CSE 374 Programming concepts and tools

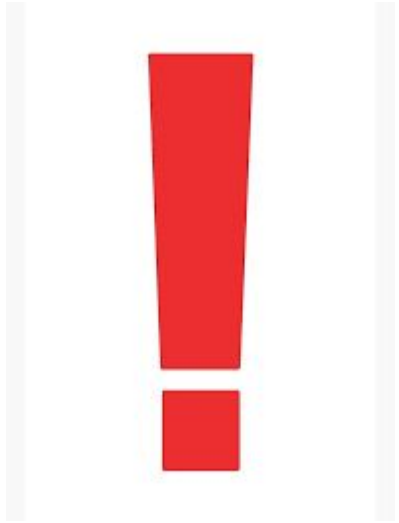
Winter 2024 Instructor: Alex McKinney

HW9 🤔

HW9: Concurrency

I've decided to make HW9 an **extra credit** assignment.

- Released on Sunday, March 3
- Due Sunday, March 10
  - Late days will not be accepted on this assignment.
  - No extensions.



This is your chance to recuperate some points you may have lost.

This assignment is challenging.

- It covers concurrency, which we will go over in class on Friday.
- It will require a substantial amount of **self-learning and research**.
- No autograder.

HW9: Concurrency (cont.)

The purpose of this assignment is to show you how far you've come!

- We've already learned three different programming language thus far (bash, C/C++).
- You now have the skills required to learn any programming language out there.
- "Learn how to acquire additional information and skills independently" (syllabus)

You will need to write code in a foreign code base (Go).

- TAs are not expected to have finished this assignment.
- Assistance will be scarce.

A Tour of Go

https://go.dev/tour

Go is modern language widely used in the software engineering industry (just like bash and C/C++).

- Open-sourced by Google
- Designed by C++ experts
- Syntax is similar to C, but has garbage collection.

Questions?

Review

Review: `std::unique_ptr`

A `unique_ptr` **takes ownership** of a pointer

- Part of C++'s standard library (C++11)
- Its destructor invokes `delete` on the owned pointer
  - Invoked when `unique_ptr` object is `delete`'d or falls out of scope via the `unique_ptr` destructor
- Guarantees uniqueness by disabling copy and assignment.

Review: `std::shared_ptr`

`shared_ptr` is similar to `unique_ptr` but we allow shared objects to have multiple owners

- The copy/assign operators are not disabled and they *increment* reference counts as needed
- When a `shared_ptr` is destroyed, the reference count is decremented
  - When the reference count hits 0, we `delete` the pointed-to object!
- Allows us to create complex linked structures (double-linked lists, graphs, etc.) at the cost of maintaining reference counts

Review: `std::weak_ptr`

`weak_ptr` is similar to a `shared_ptr` but doesn't affect the reference count

- Can *only* "point to" an object that is managed by a `shared_ptr`
- Not *really* a pointer – can't actually dereference unless you "get" its associated `shared_ptr`
- Because it doesn't influence the reference count, `weak_ptr`s can become "*dangling*"
  - Object referenced may have been `delete`'d
  - But you can check to see if the object still exists
- Can be used to break our cycle problem!

Questions?

C++ Inheritance

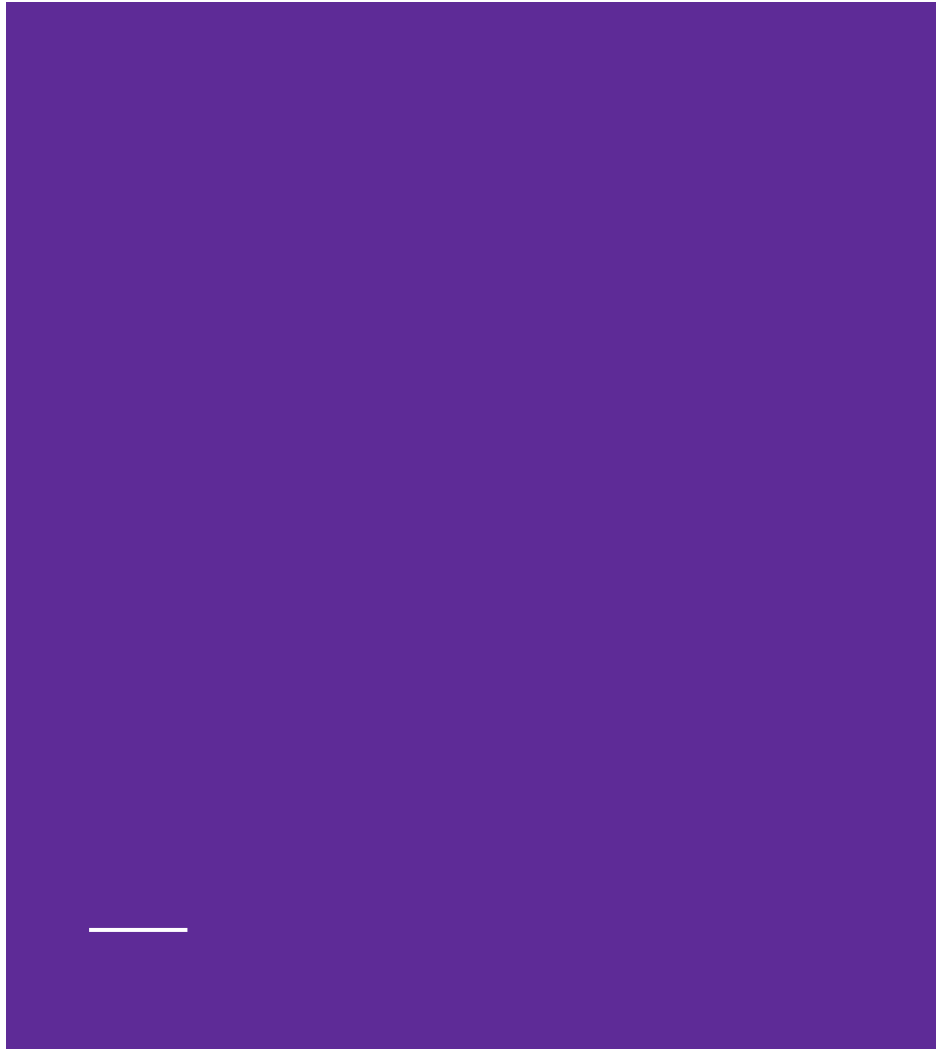
Overview

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 (i.e., dynamic dispatch) are optional*
  - *Pure virtual functions, abstract classes, why no Java "interfaces"*
  - *Assignment slicing, using class hierarchies with STL*
- HW8 Walkthrough

Motivation

and C++ Syntax



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

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:

Java	C++
Superclass	Base Class
Subclass	Derived Class

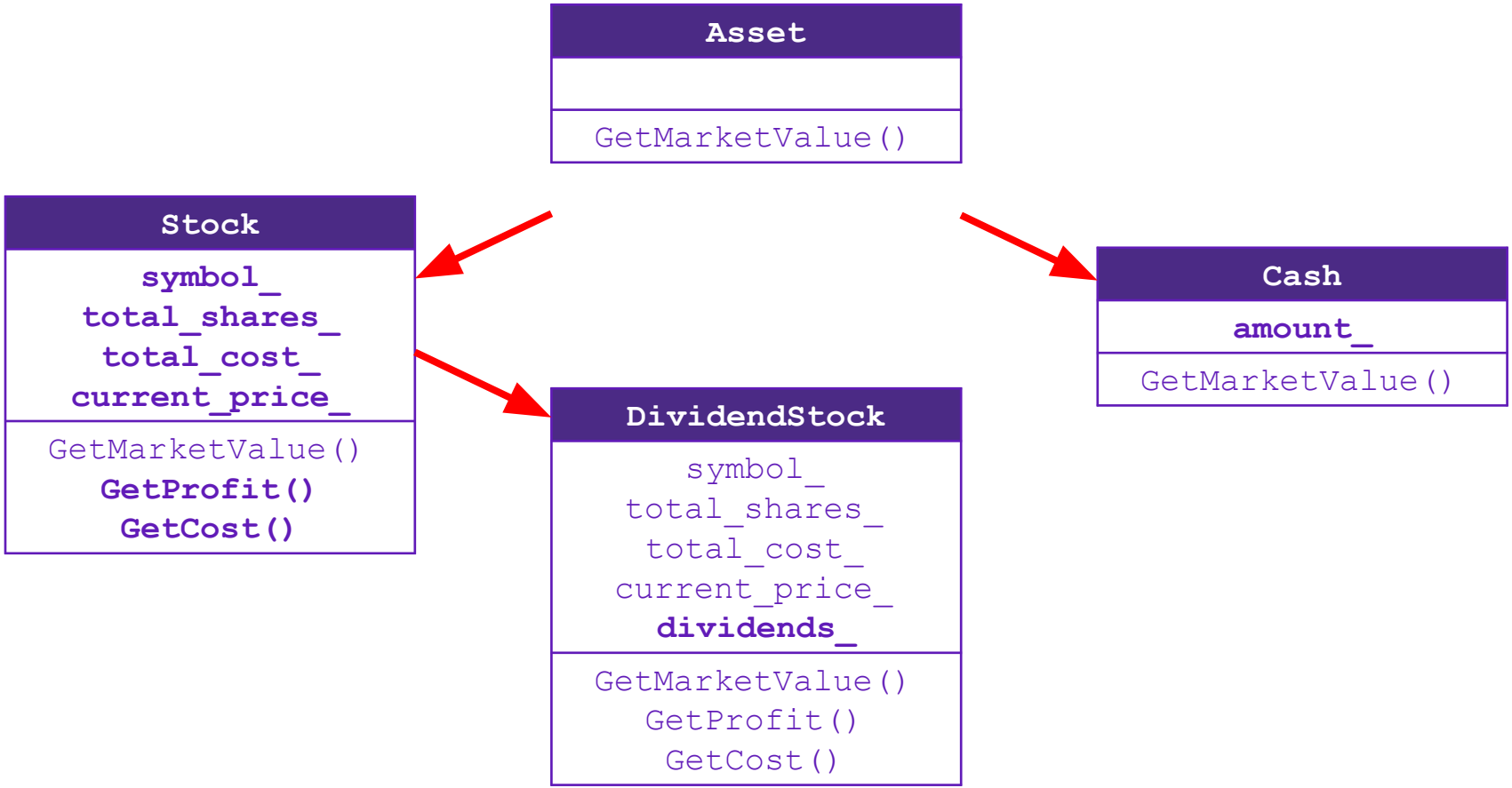
- Java: Subclass inherits from super class. (Superclass is “higher” in the hierarchy)
- C++: Derived class inherits from base class. (Base class is “higher” in the hierarchy)
  - Think of derived class as a derivative of the base class (e.g. car class is a derivative of vehicle class)
  - Mean the same things. You’ll hear both.

Inheritance

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

- `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
- Derived classes must have access but clients should not be allowed

`protected` isn't truly protected as an adversary client can just extend a protected class and get access to the "protected" information. Hence its use is rather limited in the real world.

Class derivation List

Comma-separated list of classes to inherit from:

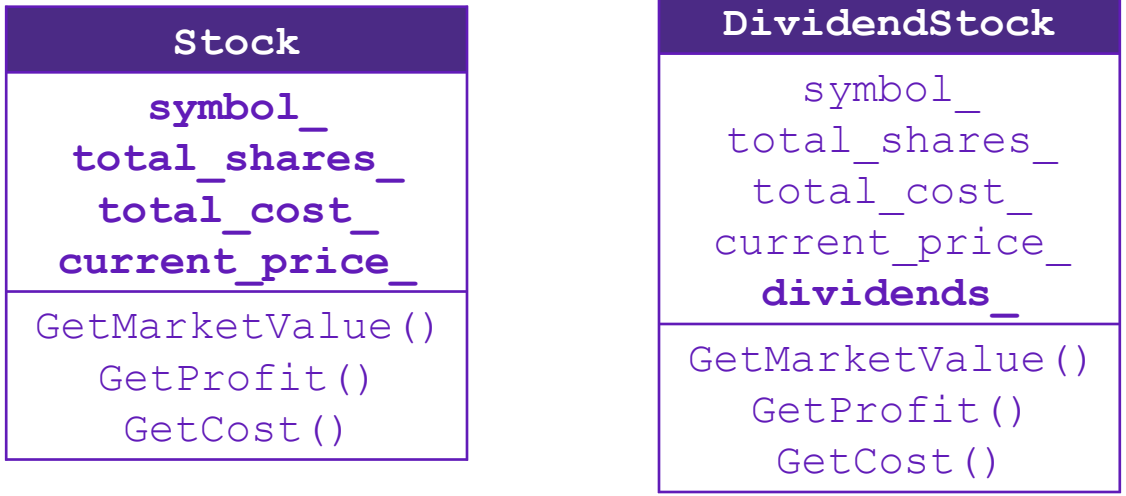
```
#include "BaseClass.h"
class Name : public BaseClass {
  ...
};
```

- Focus on single inheritance, but *multiple inheritance* possible
  - `: public Base1, public Base2 {`

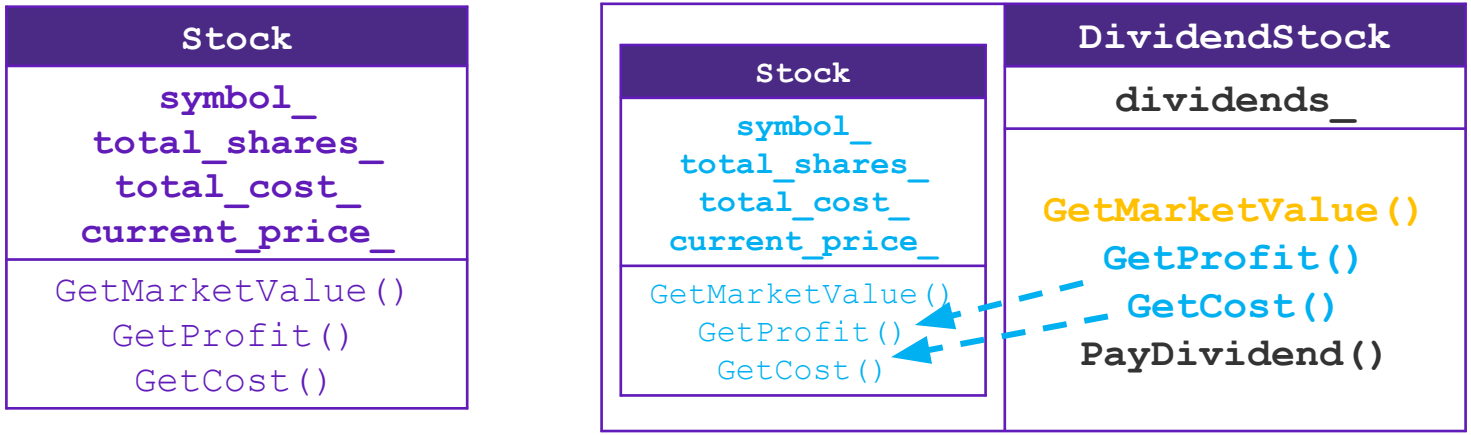
Almost always you will want public inheritance

- 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



Back to Stocks



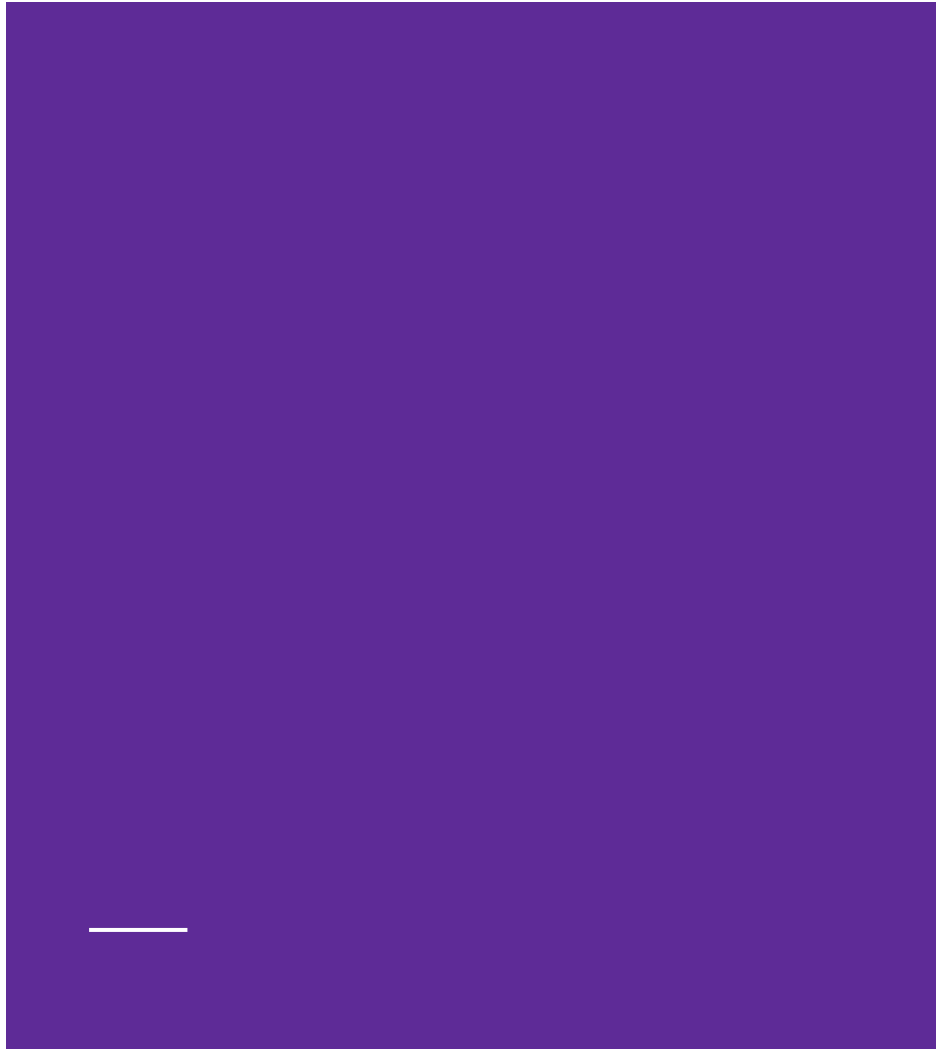
A derived class:

- **Inherits** the behavior and state (specification) of the base class
- **Overrides** some of the base class' member functions (optional)
- **Extends** the base class with new member functions, variables (optional)

Questions?

Polymorphism

& Dynamic Dispatch



Polymorphism 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`
- `PromisedType` defines the *interface* (*i.e.* what can be called on `var_p`), but `ActualType` may determine which *version* gets invoked

Analogy: A box labeled “cell phone” could hold Android or iPhone

- `PromisedType` is the box, `ActualType` is the Android or iPhone

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
- This is the behavior in Java

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

Is this a Stock or a DividendStock ?

Example: `void PrintStock(Stock* s) { s->Print(); }`

- Calls the appropriate `Print()` without knowing the exact class of `*s`, other than it is some sort of Stock

Requesting Dynamic Dispatch

Prefix the member function declaration with the `virtual` keyword

- Derived/child functions don't need to repeat `virtual`, but is 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
- C++ doesn't do dynamic dispatch by default so virtual keyword is strictly required if we want to make sure we're calling the most derived version of a function.

`override` keyword (C++11)

- Tells compiler this method should be overriding an inherited virtual function – **always** use when you can
- Prevents overloading vs. overriding bugs

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)

Inherited from stock

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

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

Should call DividendStock::GetMarketValue()

DividendStock.cc

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

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

Stock.cc

Dynamic Dispatch Example

```
#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();
```

A DividendStock “is-a” Stock, and has every part of Stock’s interface

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()
};
```

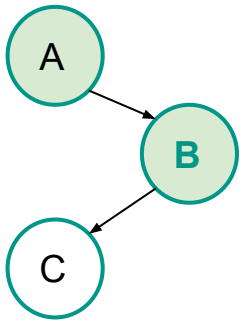
```
void Bar() {
  A* a_ptr;
  C c;

  a_ptr = &c;

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

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;

  // B::Foo() is called
  a_ptr->Foo();
}
```

Has `Foo` definition



Poll Question (PollEv.com/cs374)

Whose **Foo**() is called?

	Q1	Q2
A.	A	B
B.	A	D
C.	B	B
D.	B	D

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

  // Q1:
  a_ptr = &c;
  a_ptr->Foo();

  // Q2:
  a_ptr = &e;
  a_ptr->Foo();
}
```

```
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 {
};
```

Who Foo() is called?

0

A B

0%

A D

0%

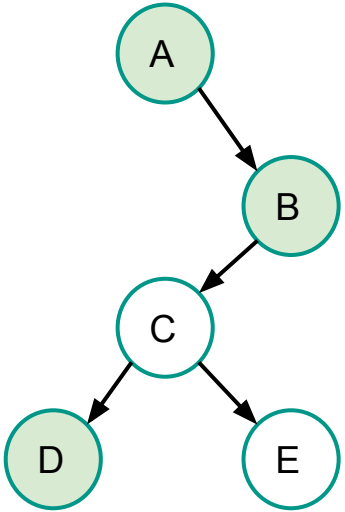
B B

0%

B D

0%

Poll Question (PollEv.com/cs374)



Has `Foo` definition

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

  // Q1:
  a_ptr = &c;
  a_ptr->Foo();

  // Q2:
  a_ptr = &e;
  a_ptr->Foo();
}
```

```
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 {
};
```

Questions?

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!**

Stock.h

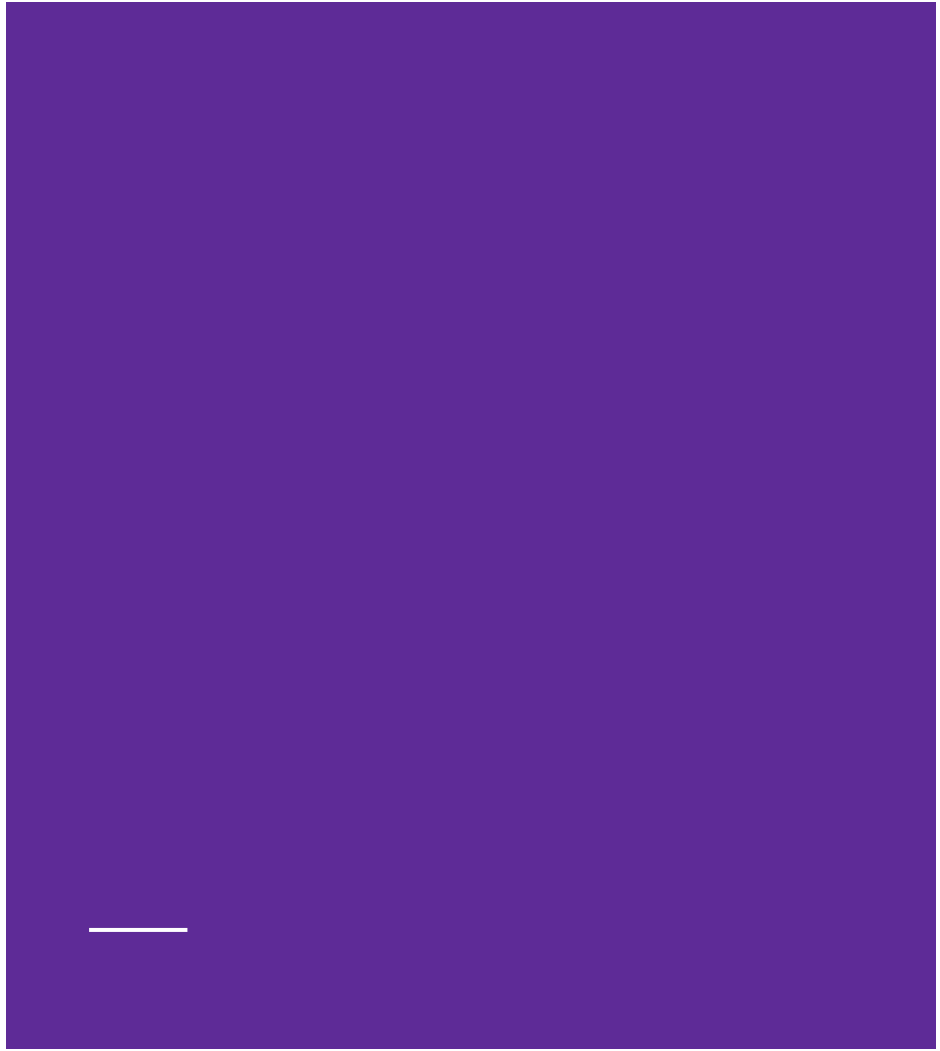
```
virtual double Stock::GetMarketValue() const;
virtual double Stock::GetProfit() const;
```

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

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

Stock.cc

Virtual Table

Virtual tables & virtual table pointers

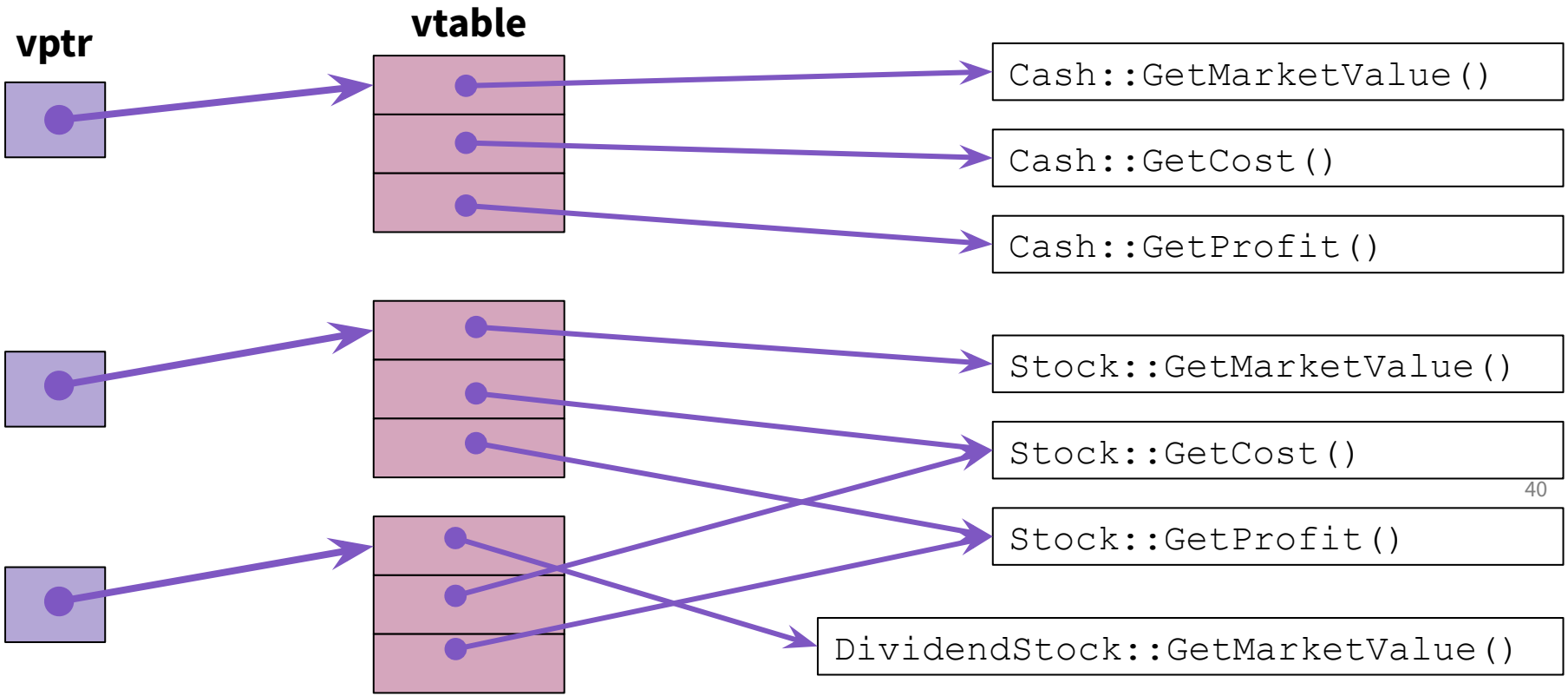


vtables and the vptr

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 newly constructed object’s class
  - Thus, the vptr “remembers” what class the object is

vptr and vtable Visualization



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();
};
```

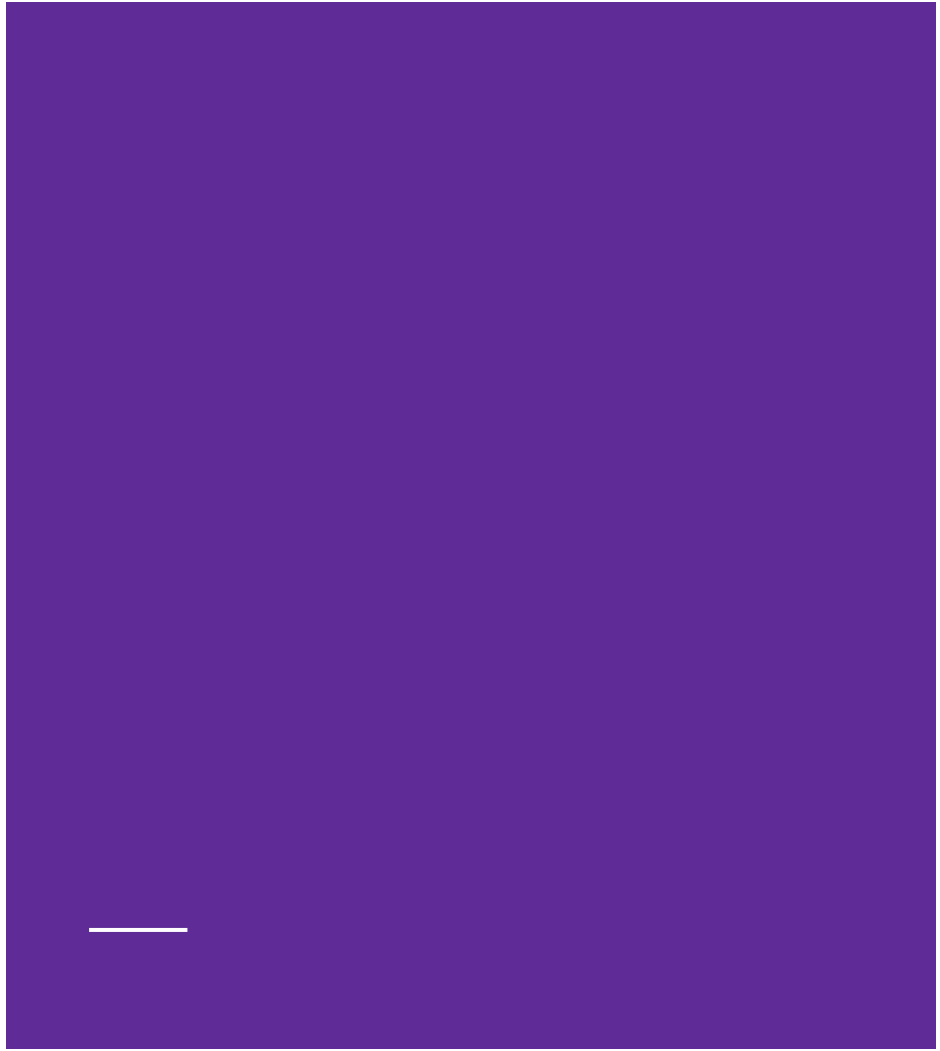
```
Base b;
Der1 d1;
Der2 d2;

Base* b0ptr = &b;
Base* b1ptr = &d1;
Base* b2ptr = &d2;

b0ptr->f1();
b0ptr->f2();

b1ptr->f1();
b1ptr->f2();

d2.f1();
b2ptr->f1();
b2ptr->f2();
```

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()
```

Static Dispatch

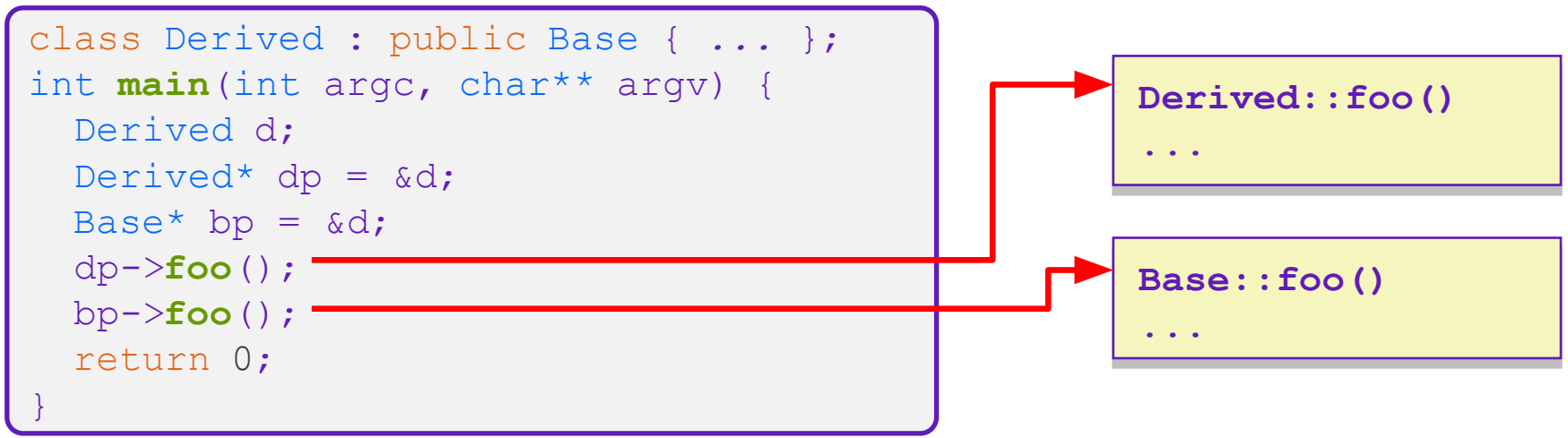


What happens if we omit “virtual”?

By default, without `virtual`, methods are dispatched statically

- At compile time, the compiler writes in a `call` to the address of the class’ method based on the compile-time visible type of the callee
- This is *different* than Java

```
class Derived : public Base { ... };
int main(int argc, char** argv) {
  Derived d;
  Derived* dp = &d;
  Base* bp = &d;
  dp->foo();
  bp->foo();
  return 0;
}
```



Static Dispatch Example

Removed `virtual` on methods:

Stock.h

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

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

ds->GetMarketValue(); // Calls DividendStock::GetMarketValue()
s->GetMarketValue();  // Calls Stock::GetMarketValue()

s->GetProfit(); // Calls Stock::GetProfit(). Stock::GetProfit()
calls Stock::GetMarketValue().

ds->GetProfit(); // Calls Stock::GetProfit(), since that method is
inherited. Stock::GetProfit() calls Stock::GetMarketValue().
```

`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

- Good style to help the reader *and avoid bugs* by using `override`
  - Style guide controversy, if you use `override` should you use `virtual` in derived classes? Recent style guides say just use `override`, but you'll sometimes see both, particularly in older code

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 zero 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 subclass
  - 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
- (Most of the time, you want member function to be virtual)

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 (e.g. `asset`)

Questions?

Walkthrough: HW8

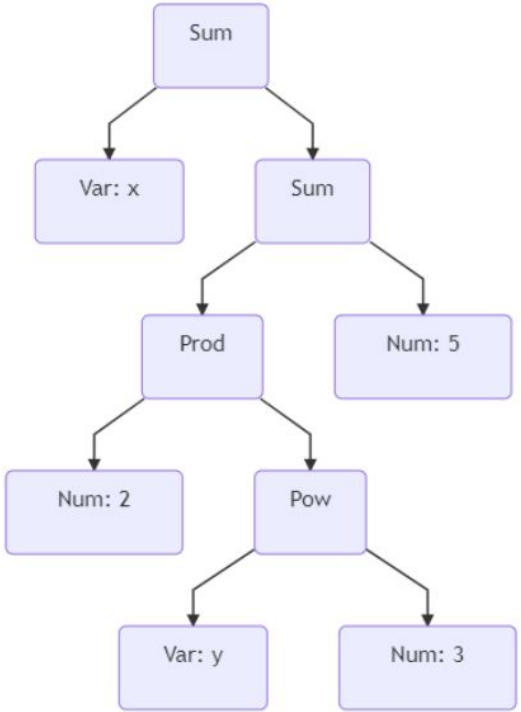
Abstract Syntax Tree

An AST is tree data structure that is commonly used to implement programming languages.

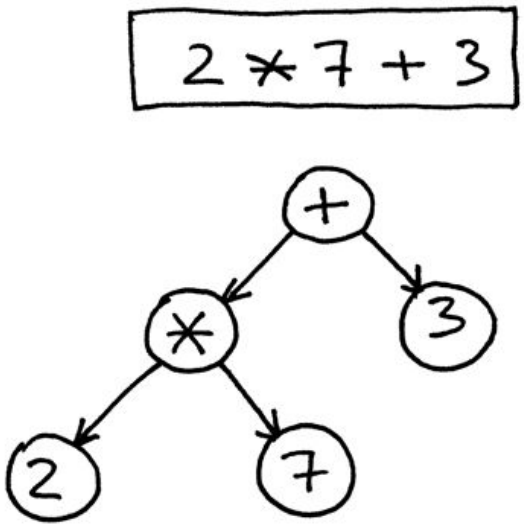
- The compiler parses your code line by line and organizes it into a tree of elements.
- In combination, these elements provide **semantic meaning**.

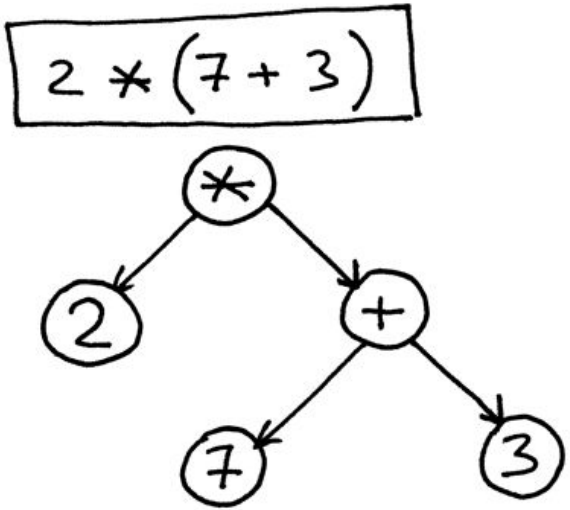
In HW8, you will not need to build this structure yourself.

- You will be implementing different types of `Expr` (via inheritance).



In HW8, each expression is mathematical.





Evaluating Expressions

You will define classes for `Sum`, `Prod`, `Var`, and `Num`.

- These classes can be combined and used to ***evaluate*** expressions.

For example,

- Prod(Sum(Num(5), Num(10)), Num(10)) == 150
- (5 + 10) * 10 == 150

Generically, this same example could just be:

- Expr(Expr(Expr, Expr), Expr)

These expressions are naturally recursive (just like trees in general).
Functions like a calculator that accepts parameters (e.g. X=5, Y=10)

Ex21 due Friday, HW8 due Sunday!

Ex21 is due before the beginning of the next lecture

- Link available on the website: https://courses.cs.washington.edu/courses/cse374/24wi/exercises/

HW8 due Sunday 11.59pm

- Instructions on course website: https://courses.cs.washington.edu/courses/cse374/24wi/homeworks/hw8/

HW9 will be released Sunday and due the following Sunday

- You’re almost to the end! 🎉