CSE333 Systems Programming
Smart Pointers (cont’d)

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

Administrivia

- Midquarter survey out on Canvas (lol), due a week from whenever Canvas comes back up (😿)…

Choosing Between Smart Pointers

- `unique_ptr`s make ownership very clear
  - Generally the default choice due to reduced complexity – the owner is responsible for cleaning up the resource
    - Example: would make sense in HW1 & HW2, where we specifically documented who takes ownership of a resource
  - Less overhead: small and efficient

- `shared_ptr`s allow for multiple simultaneous owners
  - Reference counting allows for "smarter" deallocation but consumes more space and logic and is trickier to get right
  - Common when using more "well-connected" data structures

Aside: Smart Pointers and Arrays

- Smart pointers can store arrays as well and will call `delete[]` on destruction

uniquearray.cc

```
#include <memory>    // for std::unique_ptr
#include <cstdlib>   // for EXIT_SUCCESS

using std::unique_ptr;

int main(int argc, char** argv) {
  unique_ptr<int[]> x(new int[5]);

  x[0] = 1;
  x[2] = 2;

  return EXIT_SUCCESS;
}
```

Lecture Outline

- **Smart Pointer Limitations**
  - **`std::weak_ptr`**
- HW3 Overview
- Sneak preview: C++ Class Inheritance

Limitations with Smart Pointers

- Smart pointers are only as "smart" as the behaviors that have been built into their class methods and non-member functions!

- Limitations we will look at now:
  - Can't tell if pointer is to the heap or not
  - Circumventing ownership rules
  - Still possible to leak memory!
  - Sorting smart pointers

Using a Non-Heap Pointer

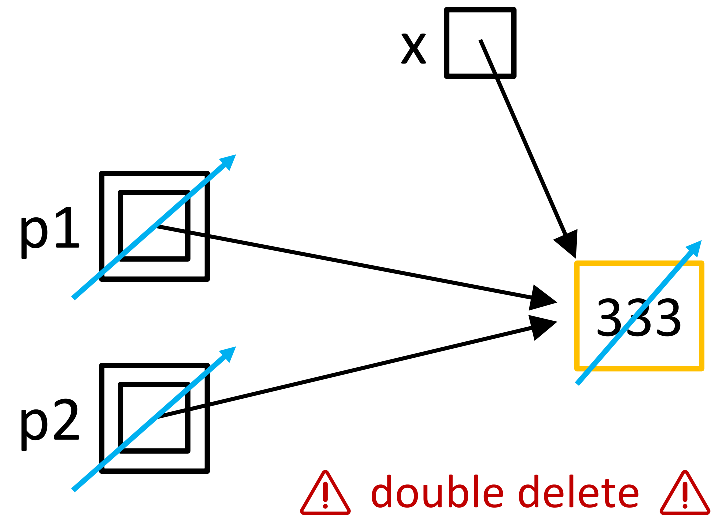
- Smart pointers will still call `delete` when destructed

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

int main(int argc, char** argv) {
  int x = 333;

  shared_ptr<int> p1(&x);

  return EXIT_SUCCESS;
}
```

Re-using a Raw Pointer (unique_ptr)

- Smart pointers can't tell if you are re-using a raw pointer

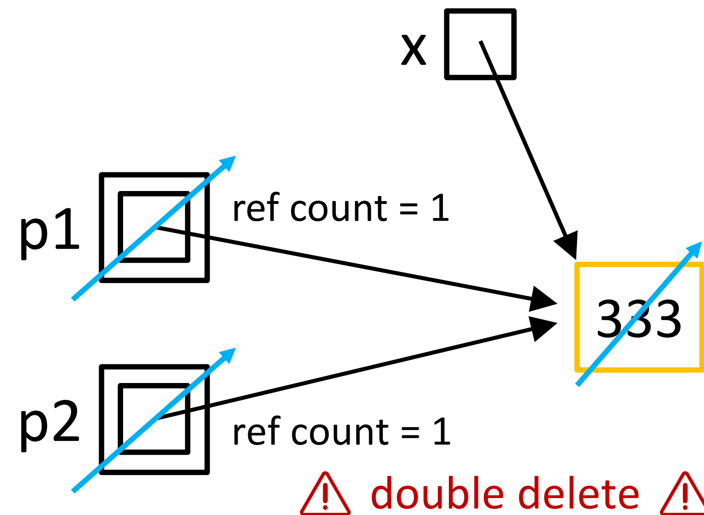
```
#include <cstdlib>
#include <memory>

using std::unique_ptr;

int main(int argc, char** argv) {
  int* x = new int(333);

  unique_ptr<int> p1(x);

  unique_ptr<int> p2(x);

  return EXIT_SUCCESS;
}
```



Re-using a Raw Pointer (shared_ptr)

- Smart pointers can't tell if you are re-using a raw pointer

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

int main(int argc, char** argv) {
  int* x = new int(333);

  shared_ptr<int> p1(x);

  shared_ptr<int> p2(x);

  return EXIT_SUCCESS;
}
```



Solution: Don't Use Raw Pointer Variables

- Smart pointers replace your raw pointers; passing `new` and then using the copy constructor is safer:

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

int main(int argc, char** argv) {
  int* x = new int(333);

  shared_ptr<int> p1(new int(333));

  shared_ptr<int> p2(p1);

  return EXIT_SUCCESS;
}
```

Caution Using get()

- Smart pointers still have functions to return the raw pointer without losing its ownership
  - get() can circumvent ownership rules!

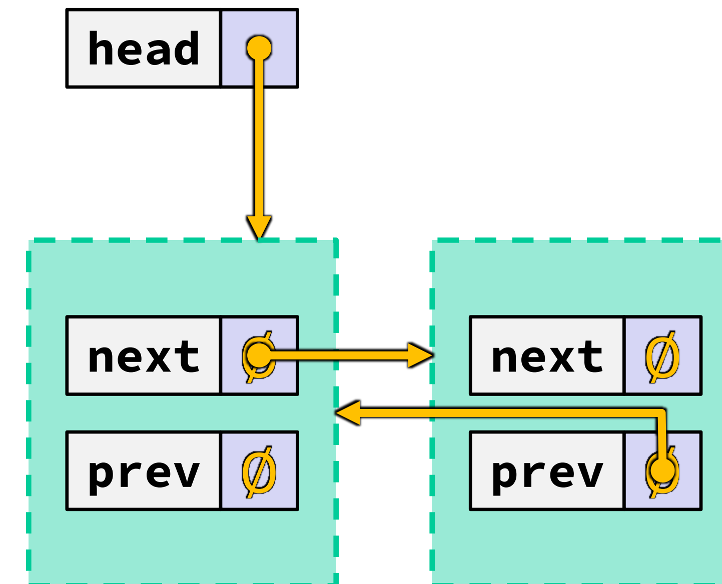
```
#include <cstdlib>
#include <memory>

// Same as re-using a raw pointer
int main(int argc, char** argv) {

  unique_ptr<int> p1(new int(5));

  unique_ptr<int> p2(p1.get());

  return EXIT_SUCCESS;
}
```

Cycle of `shared_ptr`s

- What happens when `main` returns?

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

struct A {
  shared_ptr<A> next;
  shared_ptr<A> prev;
};

int main(int argc, char** argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```

sharedcycle.cc



Solution: `weak_ptr`s

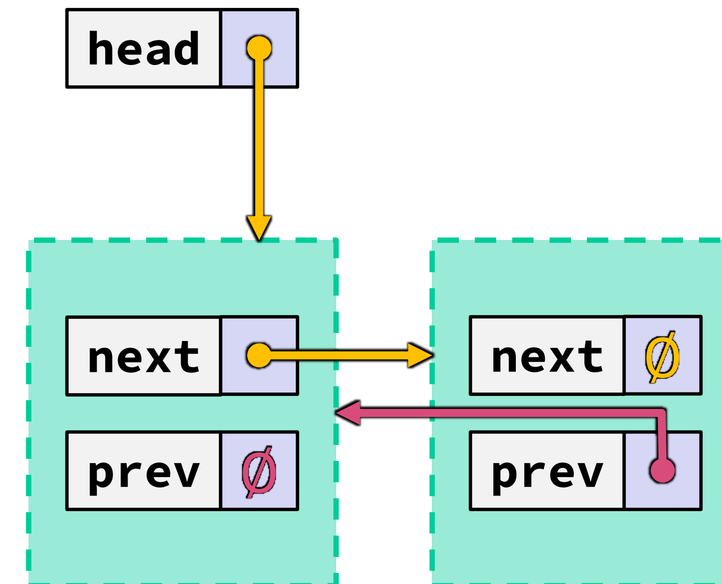
- `weak_ptr` is similar to a `shared_ptr` but *doesn't affect* the reference count
  - https://cplusplus.com/reference/memory/weak_ptr/
  - Not really a pointer as it **cannot be dereferenced** (!) – would break our notion of shared ownership
    - To deference, you first use the **`lock`** method to get an associated `shared_ptr`

Breaking the Cycle with weak_ptr

- Now what happens when main returns?

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

struct A {
  shared_ptr<A> next;
  weak_ptr<A> prev;
};

int main(int argc, char** argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```

weakcycle.cc



Dangling `weak_ptr`s

- `weak_ptr`s don't change reference count and can become *"dangling"*
  - Data referenced may have been `delete`'d

weakrefcount.cc

```
... (includes and other examples)
int main(int argc, char** argv) {
  std::weak_ptr<int> w;

  {  // temporary inner scope
    std::shared_ptr<int> y(new int(10));
    w = y; // assignment operator of weak_ptr takes a shared_ptr
    std::shared_ptr<int> x = w.lock();  // "promoted" shared_ptr

    std::cout << *x << " " << w.expired() << std::endl;
  }
  std::cout << w.expired() << std::endl;
  w.lock();  // returns a nullptr

  return EXIT_SUCCESS;
}
```

Smart Pointers and "<"

- Smart pointers implement some comparison operators, including `operator<`
  - However, it doesn't invoke `operator<` on the pointed-to objects; instead, it just promises a stable, strict ordering (probably based on the pointer address, not the pointed-to-value)

- To use the `sort()` algorithm on a container like `vector`, you need to provide a comparison function

- To use a smart pointer in a sorted container like `map`, you need to provide a comparison function when you *declare* the container

unique_ptr and STL Sorting

uniquevecsort.cc

```
using namespace std;
bool sortfunction(const unique_ptr<int>& x,
                  const unique_ptr<int>& y) { return *x < *y; }
void printfunction(unique_ptr<int>& x) { cout << *x << endl; }

int main(int argc, char** argv) {
  vector<unique_ptr<int> > vec;
  vec.push_back(unique_ptr<int>(new int(9)));
  vec.push_back(unique_ptr<int>(new int(5)));
  vec.push_back(unique_ptr<int>(new int(7)));

  // buggy: sorts based on the values of the ptrs
  sort(vec.begin(), vec.end());
  cout << "Sorted:" << endl;
  for_each(vec.begin(), vec.end(), &printfunction);

  // better: sorts based on the pointed-to values
  sort(vec.begin(), vec.end(), &sortfunction);
  cout << "Sorted:" << endl;
  for_each(vec.begin(), vec.end(), &printfunction);

  return EXIT_SUCCESS;
}
```

unique_ptr, "<", and maps

- Similarly, you can use unique_ptrs as keys in a map
  - Reminder: a map internally stores keys in sorted order
    - Iterating through the map iterates through the keys in order
  - By default, "<" is used to enforce ordering
    - You must specify a comparator when *constructing* the map to get a meaningful sorted order using "<" of unique_ptrs

- Compare (the 3rd template) parameter:
  - "A binary predicate that takes two element *keys* as arguments and returns a bool. This can be a function pointer or a function object."
    - `bool fptr(T1& lhs, T1& rhs);` OR member function `bool operator() (const T1& lhs, const T1& rhs);`

unique_ptr and map Example

uniquemap.cc

```
struct MapComp {
  bool operator()(const unique_ptr<int>& lhs,
        const unique_ptr<int>& rhs) const { return *lhs < *rhs; }
};

int main(int argc, char** argv) {
  map<unique_ptr<int>, int, MapComp> a_map;  // Create the map

  unique_ptr<int> a(new int(5));  // unique_ptr for key
  unique_ptr<int> b(new int(9));
  unique_ptr<int> c(new int(7));

  a_map[std::move(a)] = 25;  // move semantics to get ownership
  a_map[std::move(b)] = 81;  // of unique_ptrs into the map.
  a_map[std::move(c)] = 49;  // a, b, c hold NULL after this.

  map<unique_ptr<int>,int>::iterator it;
  for (it = a_map.begin(); it != a_map.end(); it++) {
    std::cout << "key: " << *(it->first);
    std::cout << " value: " << it->second << std::endl;
  }
  return EXIT_SUCCESS;
}
```

Summary of Smart Pointers

- A `shared_ptr` utilizes *reference counting* for multiple owners of an object in memory
  - `delete`s an object once its reference count reaches zero
- A `weak_ptr` works with a shared object but doesn't affect the reference count
  - Can't actually be dereferenced, but can check if the object still exists and can get a `shared_ptr` from the `weak_ptr` if it does
- A `unique_ptr` ***takes ownership*** of a pointer
  - Cannot be copied, but can be moved

Some Important Smart Pointer Methods

Visit http://www.cplusplus.com/ for more information on these!

- `std::unique_ptr<T> U;`
  - `U.get()` Returns the raw pointer U is managing
  - `U.release()` U stops managing its raw pointer and returns the raw pointer
  - `U.reset(q)` U cleans up its raw pointer and takes ownership of q
- `std::shared_ptr<T> S;`
  - `S.get()` Returns the raw pointer S is managing
  - `S.use_count()` Returns the reference count
  - `S.unique()` Returns true iff S.use_count() == 1
- `std::weak_ptr<T> W;`
  - `W.lock()` Constructs a shared pointer based off of W and returns it
  - `W.use_count()` Returns the reference count
  - `W.expired()` Returns true iff W is expired (W.use_count() == 0)

HW3 overview

- Let's take a look

Lecture Outline

- Smart Pointer Limitations
- HW3 Overview
- **Sneak preview: C++ Class Inheritance**

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 Terminology

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

- Terminology:

Java	C++
Superclass	Base Class
Subclass	Derived Class

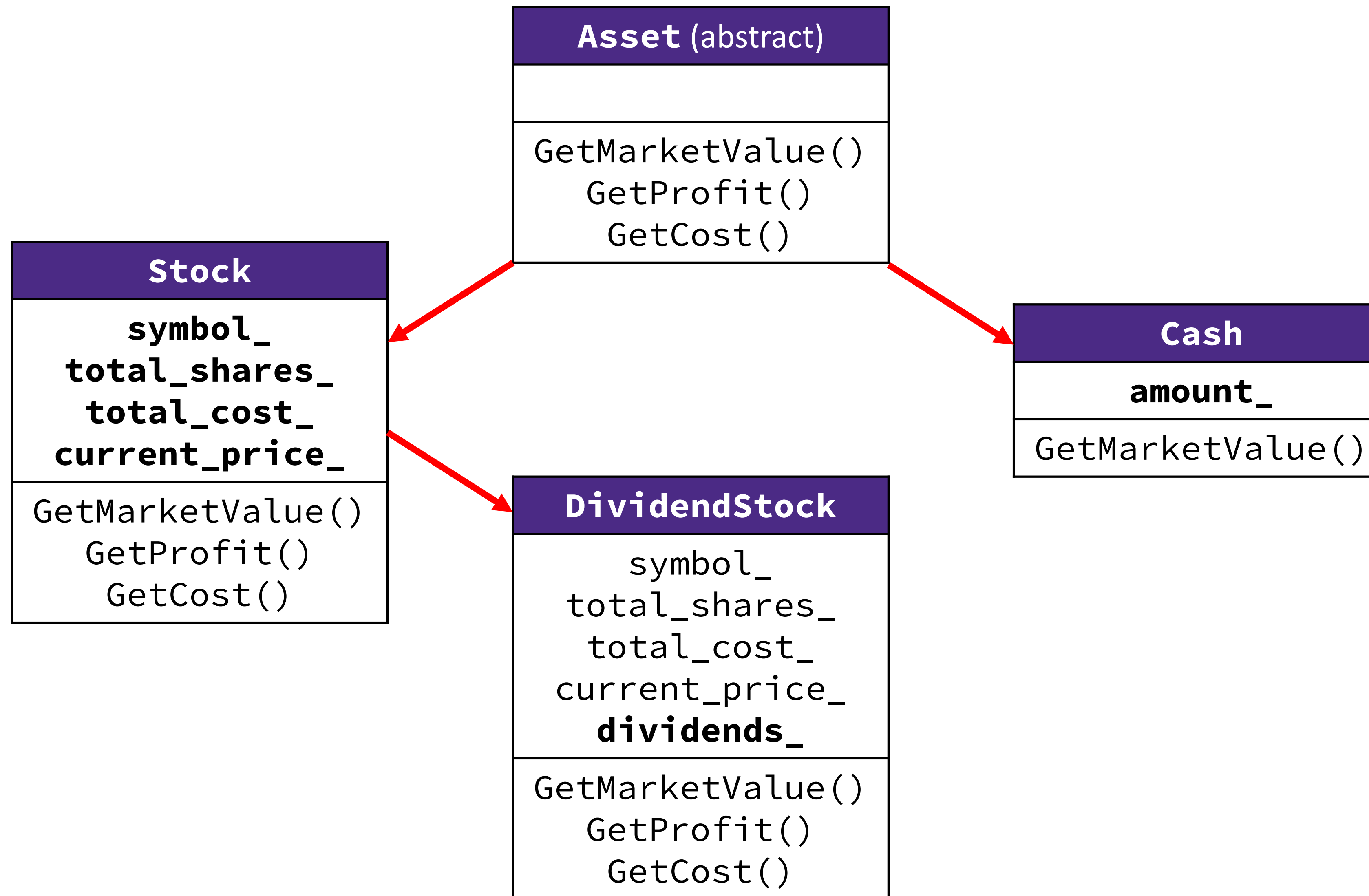
  - Mean the same things. You'll hear both.

Inheritance Benefits

- 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



Access Modifiers (like Java)

- `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:

```
#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

Stocks Classes Relationship (1/2)

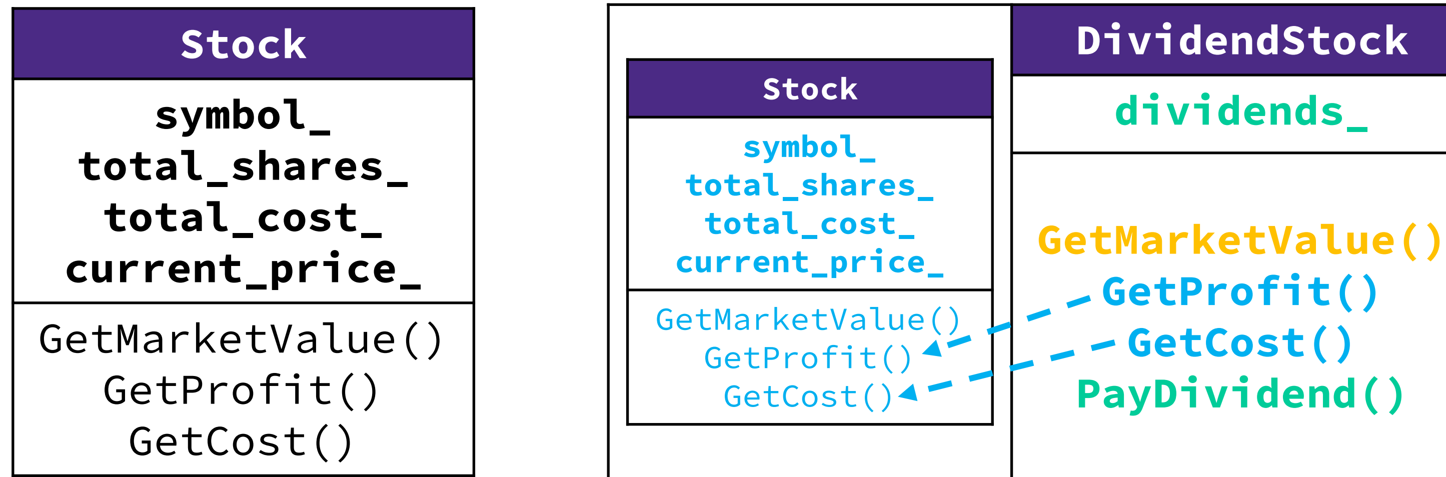
Stock
symbol_ **total_shares_** **total_cost_** **current_price_**
GetMarketValue() GetProfit() GetCost()

BASE

DividendStock
symbol_ total_shares_ total_cost_ current_price_ **dividends_**
GetMarketValue() GetProfit() GetCost()

DERIVED

Stocks Classes Relationship (2/2)



- 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.)

Extra Exercise #1

- Take one of the books from HW2's `test_tree` and:
  - Read in the book, split it into words (you can use your hw2)
  - For each word, insert the word into an STL `map`
    - The key is the word, the value is an integer
    - The value should keep track of how many times you've seen the word, so each time you encounter the word, increment its map element
    - Thus, build a histogram of word count
  - Print out the histogram in order, sorted by word count
  - Bonus: Plot the histogram on a log-log scale (use Excel, gnuplot, etc.)
    - x-axis: log(word number), y-axis: log(word count)

Extra Exercise #2

- Implement `Triple`, a class template that contains three "things," *i.e.*, it should behave like `std::pair` but hold 3 objects instead of 2
  - The "things" can be of different types

- Write a program that:
  - Instantiates several `Triple`s that contain `ToyPtr<int>`s
  - Insert the `Triple`s into a `vector`
  - Reverse the `vector`
  - Doesn't have any memory errors (use Valgrind!)
  - Note: You will need to update `ToyPtr.h` – how?