CSE 374 - Week 9 (Mon)
C++ Inheritance

Instructor: Andrew Hu

TAs: Jim Limprasert, Kaelin Laundry, Keith Jiang, Nick Durand, Simon McFarlane, Yitian Hao
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:

  - **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()

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

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

- Terminology:

<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>

- Mean the same things. You’ll hear both.
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 derived classes
- Derived classes must have access but clients should not be allowed
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; both interface and implementation inheritance
    - Except that constructors, destructors, copy constructor, and assignment operator are never inherited
Back to Stocks

<table>
<thead>
<tr>
<th>Base Stock</th>
<th>Derived 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>GetMarketValue()</td>
</tr>
<tr>
<td>GetProfit()</td>
<td>GetProfit()</td>
</tr>
<tr>
<td>GetCost()</td>
<td>GetCost()</td>
</tr>
<tr>
<td>dividends_</td>
<td>dividends_</td>
</tr>
</tbody>
</table>
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
Back to Stocks

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:**
  - ```cpp
      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
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();
}
```

DividendStock.cc

Stock.cc
#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();
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();
}
Poll Question: PollEv.com/andrewhu
Poll Question ([PollEv.com/andrewhu](https://PollEv.com/andrewhu))

- Whose `Foo()` is called?

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>D</td>
</tr>
</tbody>
</table>

```cpp
class A {
    public:
    virtual void Foo();
};

class B : public A {
    public:
    virtual void Foo();
};
class C : public B {
};
class D : public C {
    public:
    virtual void Foo();
};
class E : public C {
};
```
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?

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

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

Stock.h

Stock.cc
vtables and the vptr

- If a class contains *any* virtual methods, the compiler emits:
  - A (single) 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
vptr and vtable Visualization

```
vptr
```

```
vtable
```

```
Implementation
```

Cash::GetMarketValue(
Cash::GetCost()
Cash::GetProfit()

Stock::GetMarketValue(
Stock::GetCost()
Stock::GetProfit()

DividendStock::GetMarketValue()
Reminder: *virtual* is “sticky”

- If `X::f()` is declared virtual, then a vtable will be created for class `X` and for *all* of its subclasses
  - The vtables will include function pointers for (the correct) `f`

- `f()` will be called using dynamic dispatch even if overridden in a derived class without the *virtual* keyword
What happens if we omit “virtual”?

- By default, without `virtual`, methods are dispatched **statically**
  - At compile time, the compiler writes in a `call` to the address of the class’ method in the `.text` segment
    - Based on the compile-time visible type of the callee
  - This is **different** than Java

```cpp
class Derived : public Base { ... };

int main(int argc, char** argv) {
    Derived d;
    Derived* dp = &d;
    Base* bp = &d;
    dp->foo();
    bp->foo();
    return EXIT_SUCCESS;
}
```
Static Dispatch Example

- Removed `virtual` on methods:

```cpp
#include <Stock.h>

DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;

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

// Invokes `Stock::GetMarketValue()`
s->GetMarketValue();

// invokes `Stock::GetProfit()`.
// `Stock::GetProfit()` invokes `Stock::GetMarketValue()`.

s->GetProfit();

// invokes `Stock::GetProfit()`, since that method is inherited.
// `Stock::GetProfit()` invokes `Stock::GetMarketValue()`.

ds->GetProfit();
```

```cpp
Stock.h

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

```cpp
// invokes `Stock::GetProfit()`, since that method is inherited.
// `Stock::GetProfit()` invokes `Stock::GetMarketValue()`.
```
Why Not Always Use `virtual`?

- Two (fairly uncommon) reasons:
  - Efficiency:
    - Non-virtual function calls are a tiny bit faster (no indirect lookup)
    - A class with no virtual functions has objects without a vptr field
  - Control:
    - If `f()` calls `g()` in class X and `g` is not virtual, we’re guaranteed to call `X::g()` and not `g()` in some derived class
      - Particularly useful for framework design
- In Java, all methods are virtual, except static class methods, which aren’t associated with objects
- In C++ you can pick what you want
  - Omitting virtual can cause obscure bugs
Abstract Classes

- Sometimes we want to include a function in a class but only implement it in derived classes
  - In Java, we would use an abstract method
  - In C++, we use a “pure virtual” function
    - Example: `virtual string noise() = 0;`

- A class containing any pure virtual methods is abstract
  - You can’t create instances of an abstract class
  - Extend abstract classes and override methods to use them

- A class containing only pure virtual methods is the same as a Java interface
  - Pure type specification without implementations
STL and Inheritance

- Recall: STL containers store **copies of values**
  - What happens when we want to store mixes of object types in a single container? (e.g. `Stock` and `DividendStock`)
  - You get sliced 😞

```cpp
#include <list>
#include "Stock.h"
#include "DividendStock.h"

int main(int argc, char** argv) {
    Stock s;
    DividendStock ds;
    list<Stock> li;
    li.push_back(s); // OK
    li.push_back(ds); // OUCH!
    return EXIT_SUCCESS;
}
```
STL and Inheritance

- Instead, store **pointers to heap-allocated objects** in STL containers
  - No slicing! 😊
  - `sort()` does the wrong thing 😞
  - You have to remember to `delete` your objects before destroying the container 😞

  ■ Smart pointers!
Concurrency
Building a Web Search Engine

- We have:
  - An index (similar to database)
    - A map from `<word>` to `<list of documents containing the word>`
    - This is probably *sharded* over multiple files
  - A query processor
    - Accepts a query composed of multiple words
    - Looks up each word in the index
    - Merges the result from each word into an overall result set
Search Engine Architecture

- **index file**
- **index file**
- **index file**
- **query processor**
- **client**
Search Engine (Pseudocode)

doclist Lookup(string word) {
    bucket = hash(word);
    hitlist = file.read(bucket);
    foreach hit in hitlist {
        doclist.append(file.read(hit));
    }
    return doclist;
}

main() {
    SetupServerToReceiveConnections();
    while (1) {
        string query_words[] = GetNextQuery();
        results = Lookup(query_words[0]);
        foreach word in query[1..n] {
            results = results.intersect(Lookup(word));
        }
        Display(results);
    }
}
Execution Timeline: a Multi-Word Query

- network I/O
- main
- GetNextQuery
- disk I/O
- Lookup
- disk I/O
- Lookup
- disk I/O
- results.intersect
- CPU
- Lookup
- disk I/O
- results.intersect
- CPU
- Display
- network I/O
- GetNextQuery
What About I/O-caused Latency?

- Jeff Dean’s “Numbers Everyone Should Know” (LADIS ’09)
Execution Timeline: To Scale
Multiple (Single-Word) Queries

# is the Query Number
#.a -> GetNextQuery()
#.b -> network I/O
#.c -> Lookup() & file.read()
#.d -> Disk I/O
#.e -> Intersect() & Display()

CPU 1
  I/O 1.a

CPU 2
  I/O 2.a
  I/O 2.b

CPU 3
  I/O 3.a
  I/O 3.b

CPU 2
  I/O 2.d

CPU 3
  I/O 3.c

CPU 3
  I/O 3.d

CPU 3
  I/O 3.e

Query 1

Query 2

Query 3

Time
Multiple Queries: To Scale

query 1

query 2

query 3

I/O 1.b  I/O 1.d

I/O 1.b  I/O 1.d

I/O 1.b  I/O 1.d

---

Time
Uh-Oh (1 of 2)
The CPU is idle most of the time! (picture not to scale)

Only one I/O request at a time is “in flight”

Queries don’t run until earlier queries finish

query 1

query 2

query 3
Sequential Can Be Inefficient

- Only one query is being processed at a time
  - All other queries queue up behind the first one
  - And clients queue up behind the queries …

- Even while processing one query, the CPU is idle the vast majority of the time
  - It is *blocked* waiting for I/O to complete
    - Disk I/O can be very, very slow (10 million times slower …)

- At most one I/O operation is in flight at a time
  - Missed opportunities to speed I/O up
    - Separate devices in parallel, better scheduling of a single device, etc.
Concurrency

- Our search engine could run concurrently:
  - Example: Execute queries one at a time, but issue I/O requests against different files/disks simultaneously
    - Could read from several index files at once, processing the I/O results as they arrive
  - Example: Our web server could execute multiple queries at the same time
    - While one is waiting for I/O, another can be executing on the CPU

- Concurrency != parallelism
  - Concurrency is doing multiple tasks at a time
  - Parallelism is executing multiple CPU instructions simultaneously
A Concurrent Implementation

- Use multiple “workers”
  - As a query arrives, create a new “worker” to handle it
    - The “worker” reads the query from the network, issues read requests against files, assembles results and writes to the network
    - The “worker” uses blocking I/O; the “worker” alternates between consuming CPU cycles and blocking on I/O
  - The OS context switches between “workers”
    - While one is blocked on I/O, another can use the CPU
    - Multiple “workers”’ I/O requests can be issued at once

- So what should we use for our “workers”? 