CSE 374 Programming Concepts and Tools

Lecture 22 - C++ Inheritance

This Week

Today: C++ Inheritance

Wed: Concurrency

Fri: Exam 3

Today

Review: Smart Pointers

Why inheritance?

Polymorphism

- Dynamic dispatch
- `virtual` functions

Virtual Tables (vtables)

Static Dispatch

HW8 Walkthrough

Review:
Smart Pointers

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!

C++ Inheritance

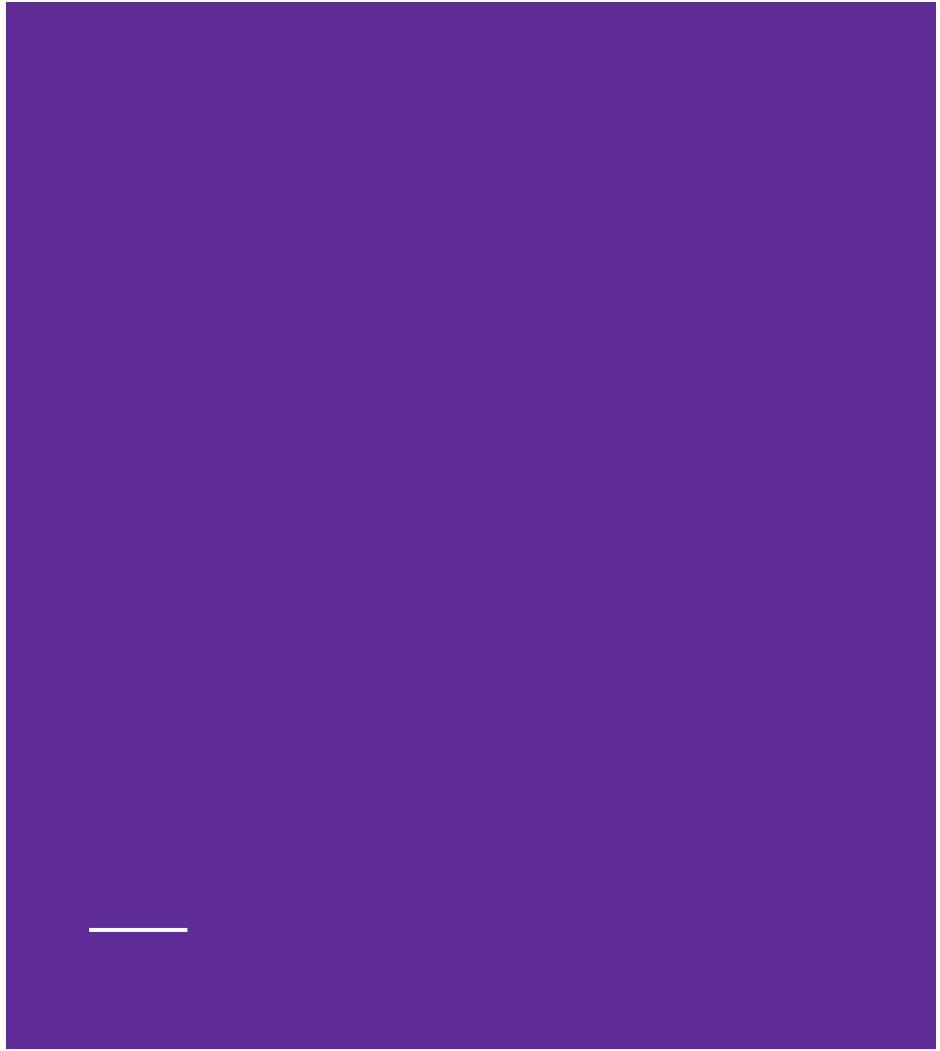
Overview

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*

Motivation

and C++ Syntax



Example: Video Game Sprites

A sprite in a video game represents any object that is being interacted with

- Each sprite has a 2D image associated with it
  - May have different sizes
- Sprites can be used to represent different things:
  - Player characters (PCs): characters that human players control.
  - Non-player characters (NPCs): characters that are a part of the video game and don't have a human player controlling them
  - Enemies or monsters: entities that can damage PCs, and may move around autonomously

Design Without Inheritance

One class per sprite type:

NonPlayerCharacter
name_ image_
GetName() GetImage() Interact()

PlayerCharacter
name_ image_ level_ health_points_ inventory_
GetName() GetImage() GetLevel() GetHP() GetInventory()

Enemy
name_ image_ health_points_ loot_dropped_
GetName() GetImage() GetHP() GetLoot()

- Redundant!
- Cannot treat multiple sprites together
    - i.e., cannot have one array with all three sprite types

Inheritance

A parent-child relationship between classes

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

Terminology:

Java	C++
Superclass	Base Class
Subclass	Derived Class

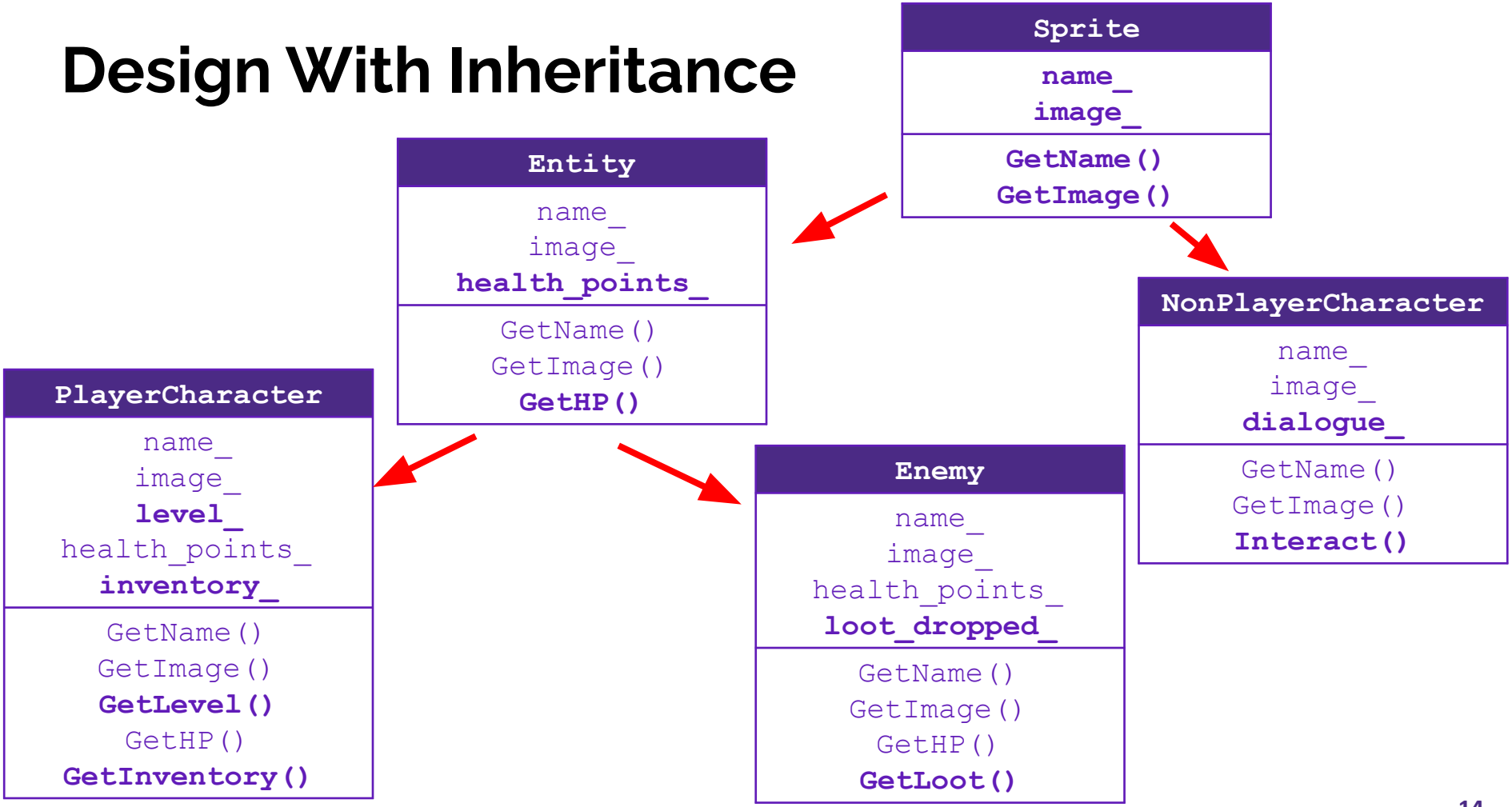
- Java: Subclass inherits from super class. (Superclass is “higher” in the hierarchy)
- C++: Derived class inherits from base class. (Base class is “higher” in the hierarchy)
  - Think of *derived class* as a derivative of the base class (e.g. car class is a derivative of vehicle class)
  - Super/base and sub/derived 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:

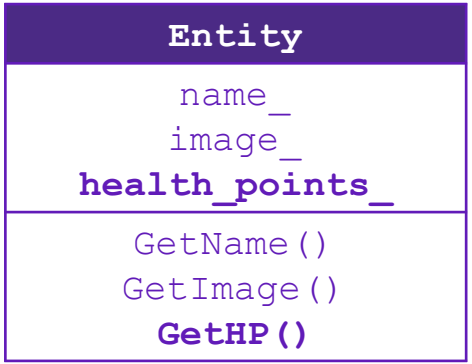
```
#include "BaseClass.h"
class Name : public BaseClass {
  ...
};
```

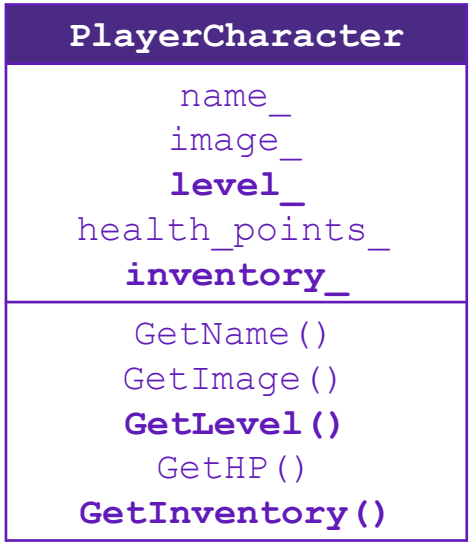
- Focus on single inheritance, but *multiple inheritance* possible
  - `: public Base1, public Base2 {`

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 Sprites

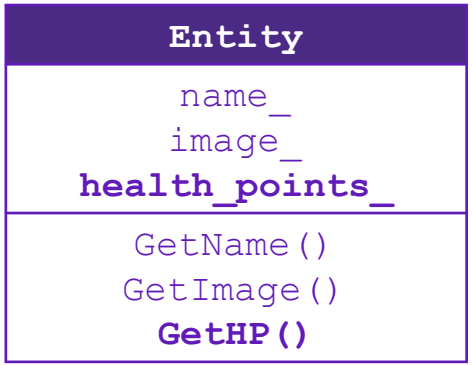


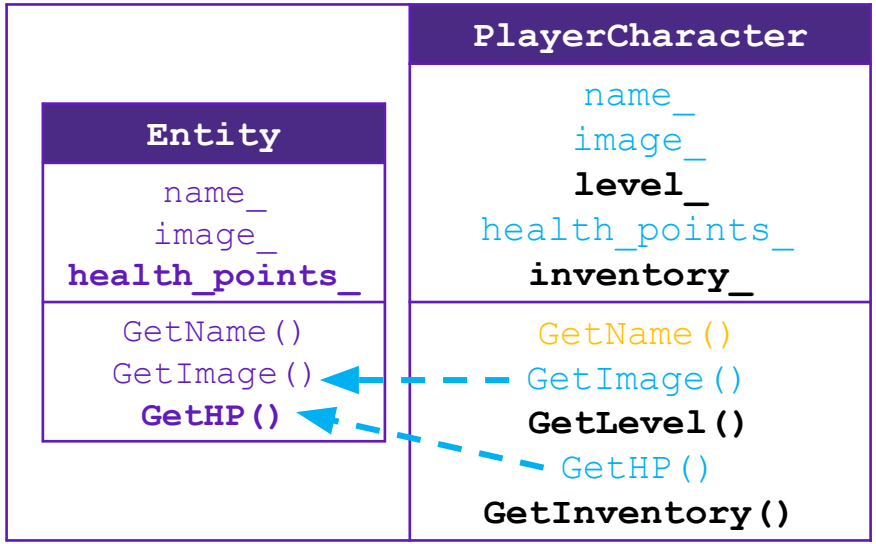


BASE

DERIVED

Back to Sprites



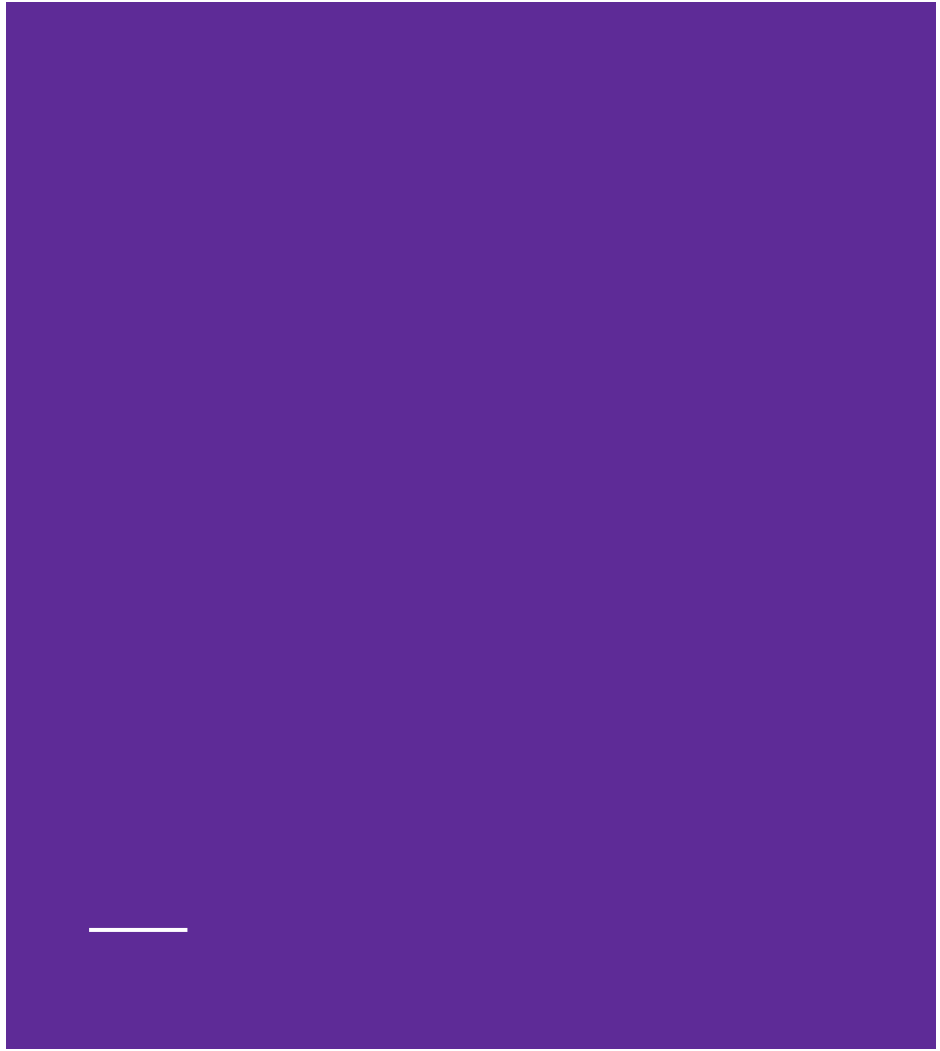


A derived class:

- **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)

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 type or a derived* class of `PromisedType`
- `PromisedType` defines the *interface* (*i.e.* what can be called on `var_p`), but `ActualType` determines which *version* gets invoked

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

- `PromisedType` is the label “cell phone”, `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 an Entity, PlayerCharacter, or Enemy?

Example: `void PrintName(Entity* e) { cout << e->GetName(); }`

- Calls the appropriate `GetName()` without knowing the exact class of `*e`, other than it is some sort of Entity

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 Sprite

```
string PlayerCharacter::GetName() const {
  std::stringstream str;
  str << name_ << "(player)";
  return str.str();
}
```

PlayerCharacter.cc

```
string Sprite::GetName() const {
  return this->name_;
}

void Sprite::Print() const {
  std::cout << "Sprite: " << GetName() << std::endl;
}
```

Sprite.cc

Dynamic Dispatch Example

```
#include "Entity.h"
#include "PlayerCharacter.h"

PlayerCharacter player;
player.set_name("p");
PlayerCharacter* p = &player;
Entity* e = &player; // why is this allowed?

// Invokes PlayerCharacter::GetName()
std::cout << p->GetName() << std::endl;
// Invokes PlayerCharacter::GetName()
std::cout << e->GetName() << std::endl;
// Invokes Sprite::set_name() since that method is inherited
e->set_name("e");
// Invokes Sprite::Print() since that method is inherited,
// which invokes PlayerCharacter::GetName()
e->Print();
```

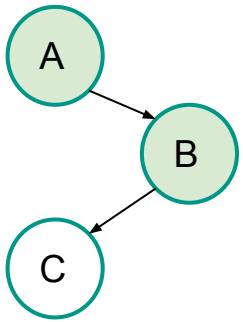
A PlayerCharacter “is a(n)” Entity, and has every part of Entity’s interface

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();
}
```

Has `Foo` definition



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();
}
```

Polling Time!

Please answer in the LP22 assignment on Gradescope!

Which class's **Foo**() is called?

Q1:

- **A**
- **B**
- **C**
- **D**
- **E**

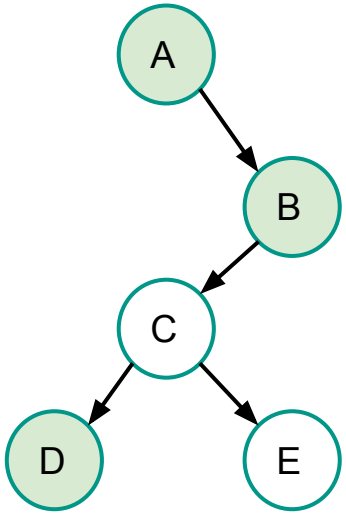
Q2:

- **A**
- **B**
- **C**
- **D**
- **E**

```
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 {
};
```

Polling Question Explained



Has `Foo` definition

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

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

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

How Can This Possibly Work?

The compiler produces `Sprite.o` from *just* `Sprite.cc`

- It doesn't know that `PlayerCharacter` exists during this process
- So then how does the emitted code know to call `Sprite::GetName()` or `PlayerCharacter::GetName()`

  or something else that might not exist yet?

  - **Function pointers!**

```
virtual string GetName() const;
virtual void Print() const;
```

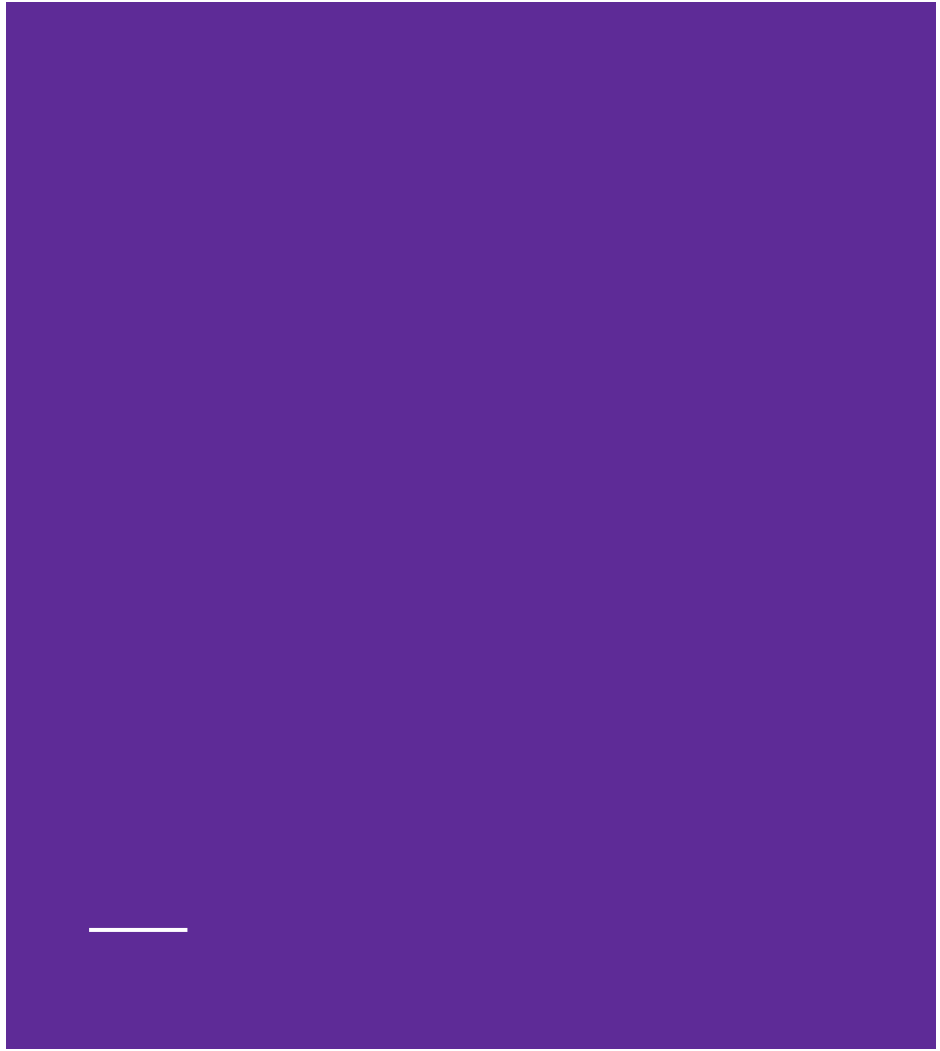
Sprite.h

```
string Sprite::GetName() const {
  return this->name_;
}

void Sprite::Print() const {
  std::cout << "Sprite: " << GetName() << std::endl;
}
```

Sprite.cc

Virtual Tables

Virtual tables & virtual table pointers

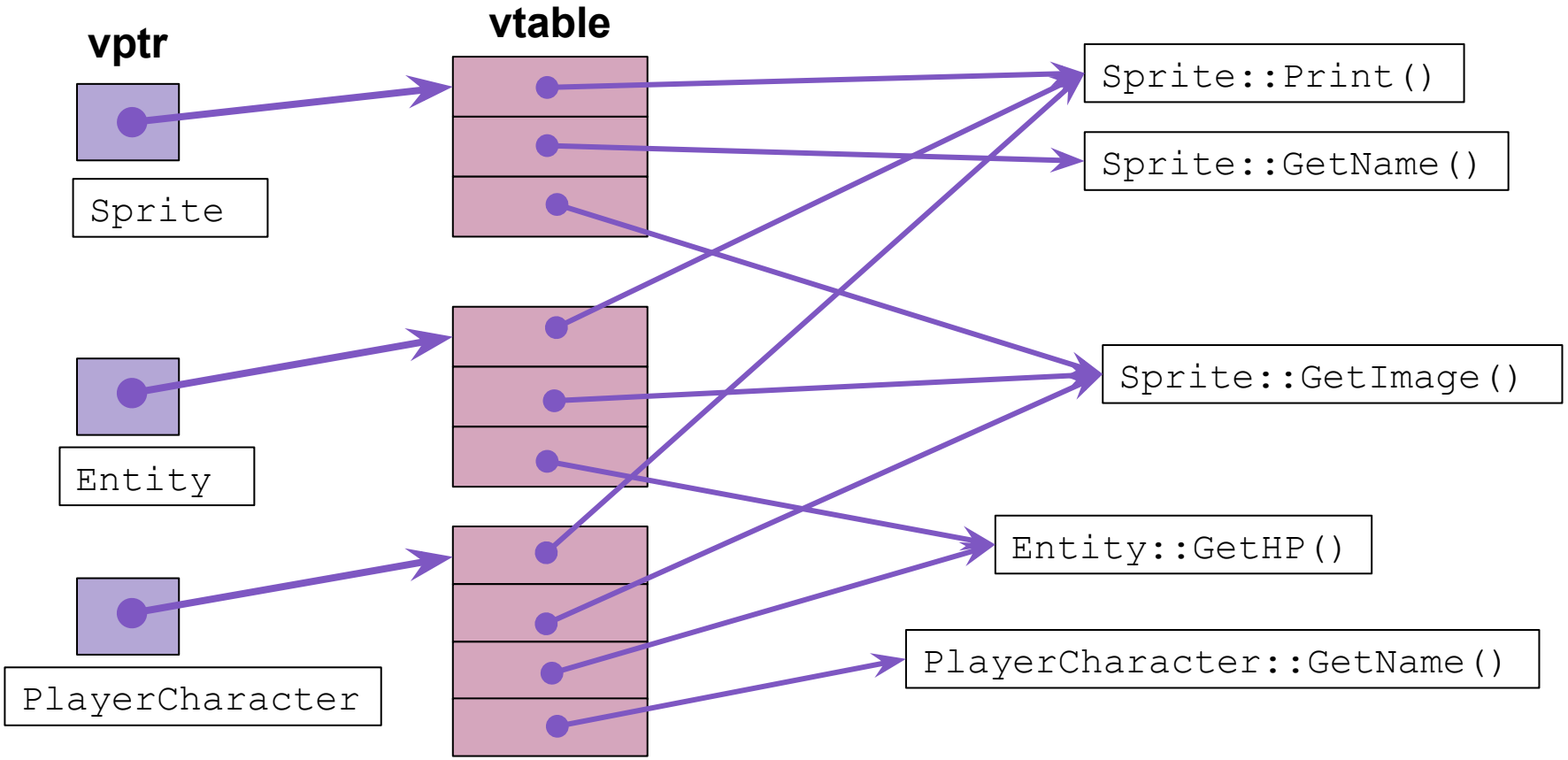


vtables and the vptr

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();
};
```

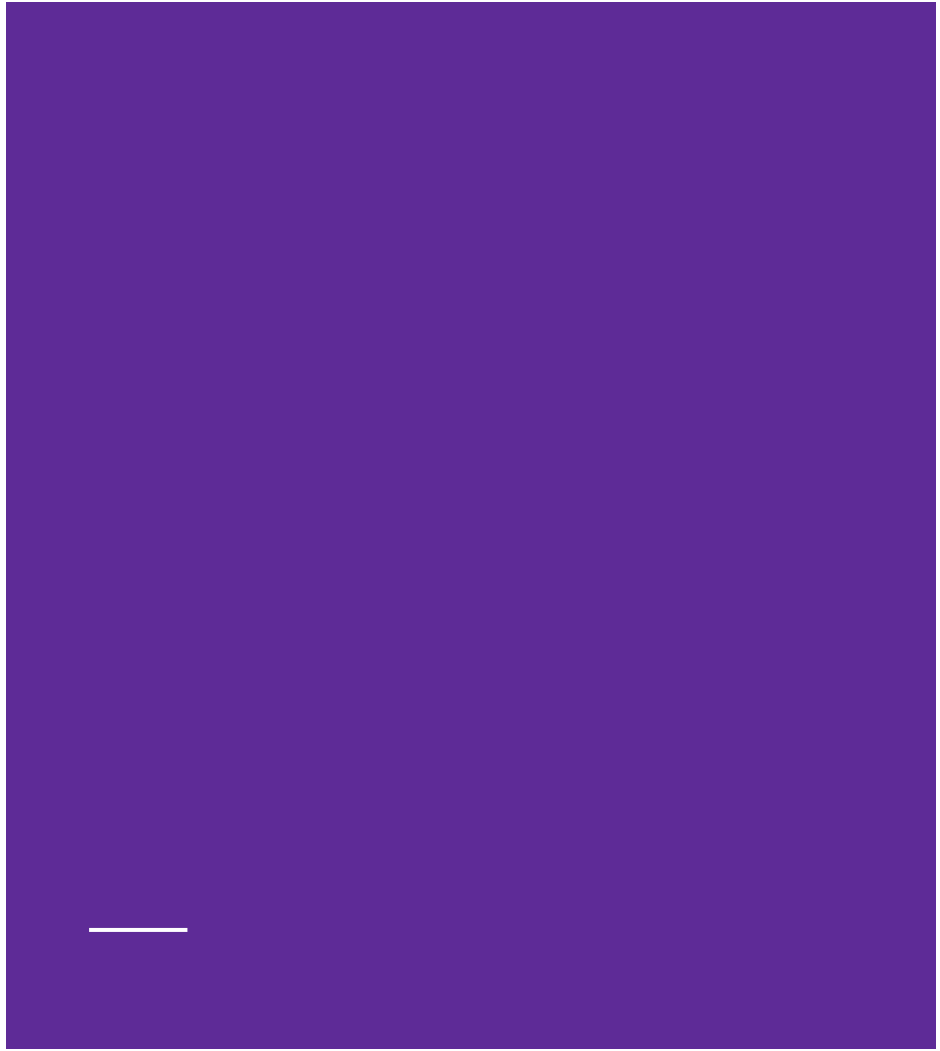
```
Base b;
Der1 d1;
Der2 d2;

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

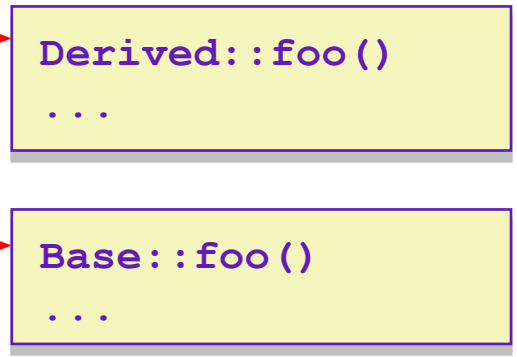


What happens if we omit “virtual”?

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:

Sprite.h

```
string GetName() const;
```

```
PlayerCharacter player;
player.set_name("p");
PlayerCharacter* p = &player;
Entity* e = &player; // why is this allowed?
// Invokes PlayerCharacter::GetName()
std::cout << p->GetName() << std::endl;
// Invokes Sprite::GetName()
std::cout << e->GetName() << std::endl;
// Invokes Sprite::set_name() since that method is inherited
e->set_name("e");
// Invokes Sprite::Print() since that method is inherited,
// which invokes Sprite::GetName()
e->Print();
```

`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 functions 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

Walkthrough: C++ HW (Extra Credit)

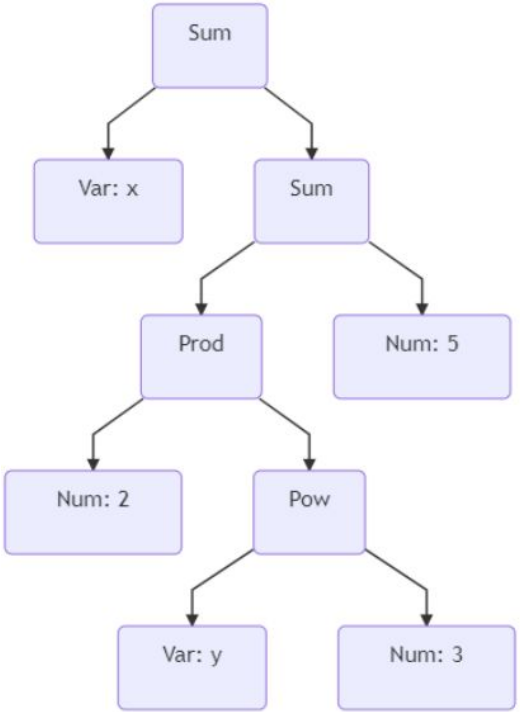
Abstract Syntax Tree (AST)

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

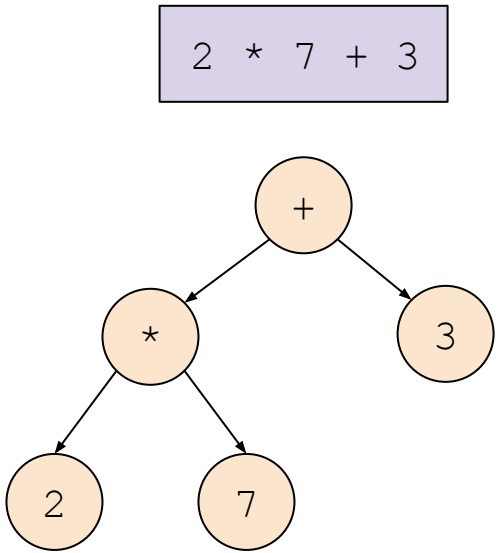
- The compiler parses your code line by line and organizes it into a tree of elements.
- In combination, these elements provide **semantic meaning**.

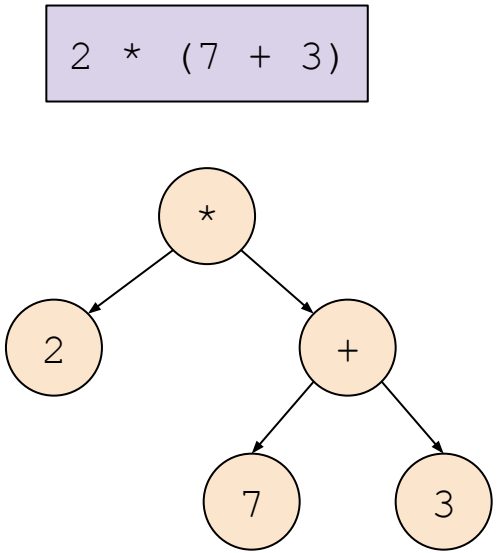
In HW8, you will not need to build this structure yourself.

- 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`)

Assignment Reminders

Exam 3 is this Friday from **10:50-12:50** in **PCAR 291**

- Recommend that you use the 26sp final as a practice exam
- Make your 1 page handwritten cheat sheet!

HW 5 is due on Wednesday, but **late submissions allowed until Sunday @11:59 PM**

HW 6 is extra credit

- You may submit up until Monday, Aug 24 @ 11:59 PM