CSE 303
Lecture 23

Inheritance in C++

slides created by Marty Stepp
http://www.cs.washington.edu/303/
Represent a portfolio of a person's financial investments.

- Every asset has a *cost* (how much was paid for it) and a *market value* (how much it is currently worth).
  - The difference between these is the *profit*.
- Different assets compute their market value in different ways.

**Types of assets can be in a portfolio:**

- A **Stock** has a symbol (such as "MSFT" for Microsoft), a number of shares, the total cost paid, and a current price per share.
- A **Dividend Stock** is a stock that also gives back *dividend* payments.
- **Cash** is simply an amount of money. It never incurs profit or loss.
A possible design

- A class represents each type of asset.
  - *Problem*: Redundancy.
  - *Problem*: Cannot treat multiple investment types the same way (such as putting them into a portfolio array).
Inheritance

- **inheritance**: A parent-child relationship between classes.
  - a child (**derived** class) extends a parent (**base** class)

- **benefits of inheritance:**
  - **code reuse**: inherit code from superclass
  - **polymorphism**: Ability to redefine existing behaviors, so that when a client makes calls on different objects, it can have different results.
A better design (Java)

- an interface represents the top-level supertype (no code sharing)
- inheritance and subclassing gives us code sharing (DividendStock)
Access specifiers

<table>
<thead>
<tr>
<th>directory</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>visible to all other classes</td>
</tr>
<tr>
<td>private</td>
<td>visible only to the current class (even subclasses cannot directly access it)</td>
</tr>
<tr>
<td>protected</td>
<td>visible to current class and its subclasses</td>
</tr>
</tbody>
</table>

• declare a member as protected if:
  ▪ you don't want random clients accessing them, but
  ▪ you expect to be subclassed, and
  ▪ you don't mind for your subclasses to have access to it
Public inheritance

#include "BaseClass.h"

class Name : public BaseClass {
    ...
};

- inherits all behavior from the given base class (derived class must include base class's .h file)

- the following are not inherited:
  - constructors and destructors
  - the assignment operator = (if it was overridden)
#ifndef _DIVIDENDSTOCK_H
#define _DIVIDENDSTOCK_H

#include <string>
#include "Stock.h"

using namespace std;

// Represents a stock purchase that also pays dividends.
class DividendStock : public Stock {
  private:
    double m_dividends;   // amount of dividends paid

  public:
    DividendStock(string symbol, double sharePrice = 0.0);
    double dividends() const;
    double marketValue() const;
    void payDividend(double amountPerShare);
};

#endif
Inheritance and constructors

```
ClassName::ClassName(params) :
   BaseClassName(params) {
   statements;
}
```

• Constructors are not inherited
  ▪ but every time a subclass object is constructed, a constructor from the base class must be called (to initialize that part of the object)
  ▪ by default, calls the base's () constructor (if one exists)
```cpp
#include "DividendStock.h"

// Constructs a new stock with the given symbol and no shares.
DividendStock::DividendStock(string symbol, double sharePrice)
    : Stock(symbol, sharePrice) {
    m_dividends = 0.0;
}

// Returns this DividendStock's market value, which is
// a normal stock's market value plus any dividends.
double DividendStock::marketValue() const {
    return shares() * sharePrice() + dividends();
}

// Returns the total amount of dividends paid on this stock.
double DividendStock::dividends() const {
    return m_dividends;
}

// Records a dividend of the given amount per share.
void DividendStock::payDividend(double amountPerShare) {
    m_dividends += amountPerShare * shares();
}
```
A problem

Client program's old output:

<table>
<thead>
<tr>
<th>Value</th>
<th>Cost</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1234.56</td>
<td>$1234.56</td>
<td>$0.00</td>
</tr>
<tr>
<td>$475.00</td>
<td>$500.00</td>
<td>$-25.00</td>
</tr>
<tr>
<td>$3500.00</td>
<td>$2000.00</td>
<td>$1500.00</td>
</tr>
</tbody>
</table>

Client program's new output:

<table>
<thead>
<tr>
<th>Value</th>
<th>Cost</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1234.56</td>
<td>$1234.56</td>
<td>$0.00</td>
</tr>
<tr>
<td>$475.00</td>
<td>$500.00</td>
<td>$-25.00</td>
</tr>
<tr>
<td>$3500.00</td>
<td>$2000.00</td>
<td>$1000.00</td>
</tr>
</tbody>
</table>

• What happened?
Method dispatching

- **static dispatch**: Method calls are looked up at compile-time.
- **dynamic (virtual) dispatch**: Method calls looked up at runtime.

```cpp
// Stock.cpp
double Stock::profit() const {
    // Stock's version of marketValue / cost is used
    return marketValue() - cost();
}
```

- In Java, all objects' methods use *dynamic dispatch* automatically.
- In C++, methods use *static dispatch* by default.
  - If you override a method, superclass code won't notice the change.
  - *(This is considered a mistake in the design of C++.)*
Virtual methods

// Stock.h
class Stock {
    ...
    public:
        virtual double marketValue() const;
    ...
};

- If you want a method/operator to use dynamic dispatch, put the keyword virtual in its header (in the .h, not .cpp).

- **Rule of thumb**: Make all methods virtual if you expect subclassing.

- Destructors should also be virtual to avoid complex leak cases.
Virtual dispatch example

class A {
    public:
        void m1()          { cout << "a1" << endl; }
        virtual void m2() { cout << "a2" << endl; }
    
};

class B : public A {
    public:
        void m1()          { cout << "b1" << endl; }
        virtual void m2() { cout << "b2" << endl; }
    
};

int main() {
    A* var1 = new B();
    var1->m1(); // a1
    var1->m2(); // b2
    B* var2 = new B();
    var2->m1(); // b1
    var2->m2(); // b2
}
Override with redundancy

// Stock.cpp
double Stock::marketValue() const {
    return shares() * sharePrice();
}

// DividendStock.cpp
double DividendStock::marketValue() const {
    return shares() * sharePrice() + dividends();
}

- DividendStock's value is really the old value plus the dividends
- We'd like the code to reflect that relationship.
Calling a base class method

BaseClassName::methodName(parameters)

// DividendStock.cpp
double DividendStock::marketValue() const {
    return Stock::marketValue() + dividends();
}

- analogous to super. methodName() in Java
Virtual destructors

class B : public A { ... }

B* b = new B();
A* a = b;
delete a;

• Will the B::~B() destructor get called?
  ▪ Only if A::~A() was declared virtual.

• In what order will the destructors be called?
  ▪ ~B(), then ~A().

• Rule of thumb: Declare all destructors virtual.