deep se
dependable evolvable pervasive software engineering group

Inheritance, final considerations

Danilo Ardagna

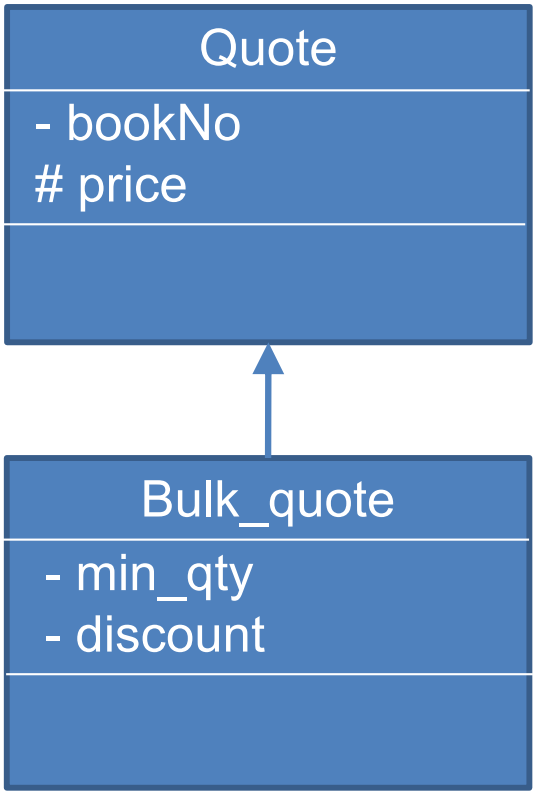
Politecnico di Milano
danilo.ardagna@polimi.it

14/11/2024

POLITECNICO DI MILANO

Derived-to-base Conversion

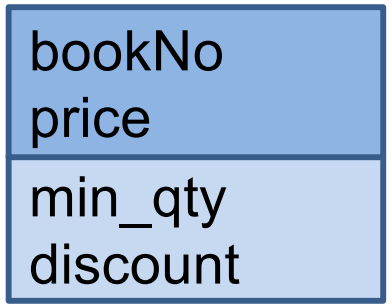
Derived-Class Objects and the Derived-to-Base Conversion



Members inherited from *Quote*

Members defined by *Bulk_quote*

Bulk_quote object

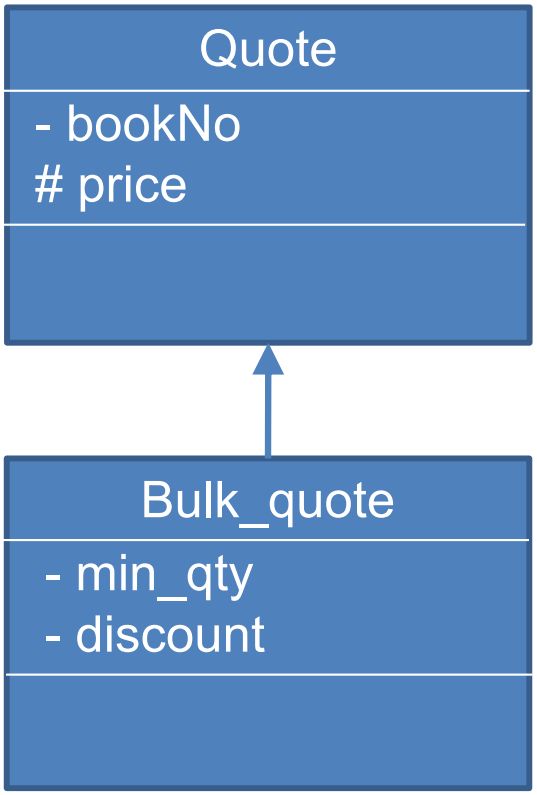


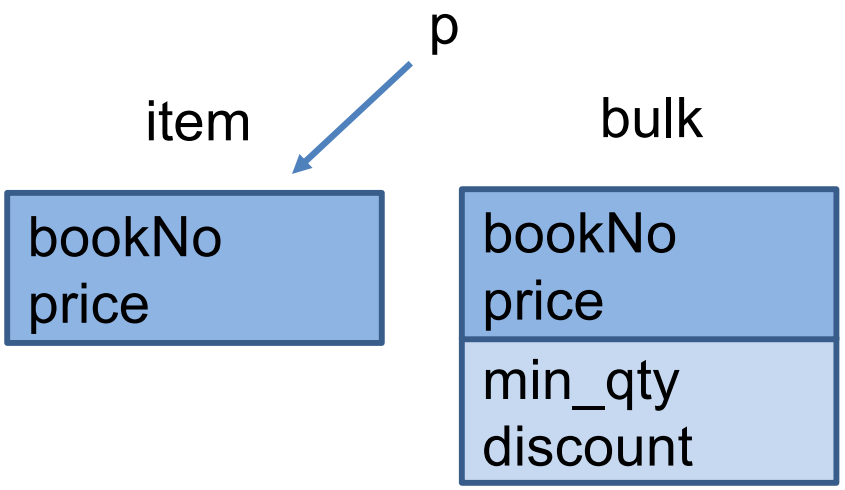
Derived-Class Objects and the **Derived-to-Base Conversion**

- 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

Derived-to-Base Conversion



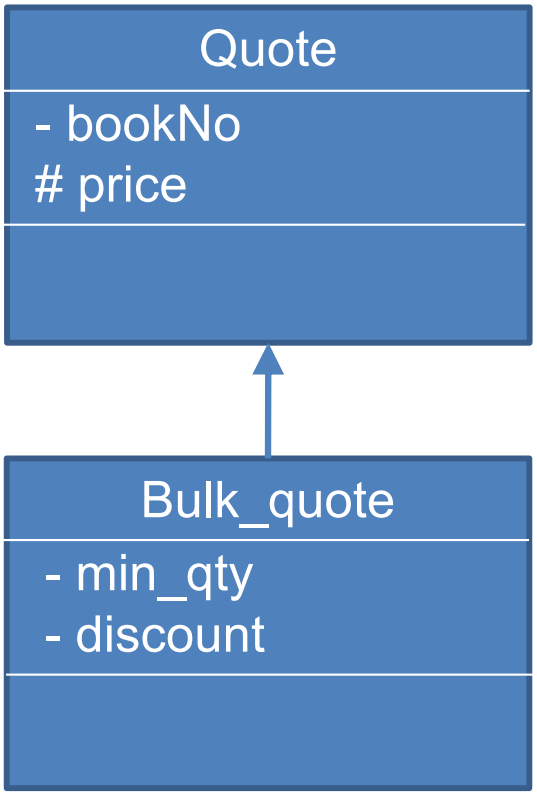


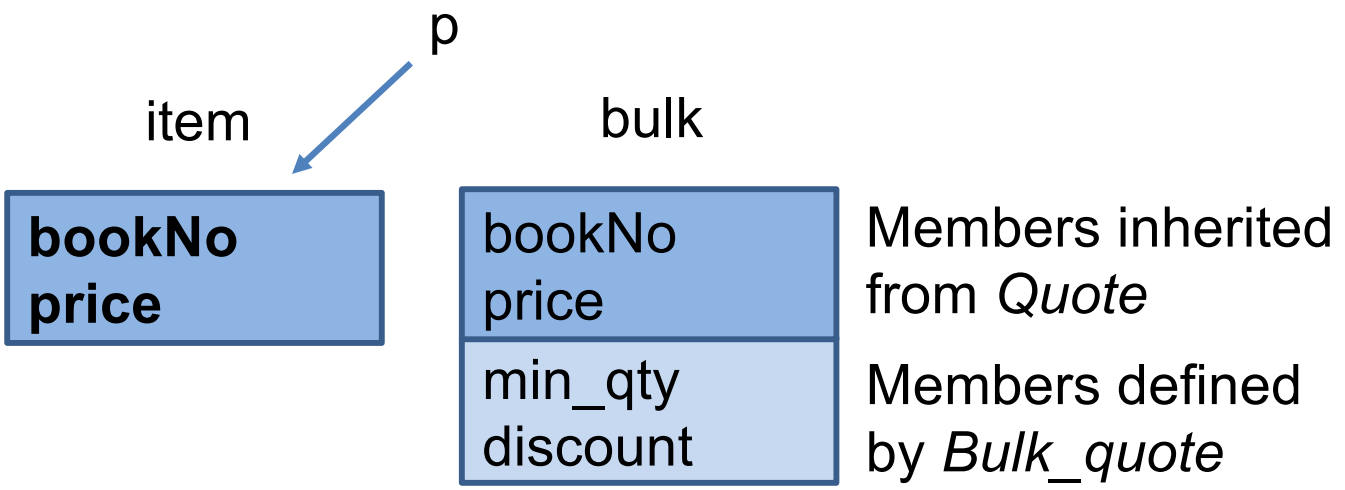
Members inherited from *Quote*

Members defined by *Bulk_quote*

```
Quote item;  // object of base type
Bulk_quote bulk;  // object of derived type

Quote *p = &item;  // p points to a quote object
```

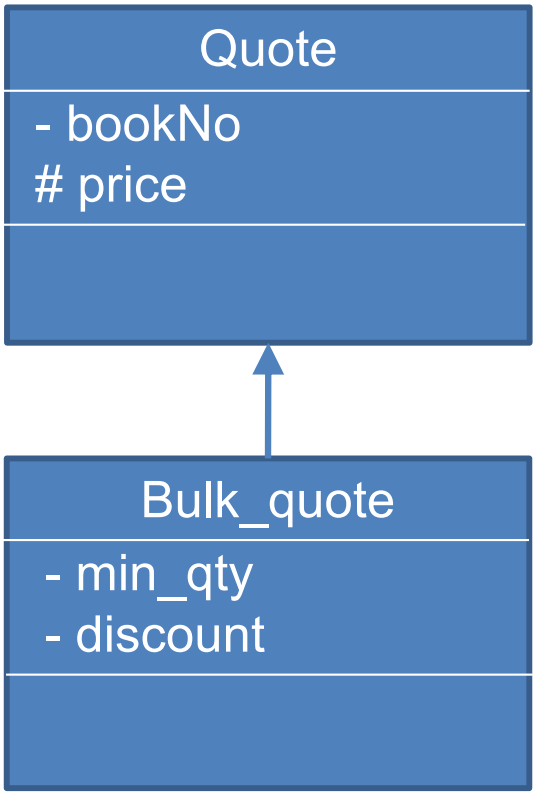
Derived-to-Base Conversion

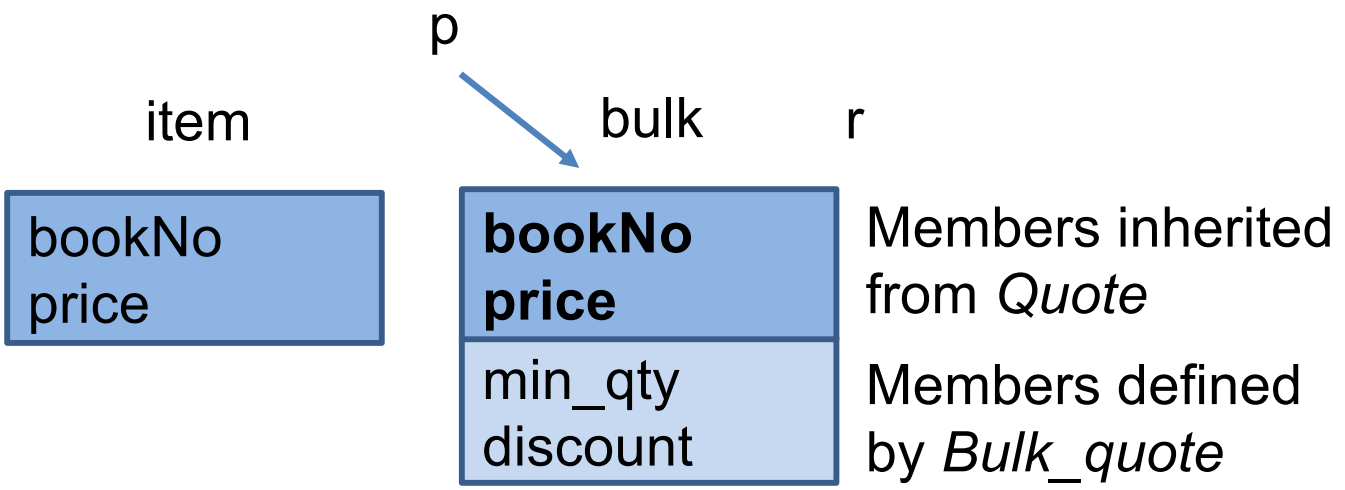





```
Quote item;  // object of base type
Bulk_quote bulk;  // object of derived type

Quote *p = &item;  // p points to a quote object
```

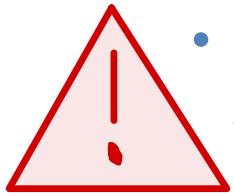
Derived-to-Base Conversion






```
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

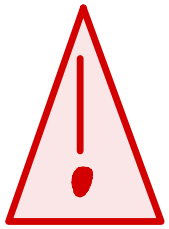


- Ordinarily, we can **bind a reference or a pointer** only to an object that has **the same type** as the corresponding reference or pointer

- 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

- **Static** and **dynamic type** of expressions including references or pointers

- **Polymorphism** and **dynamic binding**



No Implicit Conversion from Base to Derived

- The conversion from derived to base exists because **every derived object contains a base-class part** to which a **pointer or reference** of the base-class type can be bound

- There is **no similar guarantee for base-class objects**
  - A base-class object can exist either as an independent object or as part of a derived object
  - **A base object that is not part of a derived object** has **only** the members defined by the **base class**; it doesn't have the **members** defined by the derived class

- There is **no automatic conversion from** the **base** class to its **derived** class

No Implicit Conversion from Base to Derived

```
Quote base;
Bulk_quote* bulkP = &base; // error: can't convert base to
                           // derived
Bulk_quote& bulkRef = base; // error: can't convert base to
                            // derived
```

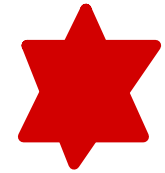
- We cannot convert from base to derived even when a base pointer or reference is bound to a derived object:

```
Bulk_quote bulk;
Quote *itemP = &bulk; // ok: dynamic type is Bulk_quote
Bulk_quote *bulkP = itemP; // error: can't convert base to derived
```

No Conversion between Objects

- The automatic **derived-to-base conversion** applies only for **conversions to a reference or pointer type**

- It is possible to **convert** an **object** of a **derived** class **to** its **base**-class type
  - Such conversions may not behave as we might want



No Conversion between Objects

- When we initialize or assign an object of a class type, we are actually calling a function
  - When we **initialize**, we're **calling a copy constructor**
  - When we **assign**, we're **calling an assignment operator**
  - These members normally have a parameter that is a reference to the const version of the class type

- Because these members take references, the **derived-to-base conversion lets us pass a derived object to** a **base-class copy operation**

No Conversion between Objects

- **These operations are not virtual!**
  - When we pass a derived object to a base-class constructor, **the constructor** that is run is **defined in the base class**
  - If we assign a derived object to a base object, the **assignment operator** that is run is the one **defined in the base class**

```
Bulk_quote bulk; // object of derived type
Quote item(bulk); // uses the Quote::Quote(const Quote&) constructor
item = bulk; // calls Quote::operator=(const Quote&)
```

Because the `Bulk_quote` part is ignored, we say that the `Bulk_quote` portion of bulk is **sliced down!**

18/11/2024

Containers and Inheritance

Containers and Inheritance

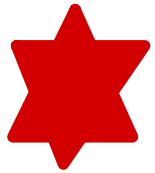
- When we use a container to store objects from an inheritance hierarchy, we generally must **store** those **objects indirectly**
  - We cannot put objects of types related by inheritance directly into a container, because there is no way to define a container that holds elements of differing types

Containers and Inheritance

- As an example, assume we want to define a vector to hold several books that a customer wants to buy
  1) We can't use a vector that holds `Bulk_quote` objects
     - ⚠ We can't convert `Quote` objects to `Bulk_quote`, so we wouldn't be able to put `Quote` objects into that vector
  2) We also can't use a vector that holds objects of type `Quote`
     - ✓ In this case, we can put `Bulk_quote` objects into the container
     - ⚠ However, those objects would no longer be `Bulk_quote` objects (slice down!)

But:

Containers and Inheritance



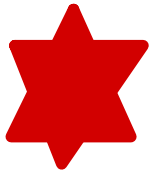
```
vector<Quote> basket;
basket.push_back(Quote("0-201-82470-1", 50));

// ok, but copies only the Quote part of the object into basket
basket.push_back(Bulk_quote("0-201-54848-8", 50, 10, .25));

// calls version defined by Quote, prints 750, i.e., 15 * $50
cout << basket[1].net_price(15) << endl;
```

sliced down:

no discount applied bc no pointer used.

- The elements in basket are `Quote` objects. When we add a `Bulk_quote` object to the vector its derived part is ignored

Containers and Inheritance



- Put (smart) pointers, not objects, in containers
  - When we need a container that holds objects related by inheritance, we typically define the container to hold pointers (preferably smart pointers) to the base class
  - The dynamic type of the object to which those pointers point might be the base-class type or a type derived from that base

Syntaxe :

```
vector<Quote*> basket;
basket.push_back(new Quote("0-201-82470-1", 50));
basket.push_back(new Bulk_quote("0-201-54848-8", 50, 10,
.25));
```

// calls the version defined by Bulk_quote*; prints 562.5, i.e., 15 * $50 less the*
// discount

```
cout << basket[1]->net_price(15) <<
```

Thanks to the use of pointers.

Here remind to delete objects before you exit!

BECAUSE WE USED RAW POINTERS.

Containers and Inheritance

- Put (smart) pointers, not objects, in containers
  - When we need a container that holds objects related by inheritance, we typically define the container to hold pointers (preferably smart pointers) to the base class
  - The dynamic type of the object to which those pointers point might be the base-class type or a type derived from that base

```
vector<shared_ptr<Quote>> basket;
basket.push_back(make_shared<Quote>("0-201-82470-1",
50));
basket.push_back(make_shared<Bulk_quote>("0-201-54848-
8", 50, 10, .25));

// calls the version defined by Bulk_quote; prints 562.5, i.e., 15 * $50 less the
// discount (once again fix to pointers)
cout << basket[1]->net_price(15) <<
```

Here you can forget to delete objects before you exit!

BECAUSE WE USED SMART POINTERS

Virtual Functions

Virtual Functions

- In C++ dynamic binding happens when a virtual member function is called through a reference or a pointer to a base-class type

Calls to Virtual Functions *May Be* Resolved at Run Time

- When a virtual function is called through a reference or pointer, the compiler generates code to decide at run time which function to call
  - The function that is called is the one that corresponds to the dynamic type of the object bound to that pointer or reference

```
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;
}
```

Calls to Virtual Functions *May Be* Resolved at Run Time

- When a virtual function is called through a reference or pointer, the compiler generates code to decide at run time which function to call
  - The function that is called is the one that corresponds to the dynamic type of the object bound to that pointer or reference

- In the case of `print_total`:
  - That function calls `net_price` on its parameter named item, which has type `Quote&`
  - Because item is a reference, and because `net_price` is virtual, the version of `net_price` that is called depends at run time on the actual (dynamic) type of the argument bound to item

Calls to Virtual Functions *May* Be Resolved at Run Time

```
Quote base("0-201-82470-1", 50);
print_total(base, 10);// calls Quote::net_price
Bulk_quote derived("0-201-82470-1", 50, 5, .19);
print_total(derived, 10);// calls Bulk_quote::net_price
```

Dynamic (run time) binding

```
base = derived; // copies the Quote part of derived into base
base.net_price(20); // calls Quote::net_price
```

Static (compile time) binding

Because "base" is from type Quote! We just copied the Quote part of "derived" into "base".

It does NOT make "base" of Bulk_quote type.

→ This can indeed be resolved at compile time.

Virtual Functions in a Derived Class return type

- **When a derived class overrides a virtual function**, it may, but **is not required to**, **repeat** the **virtual** keyword
  - Once a function is declared as virtual, it remains virtual in all the derived classes
- A derived-class function that overrides an inherited virtual function must have exactly the **same parameter type(s)** as the base-class function that it overrides
- With one exception, **the return type** of a virtual in the derived class also **must match** the return type of the function from the base class
  - The exception applies to virtuals that return a reference or pointer to types that are themselves related by inheritance
  - If `D` is derived from `B`, then a base class virtual can return a `B*` and the version in the derived can return a `D*`

Constructors and Destructors

Constructors and Destructors in Base and Derived Classes

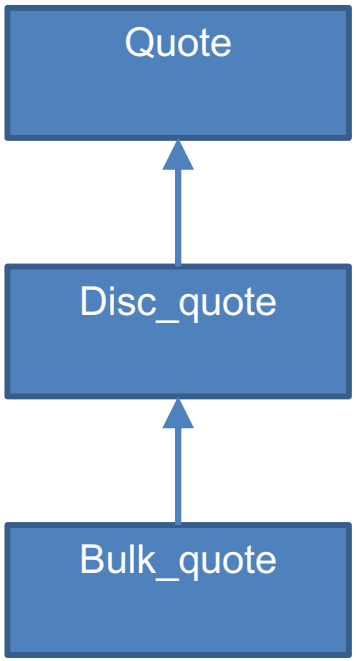
- Derived classes have their own constructors and destructors

- When an object of a derived class is created, the base class constructor is executed first, followed by the derived class constructor
  - A derived class must **use a base-class constructor** to initialize its base-class part

- When an object of a derived class is destroyed, its destructor is called first, then the one of the base class

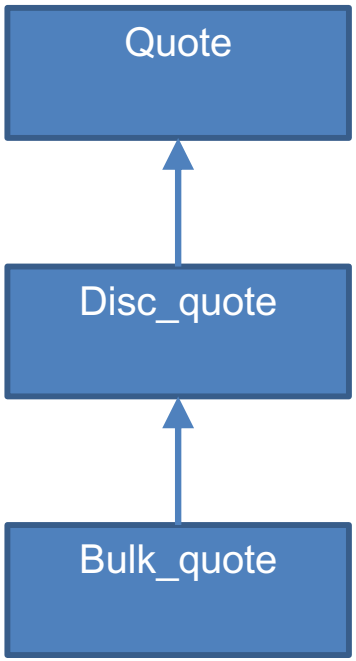
↳ opposite order.

A Derived Class Constructor initializes its direct base class only



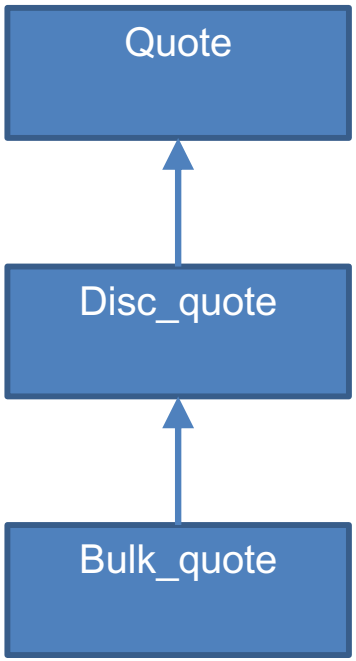

```
class Quote {
public:
        Quote() = default;
        Quote(const string &book, double
              sales_price):
             bookNo(book), price(sales_price) { }
        // as before
private:
        string bookNo; // ISBN number of this item
protected:
        double price = 0.0; // normal, undiscounted price
};
```

A Derived Class Constructor initializes its direct base class only




```
class Disc_quote : public Quote{
public:
   Disc_quote() = default;
   Disc_quote(const string& book, double sales_price,
              size_t qty, double disc):
         Quote(book, price), min_qty(qty),discount(disc) { }
        // as before
protected:
        size_t quantity = 0;
        double discount = 0.0;};
class Bulk_quote : public Disc_quote {
public:
   Bulk_quote() = default;
   Bulk_quote(const string& book, double sales_price,
              size_t qty, double disc):
         Disc_quote(book, sales_price, qty, disc){ }
        // as before
};
```

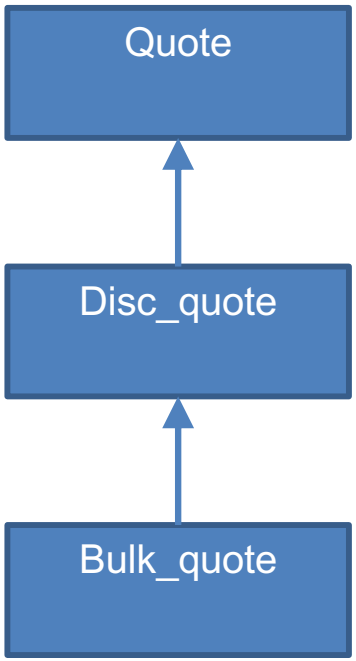
Constructors with parameters




```
Bulk_quote bulk("0-201", 50, 10, .25);
```

- `Bulk_quote` with 4 parameters constructor runs the `Disc_Quote` constructor with 4 parameters, which in turn runs the `Quote` constructor with 2 parameters
- The `Quote` constructor initializes the `bookNo` member to "0-201" and `price` to 50
- When the `Quote` constructor finishes, the `Disc_Quote` constructor continues, and initializes `min_qty` and `discount` with 10 and .25
- When the `Disc_quote` constructor finishes, the `Bulk_quote` constructor continues but has no other work to do

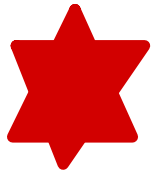
Synthesized Default constructors




```
Bulk_quote bulk;
```

- The synthesized `Bulk_quote` default constructor runs the `Disc_Quote` default constructor, which in turn runs the `Quote` default constructor
- The `Quote` default constructor default initializes the `bookNo` member to the empty string and uses the in-class initializer to initialize price to zero
- When the `Quote` constructor finishes, the `Disc_Quote` constructor continues, which uses the in-class initializers to initialize `min_qty` and `discount`
- When the `Disc_quote` constructor finishes, the `Bulk_quote` constructor continues but has no other work to do

Virtual Destructors



- A **base class** generally should define a **virtual destructor**

- Destructor needs to be virtual to allow objects in the inheritance hierarchy to be **dynamically allocated (through new or make_shared)**

- It's a good idea to make destructors virtual if the class could ever become a base class
  - Otherwise, the compiler will perform static binding on the destructor if the class ever is derived from

Classes vs. Structs & functions

C++ vs C: Who win?

OO goals

- Allow a **sw system** to **evolve without changes** to existing code

- Encapsulation:
  - if we don’t change the Class interface we can change internals and existing code continues to work properly
  - not enough to extend the system functionalities

- Inheritance and Polymorphism:
  - add new derived classes and methods override to extend functionalities

A case study

- CAD program

- Draw shapes on screen:
  - List of shapes and draw them according insertion order
  - Initial shapes: squares and circles
  - New needs: add new shapes and/or remove shapes

C version – Structs & functions

```
Enum ShapeType {circle, square};

struct Shape {
        ShapeType itsType;
};

struct Circle{
        ShapeType itsType;
        double itsRadius;
        Point itsCenter; // Point implemented somewhere else
};

struct Square{
        ShapeType itsType;
        double itsSide;
        Point itsTopLeft; // Point implemented somewhere else
};
```

C version – Structs & functions

```
// functions implemented somewhere else
void drawSquare(struct Square *);
void drawCircle(struct Circle *);

typedef struct Shape *ShapePointer;
```

C version – Structs & functions

```
void drawAllShapes(ShapePointer list[], int n){
for (int i=0; i<n; ++i){
      struct Shape* s = list[i];
      switch (s->itsType){
            case square:
                  drawSquare((struct Square *) s);
                  break;
            case circle:
                  drawCircle((struct Circle *) s);
                  break;
      }
}
```

To add new shapes we need to change this function implementation

C++ version – Classes

```
class Shape {
pubilc:
        virtual draw() const = 0;
};

class Circle: public Shape{
        double itsRadius;
        Point itsCenter; // Point implemented somewhere else
public:
        void draw() const override;

}

class Square: public Shape{
        double itsSide;
        Point itsTopLeft; // Point implemented somewhere else
public:
        void draw() const override;
}
```

C version – Structs & functions

```
void drawAllShapes(Shape* list[], int n){
for (int i=0; i<n; ++i)
     list[i] -> draw ();
}
```

To add new shapes you just need to add a new Class extending Shape overriding draw()

References

- Lippman Chapter 15