Administrivia

HW3 due a week from Thursday - how’s it look?
- Section tomorrow: HW3 debugging disk files & more!

New exercise out, due Friday morning (STL maps)

Following exercise is about subclasses and will be out Friday, due Monday morning

Midterm grading still in process - should finish in a day or two
- Using gradescope (online grading and return)
- You’ll get mail from us and gradescope when ready
  ‣ Log on to gradescope with your @uw email address
HW3 tip

HW3 writes some pretty big index files

- Hundreds of thousands of write operations
- No problem for today’s fast machines and disks!!

Except...

- If you’re running on attu or a CSE lab linux workstation, every write to your personal directories goes to a network file server(!)

  ➤ Lots of slow network packets vs full-speed disks — can take much longer to write an index to a server vs. a few sec. locally (!!)

  ➤ Suggestion: write index files to /tmp/... . That’s a local scratch disk and is very fast. But please clean up when you’re done.
Today

C++ inheritance

- Review of basic idea (pretty much the same as 143)
- What’s different in C++ (compared to Java)
  ‣ *Static vs dynamic dispatch - virtual functions and vtables*
  ‣ *Pure virtual functions, abstract classes, why no Java “interfaces”*
  ‣ *Assignment slicing, using class hierarchies with STL*
- Casts in C++
- Reference: C++ Primer, ch. 15
  • Credits: Thanks to Marty Stepp for stock portfolio example
Let’s build a stock portfolio

A portfolio represents a person’s financial investments

- each asset has a cost (how much was paid for it) and a market value (how much it is worth)
  - the difference is the profit (or loss)

- different assets compute market value in different ways
  - **stock**: has a symbol (“GOOG”), a number of shares, share price paid, and current share price
  - **dividend stock**: is a stock that also has dividend payments
  - **cash**: money; never incurs profit or loss. (hah!)
One possible design

One class per asset type

- Problem: redundancy
- Problem: cannot treat multiple investments the same way
  - e.g., cannot put them in a single array or Vector
see initial_design/
Inheritance

A parent-child “is-a” relationship between classes

- a child (derived class) extends a parent (base class)

Benefits:

- code reuse: subclasses inherit code from superclasses
- polymorphism
  ‣ ability to redefine existing behavior but preserve the interface
  ‣ children can override behavior of parent
  ‣ others can make calls on objects without knowing which part of the inheritance tree it is in
- extensibility: children can add behavior
Better design

**Stock**
- symbol_
- total_shares_
- total_cost_
- current_price_

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

**DividendStock**
- symbol_
- total_shares_
- total_cost_
- current_price_
- dividends_

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

**Asset** (abstract)
- GetMarketValue()
- GetProfit()
- GetCost()

**Cash**
- amount_

- GetMarketValue()

**Mutual Fund**
- symbol_
- total_shares_
- total_cost_
- current_price_
- assets_ []

- GetMarketValue()
- GetProfit()
- GetCost()
Like Java: Access specifiers

**public**: visible to all other classes

**protected**: visible to current class and its subclasses

**private**: visible only to the current class

declare members as **protected** if:

- you don’t want random clients accessing them, but...
  - you want to be subclassed and let subclasses access them
Like Java: Public inheritance

- "public" inheritance
  - anything that is [public, protected] in the base is [public, protected] in the derived class - this is interface (specification) + implementation inheritance

- derived class inherits almost all behavior from the base class
  - not constructors and destructors
  - not the assignment operator or copy constructor

- (Yes there is "private" inheritance — don’t ask and don’t use)
Terminology

<table>
<thead>
<tr>
<th>C++, etc.</th>
<th>Java, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>base class</td>
<td>superclass</td>
</tr>
<tr>
<td>derived class</td>
<td>subclass</td>
</tr>
</tbody>
</table>

Means the same. You’ll hear both.
Revisiting the portfolio example

Without inheritance (separate class per type)

- lots of redundancy
- no type relationship between the classes
Revisiting the portfolio example

A derived class:
- **inherits** the behavior and state of the base class
- **overrides** some of the base class’s member functions
- **extends** the base class with new member functions, variables

### Stock
- symbol_
- total_shares_
- total_cost_
- current_price_

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

### DividendStock
- dividends_

- GetMarketValue()
- GetProfit()
- GetCost()
- PayDividend()
(implement better_design/)
Like Java: Dynamic dispatch

Usually, when a derived function is available to an object, we want that derived function to be invoked by it

- as we will see, this requires a runtime decision of what code to invoke

When a member function is invoked on an object...

- the code that is invoked is decided at run time, and is the most-derived function accessible to the object’s visible type
How to use dynamic dispatch

If you want a member function to use dynamic dispatch, prefix its declaration with the “virtual” keyword

- derived (child) functions don’t need to repeat the virtual keyword, but it traditionally has been good style to do so

- “override” added in C++ 11 - always use if it’s available
  - “virtual” + “override”? Be consistent; follow local conventions - trend is to use override only; virtual is redundant (but old code and some of ours uses both)

- This is how method calls work in Java (all normal methods are virtual; no “virtual” keyword needed)

( see even_better_design/ )
Dynamic dispatch

When a member function is invoked on an object

- the code that is invoked is decided at run time, and is the **most-derived function** accessible to the object’s visible type

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

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

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

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

```cpp
DividendStock dividend();

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

// invokes Stock::GetProfit(), since that function is
// inherited (i.e., not overridden). Stock::GetProfit()
// invokes Dividend::GetMarketValue(), since that is
// the most-derived accessible function.
 ds->GetProfit();

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

// invokes DividendStock::GetMarketValue()
 s->GetMarketValue();
```
Dynamic dispatch

Here’s what “most derived” means:

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

void function() {
    A *a_ptr;
    C c;

    // Why is this OK?
    a_ptr = &c;

    // Whose Foo() is called?
    a_ptr->Foo();
}
Dynamic dispatch

```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 {
}

void function() {
    A *a_ptr;
    C c;
    E e;

    // Whose Foo() is called?
    a_ptr = &c;
    a_ptr->Foo();

    // Whose Foo() is called?
    a_ptr = &e;
    a_ptr->Foo();
}
```
But how can this possibly work??

The compiler produces Stock.o from Stock.cc

- while doing this, it can’t know that DividendStock exists
  - so, how does the code emitted for Stock::GetProfit() know to invoke Stock::GetMarketValue() some of the time, and DividendStock::GetMarketValue() other times??!!?

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

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

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

If a member function is virtual, the compiler emits:

- a “vtable”, or virtual function table, **for the class**
  - it contains a function pointer for each virtual function in the class
  - the pointer points to the most-derived function for that class

- a “vptr”, or virtual table pointer, **for each object instance**
  - the vptr is a pointer to a virtual table, and it is essentially a hidden member variable inserted by the compiler
  - when the object’s constructor is invoked, the vptr is initialized to point to the virtual table for the object’s class
  - thus, the vptr “remembers” what class the object is
vtable/vptr example

class Base {
    public:
        virtual void fn1() {};
        virtual void fn2() {};
};

class Dr1: public Base {
    public:
        virtual void fn1() {};
};

class Dr2: public Base {
    public:
        virtual void fn2() {};
};

// what needs to work
Base b;
Dr1 d1;
Dr2 d2;

Base *bptr = &b;
Base *d1ptr = &d1;
Base *d2ptr = &d2;

bptr->fn1();  // Base::fn1()
bptr->fn2();  // Base::fn2()

d1ptr->fn1();  // Dr1::fn1()
d1ptr->fn2();  // Base::fn2()

d2.fn1();     // Base::fn1()
d2ptr->fn1(); // Base::fn1()
d2ptr->fn2(); // Dr2::fn2();
vtable/vptr example

// what happens

Base b;
Dr1 d1;
Dr2 d2;
Base *d2ptr = &d2;

d2.fn1();
// d2.vptr -->
// Dr2.vtable.fn1 -->
// Base::fn1()

d2ptr->fn2();
// d2ptr -->
// d2.vptr -->
// Dr2.vtable.fn2 ->
// Dr2::fn2()
actual code

```cpp
class Base {
    public:
        virtual void fn1() {}
        virtual void fn2() {}
};

class Dr1: public Base {
    public:
        virtual void fn1() {}
};

main() {
    Dr1 d1;
    d1.fn1();
    Base *ptr = &d1;
    ptr->fn1();
}
```

Let's compile this and use objdump to see what g++ emits!

- g++ -g vtable.cc
- objdump -CDSRTtx a.out | less
Static dispatch - What if we omit "virtual"?

When a member function is invoked on an object...

- the code that is invoked is decided at compile time, based on the compile-time visible type of the callee

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

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

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

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

```
DividendStock dividend();

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

// invokes Stock::GetProfit(), since that function is
// inherited (i.e, not overridden).  Stock::GetProfit()
// invokes Stock::GetMarketValue(), since C++ uses
// static dispatch by default.
    ds->GetProfit();

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

// invokes Stock::GetMarketValue()
    s->GetMarketValue();
```
Why not always use “virtual”?

Two (fairly uncommon) reasons:

- Efficiency:
  - non-virtual function calls are a tiny bit faster (no indirect lookup)
  - if the class has no virtual functions, objects will not have 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 subclass
  - Particularly useful for framework design

In Java, all functions (methods) are virtual (exception: static class methods, but these aren’t associated with objects — no “this” ptr)

In C++ and C# you can pick what you want
  - But omitting “virtual” often causes obscure bugs
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 version of) \( f \).

\( f() \) will be called using dynamic dispatch even if overridden but not explicitly specified as virtual in a subclass

- But it’s good style to help the reader by using override (and, particularly in older code, virtual) in subclasses
Pure virtual fcns, abstract classes

Sometimes we want to include a function in a class but only implement it in subclasses. In Java we would use an abstract method. In C++ we use a “pure virtual” function.

- Example: `virtual string noise() = 0;` // see zoo.cc

A class that contains a pure virtual method is abstract

- Can’t create instances of an abstract class (like Java)
- Extend abstract classes and override methods to use them (like Java)

A class containing only pure virtual methods is the same as a Java interface (::* no separate “interface” thingys in C++)

- Pure type specification without implementations
Inheritance and constructors

A derived class **does not inherit** the base class’s constructor

- the derived class *must* have its own constructor
  - if you don’t provide one, C++ synthesizes a default constructor for you
    - it initializes derived class’s non-POD member variables to zero-equivalents and invokes the default constructor of the base class
    - if the base class has no default constructor, a compiler error
- a constructor of the base class is invoked before the constructor of the derived class
  - you can specify which base class constructor in the initialization list of the derived class, or C++ will invoke default constructor of base class
// Base has no default constructor
class Base {
public:
    Base(int x) : y(x) { }
    int y;
};

// Compiler error when you try
// to instantiate a D1, as D1’s
// synthesized default constructor
// needs to invoke Base’s default
// constructor.
class D1 : public Base {
public:
    int z;
};

// Works.
class D2 : public Base {
public:
    D2(int z) : Base(z+1) {
        this->z = z;
    }
    int z;
};

// Base has a default constructor.
class Base {
public:
    int y;
};

// Works.
class D1 : public Base {
public:
    int z;
};

// Works.
class D2 : public Base {
public:
    D2(int z) {
        this->z = z;
    }
    int z;
};
Destructors

When the destructor of a derived class is invoked...
- the destructor of the base class is invoked after the destructor of the derived class finishes

Note that static dispatch of destructors is almost always a mistake!
- good habit to always define a destructor as virtual
  ▸ empty if you have no work to do

```cpp
class Base {
public:
    Base() { x = new int; }
    ~Base() { delete x; }
    int *x;
};

class D1 : public Base {
public:
    D1() { y = new int; }
    ~D1() { delete y; }
    int *y;
};

Base *b = new Base;
Base *dptr = (Base *) new D1;
delete b; // ok
delete dptr; // leaks D1::y
```
Slicing -- C++’s revenge

C++ allows you to...
- assign to...
  ‣ an instance of a base class...
  ‣ ...the value of a derived class

class Base {
    public:
    Base(int x) : x_(x) { }
    int x_; 
};

class Dr : public Base {
    public:
    Dr(int y) : Base(16), y_(y) { }
    int y_; 
};

main() {
    Base b(1);
    Dr d(2);
    b = d; // what happens to y_?
    // d = b; // compiler error
}
Given this, STL containers?? :(

STL stores **copies of values** in containers, not pointers to object instances

- so, what if you have a class hierarchy, and want to store mixes of object types in a single container?
  
  ‣ e.g., Stock and DividendStock in the same list

- you get sliced! :( 

```cpp
class Stock {
  ...
};

class DivStock : public Stock {
  ...
};

main() {
  Stock s;
  DivStock ds;
  list<Stock> li;

  li.push_back(s); // OK
  li.push_back(ds); // OUCH!
}
```
STL + inheritance: use pointers?

Store pointers to heap-allocated objects in STL containers

- no slicing :)  
  - you have to remember to delete your objects before destroying the container :(
  - sort() does the wrong thing :(:(

Use smart pointers!

```cpp
#include <list>
using namespace std;

class Integer {
public:
    Integer(int x) : x_(x) { }
private:
    int x_;  
};

main() {
    list<Integer *> li;
    Integer *i1 = new Integer(2);
    Integer *i2 = new Integer(3);
    li.push_back(i1);
    li.push_back(i2);
    li.sort();  // waaaaaah!!
}```
Explicit casting in C

C’s explicit typecasting syntax is simple

\[ \text{lhs} = (\text{new\_type}) \text{rhs}; \]

- C’s explicit casting is used to...
  - convert between pointers of arbitrary type
  - forcibly convert a primitive type to another
  - e.g., an integer to a float, so that you can do integer division

```c
int x = 5;
int y = 2;
printf("%d\n", x / y);       // prints 2
printf("%f\n", ((float)x) / y); // prints 2.5
```
C++

You can use C-style casting in C++, but C++ provides an alternative style that is more informative:

- `static_cast<to_type>(expression)`
- `dynamic_cast<to_type>(expression)`
- `const_cast<to_type>(expression)`
- `reinterpret_cast<to_type>(expression)`

Always use these in C++ code - helps document intent.
static_cast

C++’s static_cast can convert:

- pointers to classes **related type**
  - get a compiler error if you attempt to static_cast between pointers to non-related classes
  - dangerous to cast a pointer to a base class into a pointer to a derived class
- non-pointer conversion
  - float to int, etc.

static_cast is checked at compile time

```cpp
class Foo {  
    public:  
        int x_;  
    };  

class Bar {  
    public:  
        float x_;  
    };  

class Wow : public Bar {  
    public:  
        char x_;  
    };  

int main(int argc, char **argv) {  
    Foo a, *aptr;  
    Bar b, *bptr;  
    Wow c, *cptr;  

    // compiler error  
    aptr = static_cast<Foo*>(&b);  

    // OK  
    bptr = static_cast<Bar*>(&c);  

    // compiles, but dangerous  
    cptr = static_cast<Wow*>(&b);  
    return 0;  
}  
```
C++’s `dynamic_cast` can convert:

- pointers to classes of related type
- references to classes of related type

`dynamic_cast` is checked at both compile time and run time

- casts between unrelated classes fail at compile time
- casts from base to derived fail at run-time if the pointed-to object is not a full derived object

  ▶ result is `nullptr` if cast fails
const_cast

Is used to strip or add const-ness

- dangerous!

```cpp
void foo(int *x) {
    *x++;
}

void bar(const int *x) {
    foo(x);  // compiler error
    foo(const_cast<int*>(x));  // succeeds
}

main() {
    int x = 7;
    bar(&x);
}
```
reinterpret_cast

casts between incompatible types

- storing a pointer in an int, or vice-versa
  - works as long as the integral type is “wide” enough
- converting between incompatible pointers
  - dangerous!
    - But used (carefully) in HW3!!
Implicit conversion

The compiler tries to infer some kinds of conversions

- when you don’t specify an explicit cast, and types are not equal, the compiler looks for an acceptable implicit conversion

```cpp
void bar(std::string x);

void foo() {
    int x = 5.7;       // implicit conversion float -> int
    bar("hi");        // implicit conversion, (const char *) -> string
    char c = x;        // implicit conversion, int -> char
}
```
Sneaky implicit conversions

How did the (const char *) ---> string conversion work??

- if a class has a constructor with a single parameter, the compiler will exploit it to perform implicit conversions
- at most one user-defined implicit conversion will happen
  - can do int --> Foo
  - can’t do int --> Foo --> Baz

```c++
class Foo {
public:
  Foo(int x) : x_(x) { }
  int x_;
};

int Bar(Foo f) {
  return f.x_
}

int main(int argc, char **argv) {
  // The compiler uses Foo's (int x) constructor to make an implicit conversion from the int 5 to a Foo.
  // equiv to return Bar(Foo(5));
  return Bar(5);
}
```
Avoiding sneaky implicits

Declare one-argument constructors as “explicit” if you want to disable them from being used as an implicit conversion path
- usually a good idea

```cpp
class Foo {
public:
    explicit Foo(int x) : x_(x) {} 
    int x_; 
};

int Bar(Foo f) {
    return f.x_; 
}

int main(int argc, char **argv) {
    // compiler error
    return Bar(5); 
}
```
Exercise 1

Design a class hierarchy to represent shapes:

- examples of shapes: Circle, Triangle, Square

Implement methods that:

- construct shapes
- move a shape (i.e., add (x, y) to the shape position)
- returns the centroid of the shape
- returns the area of the shape
- Print(), which prints out the details of a shape
Exercise 2

Implement a program that:

- uses your exercise 1
  ▪ constructs a vector of shapes
  ▪ sorts the vector according to the area of the shape
  ▪ prints out each member of the vector

- notes:
  ▪ to avoid slicing, you’ll have to store pointers in the vector
  ▪ to be able to sort, you’ll have to implement a wrapper for the pointers, and you’ll have to override the “<“ operator
See you on Monday!