C++ Inheritance I
CSE 333 Spring 2018

Instructor: Justin Hsia

Teaching Assistants:
Danny Allen    Dennis Shao    Eddie Huang
Kevin Bi       Jack Xu       Matthew Neldam
Michael Poulain Renshu Gu  Robby Marver
Waylon Huang   Wei Lin
Administrivia

- No exercise released today!
  - Next exercise on inheritance released on Wednesday

- hw3 is due in two Thursdays (5/17)
  - Get started early!
  - Section this week will also help you get started

- Midterm grading: scores released soon-ish?
  - Submit regrade requests via Gradescope for each subquestion
    - These go to different graders
  - Regrade requests open until end of Thursday (5/10)
  - Exam will be curved up (free points for everyone!)
Lecture Outline

- **C++ Inheritance**
  - Review of basic idea
  - Dynamic Dispatch
  - vtables and vptr

- Reference: *C++ Primer*, Chapter 15
Stock Portfolio Example

- A portfolio represents a person’s financial investments
  - Each *asset* has a cost (*i.e.* how much was paid for it) and a market value (*i.e.* how much it is worth)
    - The difference between the cost and market value is the *profit* (or loss)
  - Different assets compute market value in different ways
    - A *stock* that you own has a ticker symbol (*e.g.* “GOOG”), a number of shares, share price paid, and current share price
    - A *dividend stock* is a stock that also has dividend payments
    - *Cash* is an asset that never incurs a profit or loss

(Credit: thanks to Marty Stepp for this example)
Design Without Inheritance

- One class per asset type:

<table>
<thead>
<tr>
<th>Stock</th>
<th>DividendStock</th>
<th>Cash</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol_</td>
<td>symbol_</td>
<td>amount_</td>
</tr>
<tr>
<td>total_shares_</td>
<td>total_shares_</td>
<td></td>
</tr>
<tr>
<td>total_cost_</td>
<td>total_cost_</td>
<td>GetMarketValue()</td>
</tr>
<tr>
<td>current_price_</td>
<td>current_price_</td>
<td></td>
</tr>
<tr>
<td>GetMarketValue()</td>
<td>GetMarketValue()</td>
<td></td>
</tr>
<tr>
<td>GetProfit()</td>
<td>GetProfit()</td>
<td></td>
</tr>
<tr>
<td>GetCost()</td>
<td>GetCost()</td>
<td></td>
</tr>
<tr>
<td>dividends_</td>
<td></td>
<td>GetMarketValue()</td>
</tr>
</tbody>
</table>

- Redundant!
- Cannot treat multiple investments together
  - *e.g.* can't put in an array or vector of different assets

- Sample code for separate classes in `initial.tar`
Inheritance

- A parent-child “is-a” relationship between classes
  - A child (derived class) extends a parent (base class)

Benefits:
- Code reuse
  - Children can automatically inherit code from parents
- Polymorphism
  - Ability to redefine existing behavior but preserve the interface
  - Children can override the behavior of the parent
  - Others can make calls on objects without knowing which part of the inheritance tree it is in
- Extensibility
  - Children can add behavior
Design With Inheritance

```
Asset (abstract)

GetMarketValue()
  GetProfit()
  GetCost()

Stock
  symbol_
  total_shares_
  total_cost_
  current_price_

GetMarketValue()
  GetProfit()
  GetCost()

DividendStock
  symbol_
  total_shares_
  total_cost_
  current_price_
  dividends_

GetMarketValue()
  GetProfit()
  GetCost()

Cash
  amount_

GetMarketValue()
```
## Terminology

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superclass</td>
<td>Base Class</td>
</tr>
<tr>
<td>Subclass</td>
<td>Derived Class</td>
</tr>
</tbody>
</table>

- Mean the same things and you’ll hear both
Review: Access Modifiers

- **public**: visible to all other classes
- **protected**: visible to current class and its derived classes
- **private**: visible only to the current class
Derivation List

- Comma-separated list of classes to inherit from:

```cpp
#include "BaseClass.h"

class Name : public BaseClass {
    ...
};
```

- Focus on single inheritance, but multiple inheritance possible

- Almost always you will want public inheritance
  
  - Acts like extends does in Java
  
  - Any member that is non-private in the base class is the same in the derived class; **interface and implementation inheritance**
    - Except that constructors, destructors, copy constructor, and assignment operator are never inherited
# Back to Stocks

<table>
<thead>
<tr>
<th>Stock</th>
<th>DividendStock</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol_</td>
<td>symbol_</td>
</tr>
<tr>
<td>total_shares_</td>
<td>total_shares_</td>
</tr>
<tr>
<td>total_cost_</td>
<td>total_cost_</td>
</tr>
<tr>
<td>current_price_</td>
<td>current_price_</td>
</tr>
<tr>
<td>GetMarketValue()</td>
<td>dividends_</td>
</tr>
<tr>
<td>GetProfit()</td>
<td></td>
</tr>
<tr>
<td>GetCost()</td>
<td></td>
</tr>
</tbody>
</table>

**BASE**

**DERIVED**
Back to Stocks

- A derived class:
  - **Inherits** the behavior and state of the base class
  - **Overrides** some of the base class’ member functions
  - **Extends** the base class with new member functions, variables
Dynamic Dispatch

- Usually, when a derived function is available to an object, we want the derived function to be invoked
  - This requires a *run time* decision of what code to invoke
  - This is similar to Java

- A member function invoked on an object should be the *most-derived function* accessible to the object’s visible type
  - Can determine what to invoke from the *object* itself
Requesting Dynamic Dispatch

- Prefix the member function declaration with the `virtual` keyword
  - Derived/child functions don’t need to repeat `virtual`, but it traditionally has been good style to do so
  - This is how method calls work in Java (no virtual keyword needed)

- `override` keyword (C++11)
  - Tells compiler this method should be overriding an inherited virtual function – good to use if available
  - Prevents overloading vs. overriding bugs

- Both of these are *optional* in derived classes
  - Be consistent and follow local conventions
Dynamic Dispatch Example

- When a member function is invoked on an object:
  - The *most-derived function* accessible to the object’s visible type is invoked (decided at run time)

```cpp
double DividendStock::GetMarketValue() const {
    return get_shares() * get_share_price() + dividends_;}

double DividendStock::GetProfit() const {
    return GetMarketValue() - GetCost();
}

double Stock::GetMarketValue() const {
    return get_shares() * get_share_price();
}

double Stock::GetProfit() const {
    return GetMarketValue() - GetCost();
}
```

DividendStock.cc

Stock.cc
Dynamic Dispatch Example

```cpp
#include "Stock.h"
#include "DividendStock.h"

DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;   // why is this allowed?

// Invokes DividendStock::GetMarketValue()
ds->GetMarketValue();

// Invokes DividendStock::GetMarketValue()
s->GetMarketValue();

// invokes Stock::GetProfit(), since that method is inherited.
// Stock::GetProfit() invokes DividendStock::GetMarketValue(),
// since that is the most-derived accessible function.
s->GetProfit();
```
class A {
    public:
        // Foo will use dynamic dispatch
        virtual void Foo();
};

class B : public A {
    public:
        // B::Foo overrides A::Foo
        virtual void Foo();
};

class C : public B {
    // C inherits B::Foo()
};

void Bar() {
    A* a_ptr;
    C c;
    a_ptr = &c;
    // Whose Foo() is called?
    a_ptr->Foo();
}
Peer Instruction Question

- Whose `Foo()` is called?

Q1 | Q2
---|---
A. | A | B
B. | A | D
C. | B | B
D. | B | D
E. | We’re lost…

```cpp
void Bar() {
    A* a_ptr;
    C c;
    E e;

    // Q1:
    a_ptr = &c;
    a_ptr->Foo();

    // Q2:
    a_ptr = &e;
    a_ptr->Foo();
}
```
How Can This Possibly Work?

- The compiler produces `Stock.o` from `just Stock.cc`
  - It doesn’t know that `DividendStock` exists during this process
  - So then how does the emitted code know to invoke `Stock::GetMarketValue()` or `DividendStock::GetMarketValue()`?

  - **Function pointers**

```cpp
Stock.h

virtual double Stock::GetMarketValue() const;
virtual double Stock::GetProfit() const;

Stock.cc

double Stock::GetMarketValue() const {
    return get_shares() * get_share_price();
}

double Stock::GetProfit() const {
    return GetMarketValue() - GetCost();
}
```
vtables and the vptr

- If a class contains *any* virtual methods, the compiler emits:
  - A virtual function table *(vtable)* for *the class*
    - Contains a function pointer for each virtual method in the class
    - The pointers in the vtable point to the most-derived function for that class
  - A virtual table pointer *(vptr)* for *each object instance*
    - A pointer to a virtual table as a “hidden” member variable
    - When the object’s constructor is invoked, the vptr is initialized to point to the vtable for the object’s class
    - Thus, the vptr “remembers” what class the object is
351 Throwback: Dynamic Dispatch

Point object

```
header vtable ptr x y
```

Point vtable:

```
p -> ???
```

code for Point’s samePlace()

3DPoint object

```
header vtable x y z
```

3DPoint vtable:

```
```

code for sayHi()

Java:

```
Point p = ???;
return p.samePlace(q);
```

C pseudo-translation:

```
// works regardless of what p is
return p->vtable[1](p, q);
```
vtable/vptr Example

class Base {
    public:
        virtual void f1();
        virtual void f2();
};

class Der1 : public Base {
    public:
        virtual void f1();
};

class Der2 : public Base {
    public:
        virtual void f2();
};

Base b;
Der1 d1;
Der2 d2;

Base* b0ptr = &b;
Base* b1ptr = &d1;
Base* b2ptr = &d2;

b0ptr->f1(); //
b0ptr->f2(); //

b1ptr->f1(); //
b1ptr->f2(); //

d2.f1(); //
b2ptr->f1(); //
b2ptr->f2(); //
vtable/vptr Example

object instances

Base
f1()
f2()

Der1
f1()
f2()

Der2
f1()
f2()

class vtables

Base
vptr

Der1
vptr

Der2
vptr

compiled code

Base::f1()
push %rbp...

Base::f2()
push %rbp...

Der1::f1()
push %rbp...

Der2::f2()
push %rbp...

Base b;
Der1 d1;
Der2 d2;

Base* b2ptr = &d2;

d2.f1();
// d2.vptr -->
// Der2.vtable.f1 -->
// Base::f1()

b2ptr->f1();
// b2ptr -->
// d2.vptr -->
// Der2.vtable.f1 -->
// Base::f1()
Let’s Look at Some Actual Code

- Let’s examine the following code using `objdump`
  - `g++ -g -o vtable vtable.cc`
  - `objdump -CDS vtable > vtable.d`

```cpp
class Base {
public:
    virtual void f1();
    virtual void f2();
};

class Der1 : public Base {
public:
    virtual void f1();
};

int main(int argc, char** argv) {
    Der1 d1;
    d1.f1();
    Base* bptr = &d1;
    bptr->f1();
}
```