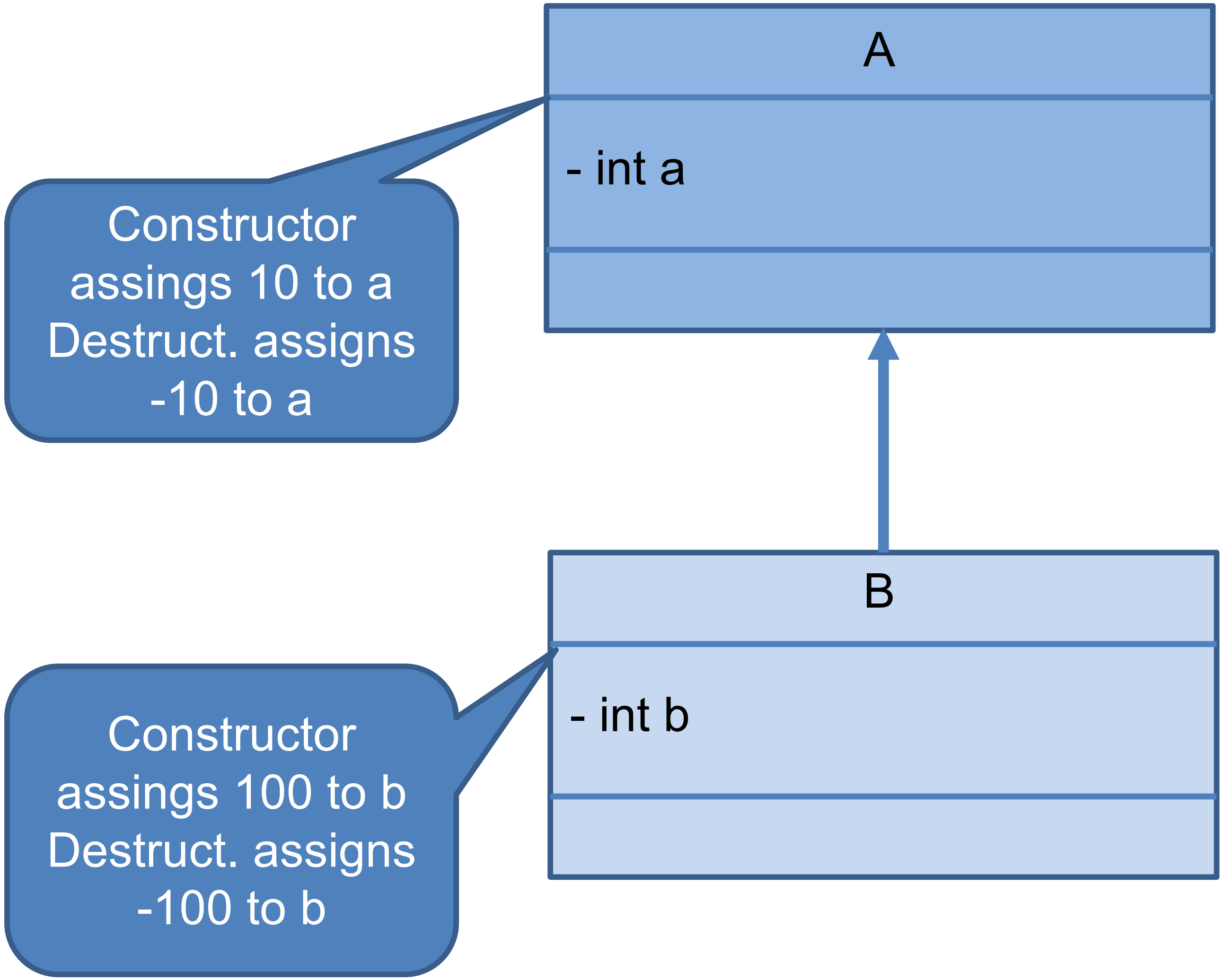



```
{
B b_obj;


// ~ B()!
}
```

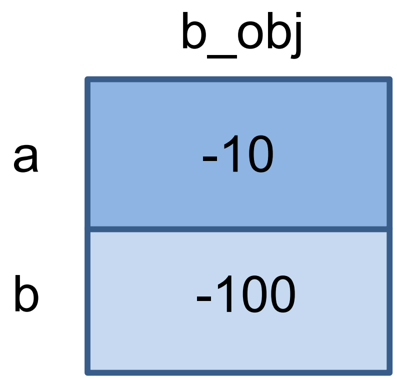


See the order: distructor first from the derived class.

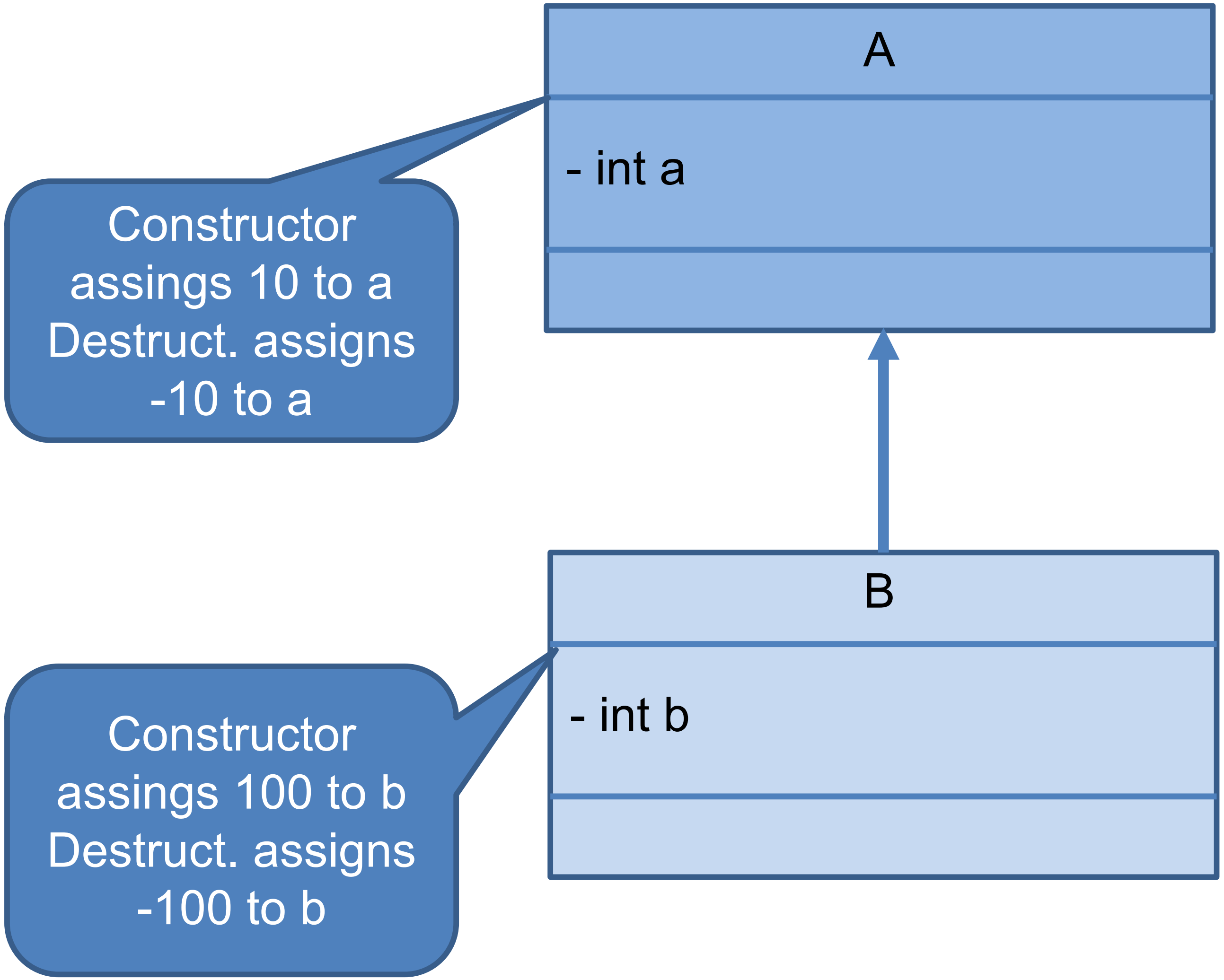
When a derived-class object is created & destroyed




```
{
B b_obj;


// ~ B()!
}
```



When a derived-class object is created & destroyed




```
{
B b_obj;


// ~ B()!
}
```

Constructors and Destructors in Base and Derived Classes

```
// This program demonstrates the order in which base and
// derived class constructors and destructors are called.
#include <iostream>
using namespace std;

//**********************************
// BaseClass declaration           *
//**********************************

```

Constructors and Destructors in Base and Derived Classes

```
class BaseClass
{
public:
   BaseClass()  // Constructor
      { cout << "This is the BaseClass constructor.\n"; }

   ~BaseClass() // Destructor
      { cout << "This is the BaseClass destructor.\n"; }
};

//********************************
// DerivedClass declaration      *
//********************************

class DerivedClass : public BaseClass
{
public:
   DerivedClass()  // Constructor
      { cout << "This is the DerivedClass constructor.\n"; }

   ~DerivedClass()  // Destructor
      { cout << "This is the DerivedClass destructor.\n"; }
};
```

Constructors and Destructors in Base and Derived Classes

```
//********************************
// main function                 *
//********************************

int main()
{
   cout << "We will now define a DerivedClass object.\n";

   DerivedClass object;

   cout << "The program is now going to end.\n";
   return 0;
}
```

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

Philosophical POV:

- In the absence of inheritance, we can think of a class as having two different kinds of developers: ordinary developers and implementors

- Ordinary developers:
  - write code that uses objects of the class type
  - such code can access only the **public** (interface) members of the class

- Implementers:
  - 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

link with "protected".

- Under inheritance, there is a **third kind of user**, namely, **derived classes programmers**

- A base class makes protected those parts of its implementation that it is willing to let its derived classes use

IMPORTANT !

- The protected members remain inaccessible to ordinary 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

VERY IMPORTANT: without polymorphism, a language is not even considered OO.

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**

very advanced topic.

SO WE WILL FOCUS ON THIS ONE

Overwriting methods

- A subclass can overwrite, i.e., change, a base class method behaviour

ex: constructors.

- Three mechanisms:
  - Overloading we already saw that: methods with same name but ≠ args.
  - 1) Redefinition
  - 2) Overriding

- Overriding provides polymorphism and is the **most powerful mechanism**

Redefining Base Class Functions

1) • 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

Base Class

EXAMPLE

```
class Quote {
public:
      Quote() = default;   // default constructor
      Quote(const string &book, double sales_price):
            bookNo(book), price(sales_price) { } // constructor
      string isbn() const { return bookNo; }
      // 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;   // so that no pb if a derived class
private:                            // has another destructor.
      string bookNo;  // ISBN number of this item
protected:
      double price = 0.0;  // normal, undiscounted price
};
```

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; // see implementation
                                                   // next
private:
      size_t min_qty = 0;// minimum purchase for the discount
                         // to apply
      double discount = 0.0; // fractional discount to apply
};
```

Derived Class

```
// if the specified number of items are purchased, use the discounted
// price
double Bulk_quote::net_price(size_t cnt) const
{   // different implementation!
        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) //item is a reference
{                                                //to the base class.
    // 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); // HERE.
    cout << "ISBN: " << item.isbn() // calls Quote::isbn
    << " # sold: " << n << " total due: " << ret << endl;
    return ret;
}
```

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

cf. DEMO.

(print_total // print_total2).

It's only @ run-time that we know the true type of the object we are running on.

It's only @ run time that we know where we have to jump (when we apply a certain method to an object of a certain type) when using a reference.

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
```

If we don't have this we would have to redefine exactly the same method for bulk_quote subclass

DEMO

Thx to dynamic binding!

see print_total2 ... I would need print_total3 exactly the same, for Bulk_quote!

→ but fortunately we don't need this!

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

This is fundamental

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** trough a **pointer** or a **reference** to the **base class object**

Fundamental !

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

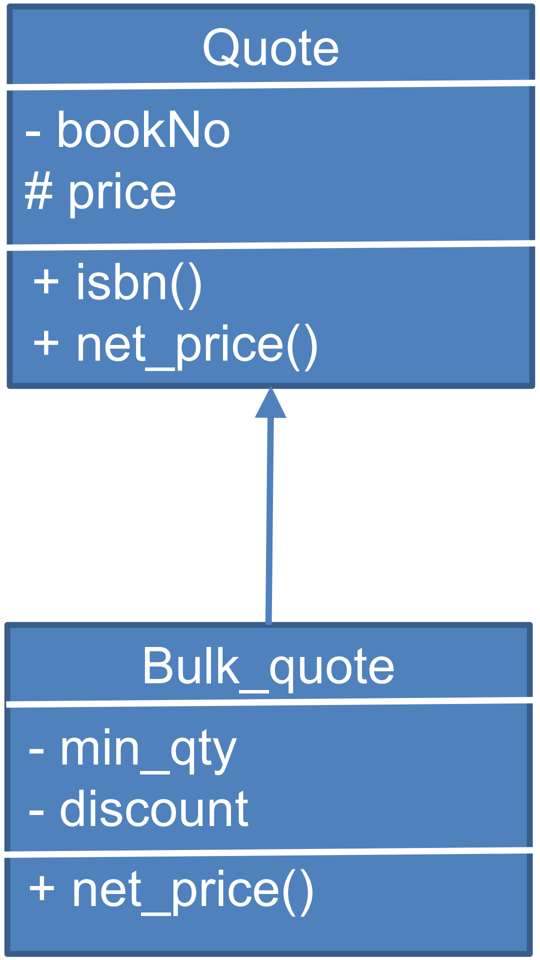
Derived-to-base Conversion

HARD TOPIC.

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

Bulk_quote object

Members inherited from *Quote*	bookNo price
Members defined by *Bulk_quote*	min_qty discount

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

- 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(const Quote &item, size_t n)` 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

Abstract Classes

Abstract Base Classes and Pure Virtual Functions

- **Pure virtual function**:
  - has no function definition in the base class (we don’t know how!)
  - must be overridden in a derived class that has objects

- Abstract base class contains at least one pure virtual function:

```
virtual void f() = 0;
```

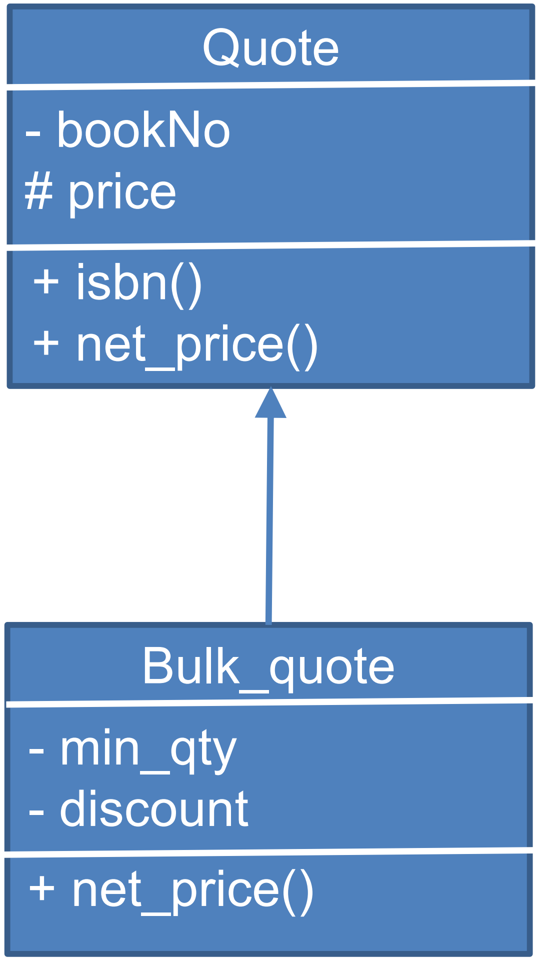
- The `= 0` indicates a pure virtual function

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

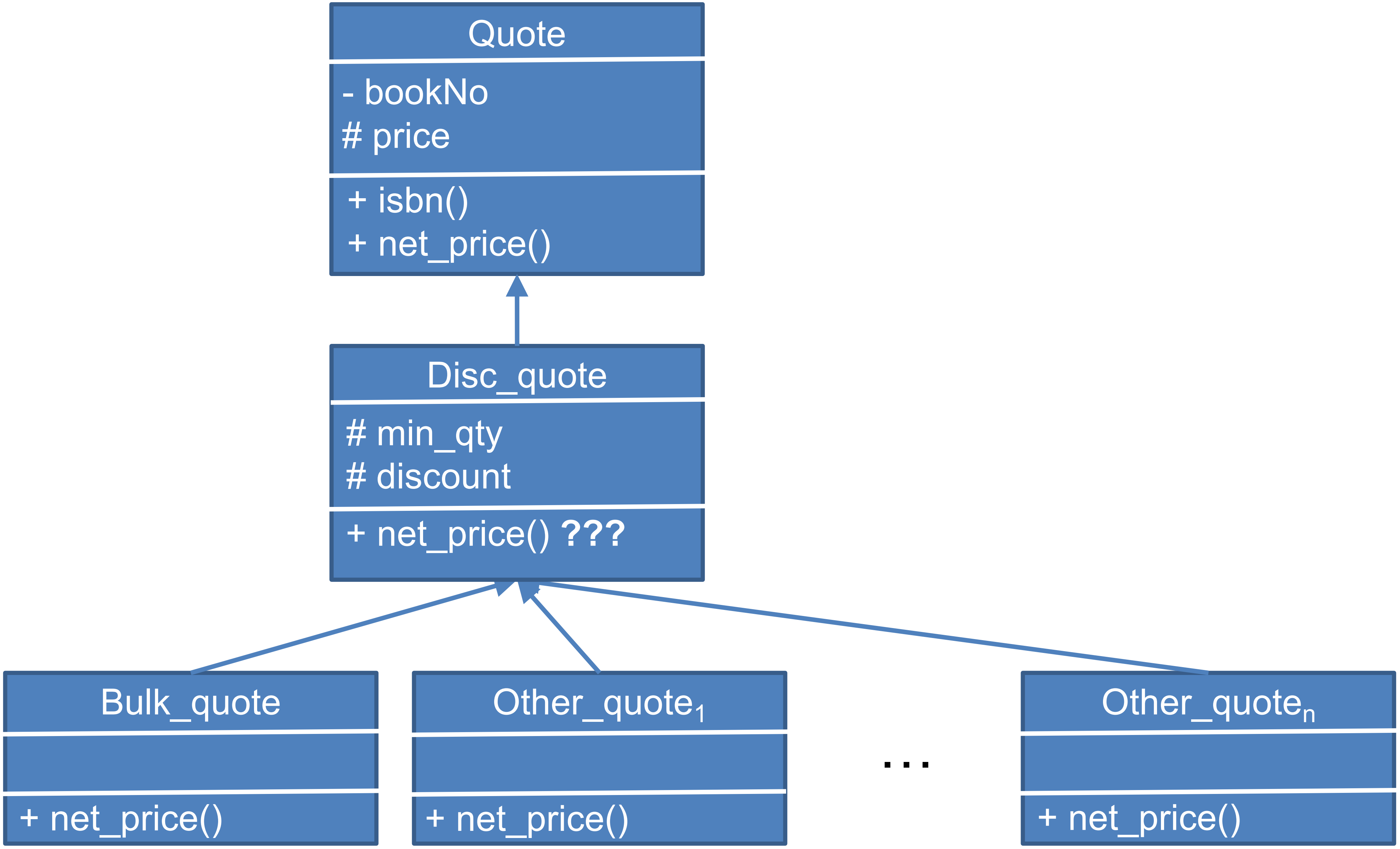
Abstract Base Classes and Pure Virtual Functions



Abstract Base Classes and Pure Virtual Functions

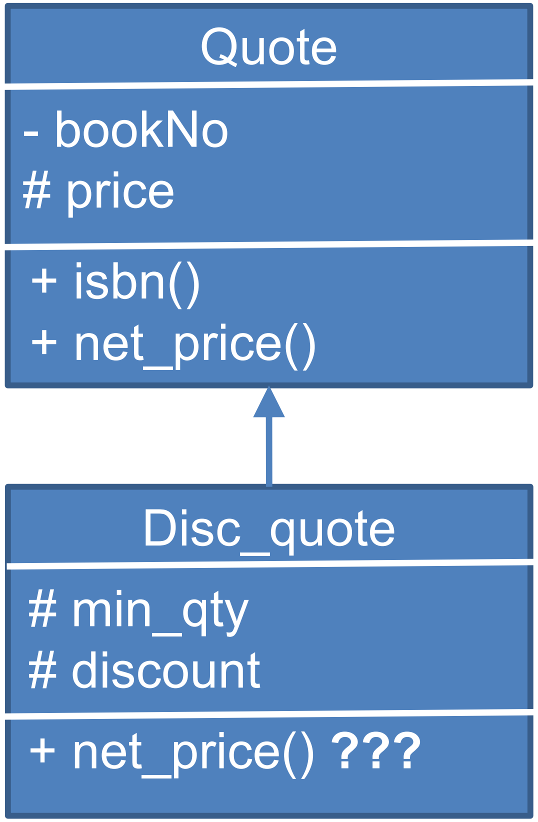
- Imagine that we want to extend our bookstore classes to support **several discount strategies**
  - **`Bulk_quote`**: discount for purchases above a certain limit but not for purchases up to that limit
  - In addition to a bulk discount, we might offer a discount for purchases up to a certain quantity 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



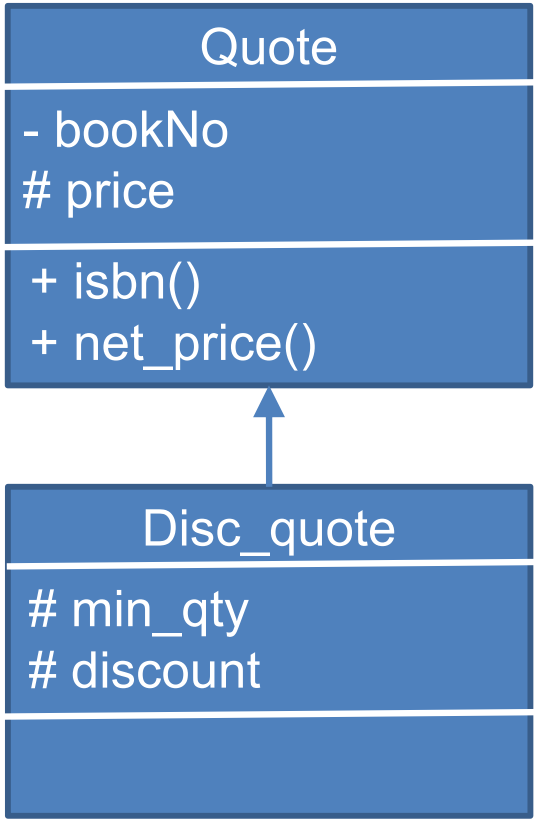
Abstract Base Classes and Pure Virtual Functions

- 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`

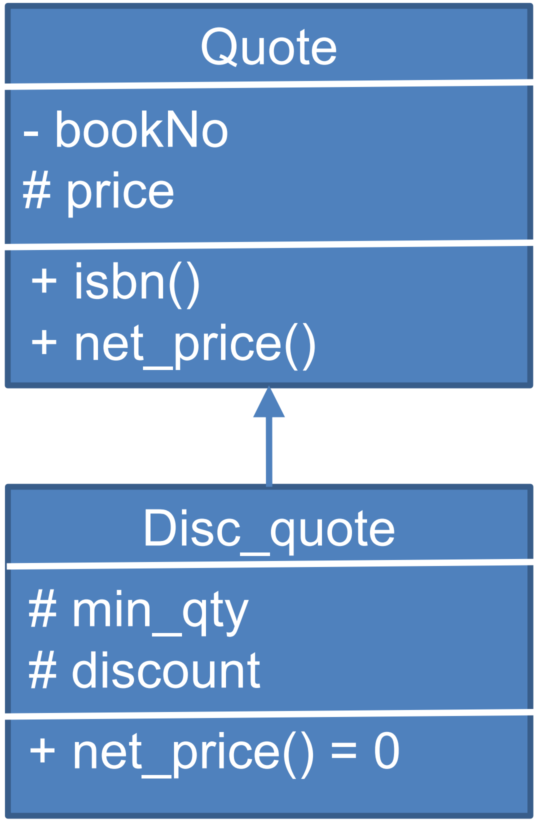


Abstract Base Classes and Pure Virtual Functions

- 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
};
```

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 strin
                       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
};
```

Used to construct the Disc_quote part of inheriting objects

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

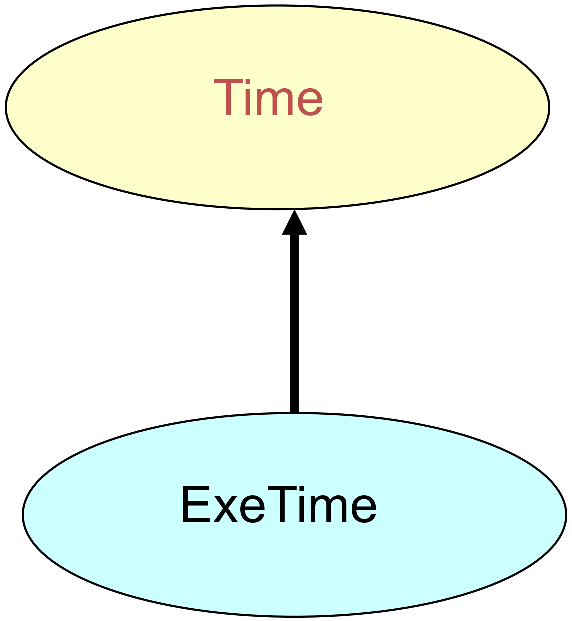
Take Home Message

- 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

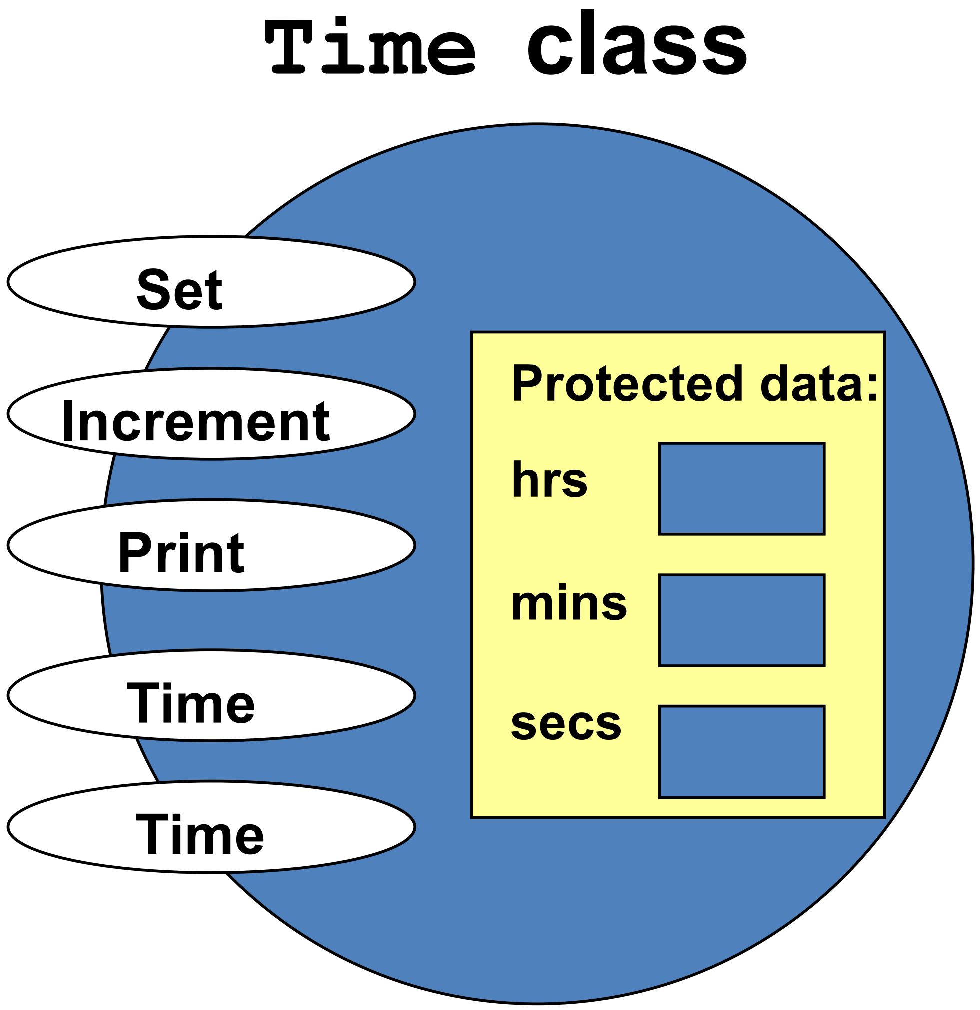


- Time is the base class
- ExeTime is the derived class with the notion of time zone

class `Time` Specification

```
// SPECIFICATION   FILE                                  ( time.h)

class  Time{

  public :

   void      Set ( int h, int m, int s ) ;
   void       Increment ( ) ;
   virtual void Print ( )  const ;
   Time     ( int initH, int initM, int initS ) ;   //  constructor
   Time     ( ) = default;                          //  default constructor

  protected :

   int            hrs = 0 ;
   int            mins = 0;
   int            secs = 0;
} ;
```

Class Interface Diagram

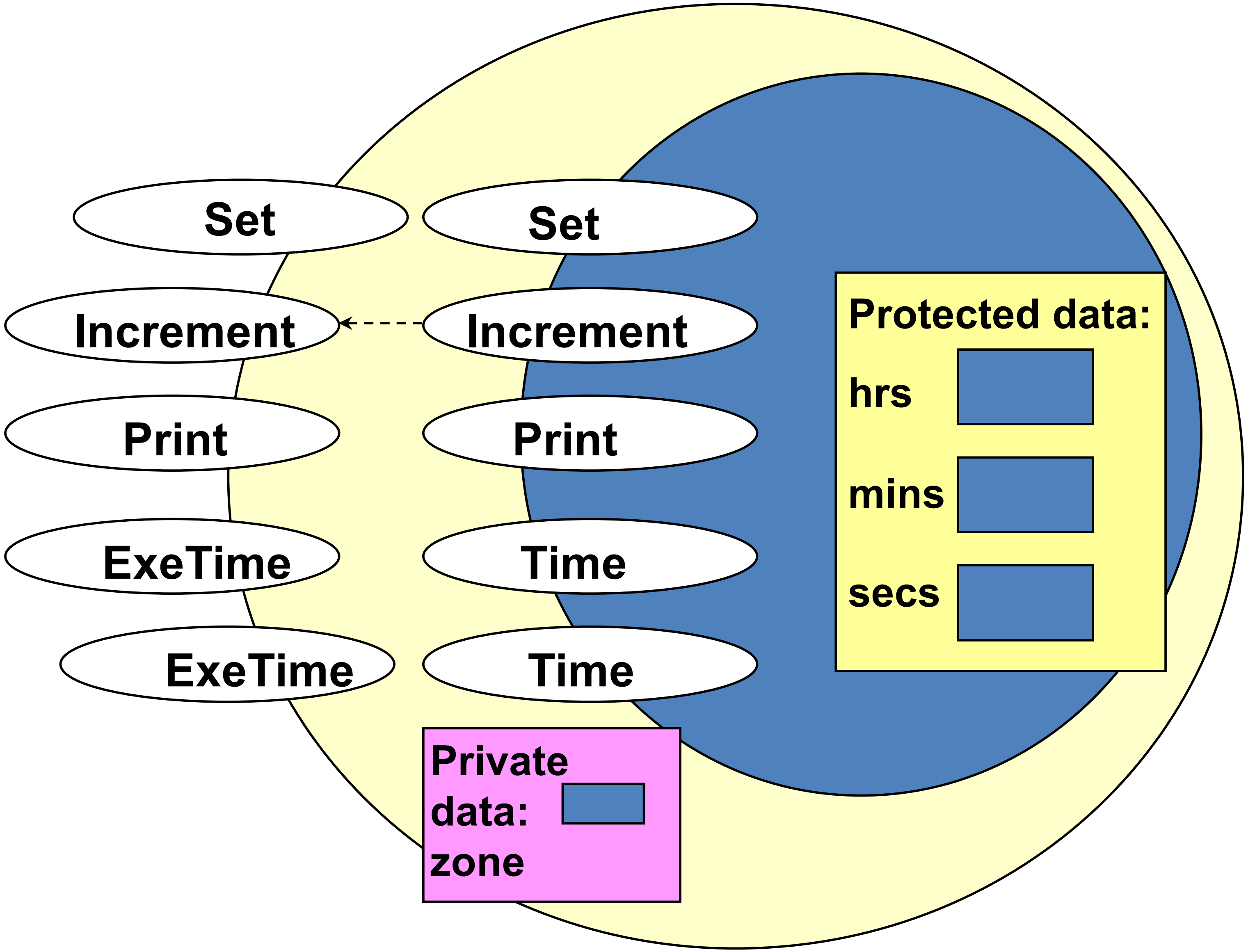


Derived Class `ExeTime`

```
// 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   Print ( )  const override;     //overridden
   ExeTime    (int initH, int initM, int initS, ZoneType initZone ) ;
   ExeTime();   // default constructor

private :
  ZoneType  zone ;       //  added data member
} ;
```

Class Interface Diagram

`ExeTime` class



Implementation of `ExeTime`

Default Constructor

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

```
ExeTime et1;
```

et1

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

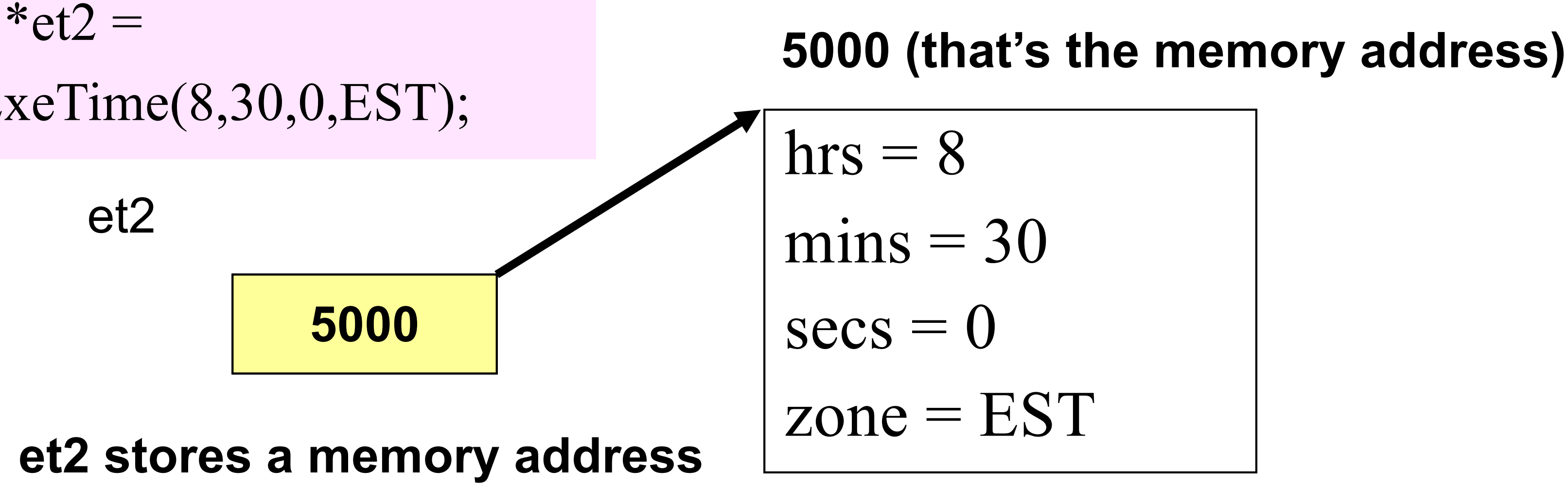
The default constructor of base class, Time(), is automatically called, when an ExtTime object is created.

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);
```



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 :: Print ( )   const  // function overriding
{
   string  zoneString[8] =
           {“EST”, “CST”, MST”, “PST”, “EDT”, “CDT”, “MDT”, “PDT”} ;

  Time :: Print ( ) ;
  cout  <<‘  ‘<<zoneString[zone]<<endl;
}
```

Working with **ExeTime**

```
#include "exetime.h"
… …
int main()
{
    ExtTime thisTime ( 8, 35, 0, PST ) ;
    ExtTime thatTime ;                  // default constructor called

    thatTime.Print( ) ;                 // outputs 00:00:00 EST

    thatTime.Set (16, 49, 23, CDT) ;
    thatTime.Print( ) ;                 // outputs 16:49:23 CDT

    thisTime.Increment ( ) ;
    thisTime.Increment ( ) ;
    thisTime.Print ( ) ;                // 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++