C++ Inheritance I
CSE 333 Spring 2019

Instructor: Justin Hsia

Teaching Assistants:
Aaron Johnston       Andrew Hu       Daniel Snitkovskiy
Forrest Timour      Kevin Bi        Kory Watson
Pat Kosakanchit     Renshu Gu       Tarkan Al-Kazily
Travis McGaha
Administrivia

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

- hw3 is due in two Thursdays (5/23)
  - Get started early!
  - Section this week will also help you get started

- Midterm grading: scores released
  - Exam and sample solution posted on website
  - Submit regrade requests via Gradescope for each subquestion
    - These go to different graders
  - Regrade requests open until end of Thursday (5/16)
Overview of Next Two Lectures

❖ C++ inheritance
  ▪ **Review of basic idea** (pretty much the same as in Java)
  ▪ What’s different in C++ (compared to Java)
    • Static vs. dynamic dispatch – virtual functions and vtables (optional)
    • Pure virtual functions, abstract classes, why no Java “interfaces”
    • Assignment slicing, using class hierarchies with STL
  ▪ Casts in C++

❖ 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></td>
</tr>
<tr>
<td>total_shares_</td>
<td>total_shares_</td>
<td></td>
</tr>
<tr>
<td>total_cost_</td>
<td>total_cost_</td>
<td></td>
</tr>
<tr>
<td>current_price_</td>
<td>current_price_</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dividends_</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>
</tbody>
</table>

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

- See sample code in *initial.tar*
Inheritance

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

- Terminology:
  - Mean the same things. You’ll hear both.

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superclass</td>
<td>Base Class</td>
<td></td>
</tr>
<tr>
<td>Subclass</td>
<td>Derived Class</td>
<td></td>
</tr>
</tbody>
</table>
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()
Like Java: Access Modifiers

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

Use **protected** for class members only when

- Class is designed to be extended by subclasses
- Subclasses must have access but clients should not be allowed
Class Derivation List

- Comma-separated list of classes to inherit from:

  ```
  #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; both **interface and implementation inheritance**
  - Except that constructors, destructors, copy constructor, and assignment operator are *never* inherited
# Back to Stocks

**Stock**

<table>
<thead>
<tr>
<th>symbol_</th>
<th>total_shares_</th>
<th>total_cost_</th>
<th>current_price_</th>
</tr>
</thead>
</table>

- GetMarketValue()
- GetProfit()
- GetCost()

**DividendStock**

<table>
<thead>
<tr>
<th>symbol_</th>
<th>total_shares_</th>
<th>total_cost_</th>
<th>current_price_</th>
<th>dividends_</th>
</tr>
</thead>
</table>

- GetMarketValue()
- GetProfit()
- GetCost()
Polymorphism in C++

- In Java: `PromisedType var = new ActualType();`
  - `var` is a reference (different term than C++ reference) to an object of `ActualType` on the Heap
  - `ActualType` must be the same class or a subclass of `PromisedType`

- In C++: `PromisedType *var_p = new ActualType();`
  - `var_p` is a `pointer` to an object of `ActualType` on the Heap
  - `ActualType` must be the same or a derived class of `PromisedType`
  - (also works with references)
  - `PromisedType` defines the *interface* (*i.e.* what can be called on `var_p`), but `ActualType` may determine which *version* gets invoked
A derived class:
- **Inherits** the behavior and state (specification) of the base class
- **Overrides** some of the base class’ member functions (opt.)
- **Extends** the base class with new member functions, variables (opt.)
Dynamic Dispatch (like Java)

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

- 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

**Example:**
- void **PrintStock**(*Stock* *s*) { *s-*>**Print**(); }
- Calls the appropriate **Print()** without knowing the actual type of *s*, other than it is some sort of *Stock*
Requesting Dynamic Dispatch (C++)

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

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

- Both of these are technically `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** based on actual type of the object)

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

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

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

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

---

```
should invoke DividendStock::GetMarketValue()  
```

---

```
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();
```

- `dynamic_cast` is used for type conversion in C++ to check if an object is an instance of a particular type.
- The `GetMarketValue()` and `GetProfit()` methods are used to demonstrate dynamic dispatch.
- The code snippet shows how dynamic dispatch works with inheritance.
- The `dividend` object is of type `DividendStock` but accessed as a `Stock` pointer, allowing for polymorphism.

The output explains the behavior of dynamic dispatch in the context of inheritance, emphasizing the flexibility and power of C++ polymorphism.
Most-Derived

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(); //B::Foo()
}
Practice 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();
    B::Foo();

    // Q2:
    a_ptr = &e;
    a_ptr->Foo();
    B::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 call
    `Stock::GetMarketValue()` or
    `DividendStock::GetMarketValue()`
    or something else that might not exist yet?
    - *Function pointers!!!*

```cpp
Stock.h

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

```cpp
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 (single) virtual function table (vtable) for the class (1 per 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 (1 per object)
    - 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

**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(); // Base::f1()
b0ptr->f2(); // Base::f2()

b1ptr->f1(); // Der1::f1()
b1ptr->f2(); // Base::f2()

\{ d2.f1(); // Base::f1()
b2ptr->f1(); // Base::f1()
b2ptr->f2(); // Der2::f2() \}
vtable/vptr Example

Object instances

```
Base b;
Der1 d1;
Der2 d2;
```

Class vtables

```
Base
f1()  f2()

Der1
f1()  f2()

Der2
f1()  f2()
```

Compiled code

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

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

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

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

```cpp
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++ -Wall -g -std=c++11 -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(); // done via hard-coded call
    Base* bptr = &d1;
    bptr->f1(); // done via indirect jump on // vtable entry
}
```
More to Come Next Time!

- Any lingering questions?