To-Do list:

- Section exercise due tonight, 11 pm
- (lecture) Exercise 9 due Monday before class
  - (A little bit of C++ subclassing)
- HW3 due next Thursday, 11 pm
  ‣ No more exercises after ex9 until hw3 is 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++

• Credits: Thanks to Marty Step 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
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()
(see stocks_virtual/*)
Like Java: Public inheritance

- “public” inheritance
  - anything that is [public, protected] in the base is [public, protected] in the derived class
- 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)

```cpp
#include "BaseClass.h"
class Name : public BaseClass {
  ...
};
```
## Terminology

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

Means the same. You’ll hear both.
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 a member as **protected** if:

- you don’t want random client code accessing them
  - you want to be subclassed and to let subclasses access them
Like Java — dynamic dispatch

Use *dynamic dispatch* to pick function to execute

- Use the actual type of the object to find method (function) appropriate for object
- *But* you have to say so with the `virtual` keyword

```cpp
class Stock {
  ...
  // Print out the Stock information.
  virtual void Print() const;
  ...
}
```
Dynamic Dispatch Example

In useassets.cc:

```cpp
Stock* stock = new Stock("MSFT");
DividendStock* dividend = new DividendStock("INTC");
...
// display info about each investment
stock->Print(); // uses Stock::Print
dividend->Print(); // uses DividendStock::Print
...
// create second pointer to DividendStock object
stock = dividend;
stock->Print(); // also uses DividendStock::Print
```
What if we omit “virtual”? 

If a function is not “virtual”, the actual function called depends on the compile-time (static) type of the variable, not the run-time (dynamic) type of the object it points at.

```cpp
Stock* stock = new Stock("MSFT");
DividendStock* dividend = new DividendStock("INTC");
...
// display info about each investment
stock->Print();   // uses Stock::Print
dividend->Print();   // uses DividendStock::Print
...
// create second pointer to DividendStock object
stock = dividend;
stock->Print();   // uses Stock::Print
// if Print is not virtual
```
( see stocks_static/* )
But how can this possibly work??

If `Print()` is virtual, then `stock->Print()` has to call the correct function depending on the (actual) object

- The compiled code has to be the same regardless of the actual object
  - and has to do the “right thing” even if the variable is assigned a new value during execution!

```cpp
Stock* stock = ...;
...
stock->Print();
```
vtables and the vptr

If a member function is virtual, the compiler emits:

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

```cpp
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() {};  
};
```

```cpp
// 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();
```
// 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.fn1 -->
// Base::fn1()
actual code

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
Why not always use “virtual”?

Two (fairly uncommon) reasons:

- Efficiency: non-virtual functions are a tiny bit faster, and 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; in C++ and C# you can pick what you want

  ‣ But omitting “virtual” causes obscure bugs
Virtual is “sticky”

If \texttt{X::f()} is declared virtual, then a vtable will be created for class \texttt{X} and for all of its subclasses. The vtables will include function pointers for (the correct version of) \texttt{f}.

\texttt{f()} will be called using dynamic dispatch even if overridden but not explicitly specified as \texttt{virtual} in a subclass.
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 it (like Java)

A class containing only pure virtual methods is the same as a Java interface

- 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 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
Examples

// 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 defined a destructor as virtual

  - empty if you have no work to do
Slicing -- C++’s revenge

C++ allows you to...

- assign to..
  ‣ an instance of a base class...
  ‣ the value of a derived class

```cpp
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;

    l.push_back(s);   // OK
    l.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 (same bits)
  - convert a primitive type to another (different bit representation)
  - 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)`
static_cast

C++’s static_cast can convert:

- pointers to classes **of 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

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
const_cast

Is used to strip or add const-ness

- dangerous!

```cpp
class MyClass {

    public:
        int x;
        MyClass() {} // Constructor
        ~MyClass() {} // Destructor

        int f() {
            return x;
        }

        const cast MyClass& operator=(const MyClass& other) {
            x = other.x;
            return *this;
        }

};
```

```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!
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 --&gt; Foo
    - can’t do int --&gt; Foo --&gt; Baz

```cpp
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) {
    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!