

COMP6771

Advanced C++ Programming

Week 9

Runtime Polymorphism

Key concepts

- **Inheritance:** ability to create new classes based on existing ones
 - Supported by class derivation
- **Polymorphism:** allows objects of a subclass to be used as if they were objects of a base class
 - Supported via virtual functions
- **Dynamic binding:** run-time resolution of the appropriate function to invoke based on the type of the object
 - Closely related to polymorphism
 - Supported via virtual functions

Tenets of C++

- Don't pay for what you don't use
 - C++ Supports OOP
 - No runtime performance penalty
 - C++ supports generic programming with the STL and templates
 - No runtime performance penalty
 - Polymorphism is extremely powerful, and we need it in C++
 - Do we need polymorphism at all when using inheritance?
 - Answer: sometimes
 - But how do we do so, considering that we don't want to make anyone who doesn't use it pay a performance penalty

Thinking about programming

- Represent concepts with classes
- Represent relations with inheritance or composition
 - **Inheritance:** A is also a B, and can do everything B does
 - "is a" relationship
 - A dog **is an** animal
 - **Composition (data member):** A contains a B, but isn't a B itself
 - "has a" relationship
 - A person **has a** name
 - Choose the right one!

Protected members

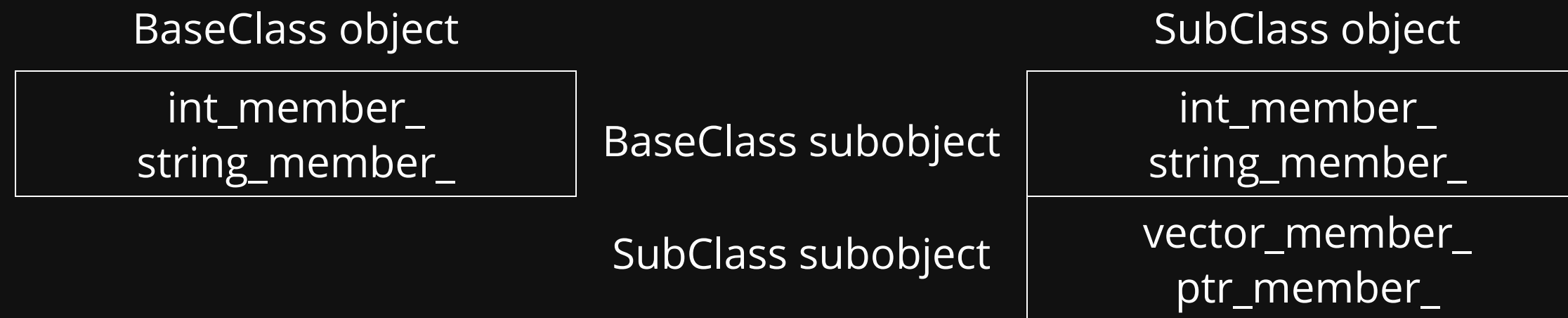
- Protected is a keyword we can use instead of public / private
- Protected members are accessible only to the class, or any subclass of it

Inheritance in C++

- To inherit off classes in C++, we use "class DerivedClass: public BaseClass"
- Visibility can be one of:
 - **public** (generally use this unless you have good reason not to)
 - If you don't want public, you should (usually) use composition
 - **protected**
 - **private**
- Visibility is the maximum visibility allowed
 - If you specify ": private BaseClass", then the maximum visibility is private
 - Any BaseClass members that were public or protected are now private

Inheritance and memory layout

This is very important, as it guides the design of everything we discuss this week



```
1 class BaseClass {
2     public:
3         int get_int_member() { return int_member_; }
4         std::string get_class_name() {
5             return "BaseClass"
6         };
7
8     private:
9         int int_member_;
10        std::string string_member_;
11 }
```

```
1 class SubClass: public BaseClass {
2     public:
3         std::string get_class_name() {
4             return "SubClass";
5         }
6
7     private:
8         std::vector<int> vector_member_;
9         std::unique_ptr<int> ptr_member_;
10 }
```

Polymorphism and values

- How many bytes is a BaseClass instance?
- How many bytes is a SubClass instance?
- One of the guiding principles of C++ is "You don't pay for what you don't use"
 - Let's discuss the following code, but pay great consideration to the memory layout

```
1 class BaseClass {
2     public:
3         int get_member() { return member_; }
4         std::string get_class_name() {
5             return "BaseClass";
6         };
7
8     private:
9         int member_;
10 }
```

```
1 class SubClass: public BaseClass {
2     public:
3         std::string get_class_name() {
4             return "SubClass";
5         }
6
7     private:
8         int subclass_data_;
9 }
```

```
1 void print_class_name(BaseClass base) {
2     std::cout << base.get_class_name()
3               << ' ' << base.get_member()
4               << '\n';
5 }
6
7 int main() {
8     BaseClass base_class;
9     SubClass subclass;
10    print_class_name(base_class);
11    print_class_name(subclass);
12 }
```


The object slicing problem

- If you declare a BaseClass variable, how big is it?
- How can the compiler allocate space for it on the stack, when it doesn't know how big it could be?
- The solution: since we care about performance, a BaseClass can only store a BaseClass, not a SubClass
 - If we try to fill that value with a SubClass, then it just fills it with the BaseClass subobject, and drops the SubClass subobject

```
1 class BaseClass {
2     public:
3         int get_member() { return member_; }
4         std::string get_class_name() {
5             return "BaseClass";
6         };
7
8     private:
9         int member_;
10 }
```

```
1 class SubClass: public BaseClass {
2     public:
3         std::string get_class_name() {
4             return "SubClass";
5         }
6
7     private:
8         int subclass_data_;
9 }
```

```
1 void print_class_name(BaseClass base) {
2     std::cout << base.get_class_name()
3               << ' ' << base.get_member()
4               << '\n';
5 }
6
7 int main() {
8     BaseClass base_class;
9     SubClass subclass;
10    print_class_name(base_class);
11    print_class_name(subclass);
12 }
```

demo901-poly.cpp

Polymorphism and References

- How big is a reference/pointer to a BaseClass
- How big is a reference/pointer to a SubClass
- Object slicing problem solved (but still another problem)
- One of the guiding principles of C++ is "You don't pay for what you don't use"
 - How does the compiler decide which version of GetClassName to call?
 - When does the compiler decide this? Compile or runtime?
 - How can it ensure that calling GetMember doesn't have similar overhead

```
1 class BaseClass {
2     public:
3         int get_member() { return member_; }
4         std::string get_class_name() {
5             return "BaseClass";
6         };
7
8     private:
9         int member_;
10 }
```

```
1 class SubClass: public BaseClass {
2     public:
3         std::string get_class_name() {
4             return "SubClass";
5         }
6
7     private:
8         int subclass_data_;
9 }
```

```
1 void print_class_name(BaseClass& base) {
2     std::cout << base.get_class_name()
3               << ' ' << base.get_member()
4               << '\n';
5 }
6
7 int main() {
8     BaseClass base_class;
9     SubClass subclass;
10    print_class_name(base_class);
11    print_class_name(subclass);
12 }
```

demo902-poly.cpp

Virtual functions

- How does the compiler decide which version of GetClassName to call?
- How can it ensure that calling GetMember doesn't have similar overhead
 - Explicitly tell the compiler that GetClassName is a function designed to be modified by subclasses
 - Use the keyword "virtual" in the base class
 - Use the keyword "override" in the subclass

```
1 class BaseClass {
2     public:
3         int get_member() { return member_; }
4         virtual std::string get_class_name() {
5             return "BaseClass"
6         };
7
8     private:
9         int member_;
10 }
```

```
1 class SubClass: public BaseClass {
2     public:
3         std::string GetClassName() override {
4             return "SubClass";
5         }
6
7     private:
8         int subclass_data_;
9 }
```

```
1 void print_stuff(const BaseClass& base) {
2     std::cout << base.get_class_name()
3               << ' ' << base.get_member()
4               << '\n';
5 }
6
7 int main() {
8     BaseClass base_class;
9     SubClass subclass;
10    print_class_name(base_class);
11    print_class_name(subclass);
12 }
```

Override

- While override isn't required by the compiler, you should **always** use it
- Override fails to compile if the function doesn't exist in the base class. This helps with:
 - Typos
 - Refactoring
 - Const / non-const methods
 - Slightly different signatures

```
1 class BaseClass {
2     public:
3         int get_member() { return member_; }
4         virtual std::string get_class_name() {
5             return "BaseClass"
6         };
7
8     private:
9         int member_;
10 }
```

```
1 class SubClass: public BaseClass {
2     public:
3         // This compiles. But this is a
4         // different function to the
5         // BaseClass get_class_name.
6         std::string get_class_name() const {
7             return "SubClass";
8         }
9
10    private:
11        int subclass_data_;
12 }
```

Virtual functions

So what happens when we start using virtual members?

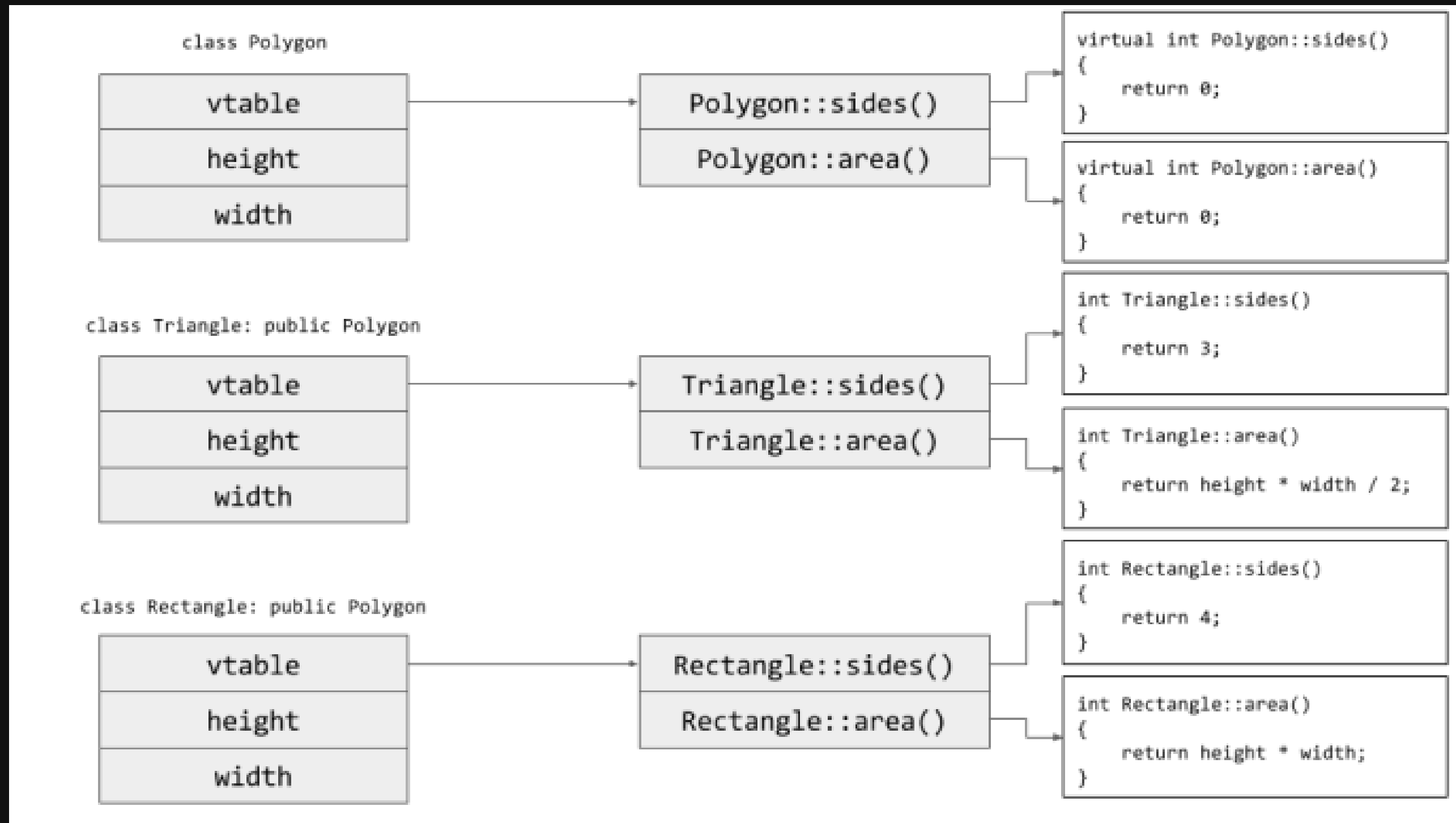
```
1 class BaseClass {
2     public:
3     virtual std::string get_class_name() {
4         return "BaseClass";
5     };
6
7     ~BaseClass() {
8         std::cout << "Destructing base class\n";
9     }
10 }
11
12 class SubClass: public BaseClass {
13     public:
14     std::string get_class_name() override {
15         return "SubClass";
16     }
17
18     ~SubClass() {
19         std::cout << "Destructing subclass\n";
20     }
21 }
```

```
1 void print_stuff(const BaseClass& base_class) {
2     std::cout << base_class.get_class_name()
3         << ' ' << base_class.get_member()
4         << '\n';
5 }
6
7 int main() {
8     auto subclass = static_cast<std::unique_ptr<BaseClass>>(
9         std::make_unique<SubClass>());
10    std::cout << subclass->get_class_name();
11 }
```

VTables

- Each class has a VTable stored in the data segment
 - A vtable is an array of function pointers that says which definition each virtual function points to for that class
- If the VTable for a class is non-empty, then every member of that class has an additional data member that is a pointer to the vtable
- When a virtual function is called **on a reference or pointer type**, then the program actually does the following
 1. Follow the vtable pointer to get to the vtable
 2. Increment by an offset, which is a constant for each function
 3. Follow the function pointer at vtable[offset] and call the function

VTable example



Another example here

Final

- Specifies to the compiler "this is not virtual for any subclasses"
- If the compiler has a variable of type SubClass&, it now no longer needs to look it up in the vtable
- This means static binding if you have a SubClass&, but dynamic binding for BaseClass&

```
1 class BaseClass {
2     public:
3         int get_member() { return member_; }
4         virtual std::string get_class_name() {
5             return "BaseClass"
6         };
7
8     private:
9         int member_;
10 }
```

```
1 class SubClass: public BaseClass {
2     public:
3         std::string get_class_name() override final {
4             return "SubClass";
5         }
6
7     private:
8         int subclass_data_;
9 }
```


Types of functions

| Syntax | Name | Meaning |
|---------------------------|-----------------|------------------------------------------------|
| virtual void fn() = 0; | pure virtual | Inherit interface only |
| virtual void fn() {} | virtual | Inherit interface with optional implementation |
| void fn() {} | nonvirtual | Inherit interface and mandatory implementation |

Note: nonvirtals can be hidden by writing a function with the same name in a subclass

DO NOT DO THIS

Abstract Base Classes (ABCs)

- Might want to deal with a base class, but the base class by itself is nonsense
 - What is the default way to draw a shape? How many sides by default?
 - A function takes in a "Clickable"
- Might want some default behaviour and data, but need others
 - All files have a name, but are reads done over the network or from a disk
- If a class has at least one "abstract" (pure virtual in C++) method, the class is abstract and cannot be constructed
 - It can, however, have constructors and destructors
 - These provide semantics for constructing and destructing the ABC subobject of any derived classes

Pure virtual functions

- Virtual functions are good for when you have a default implementation that subclasses may want to overwrite
- Sometimes there is no default available
- A pure virtual function specifies a function that a class **must** override in order to not be abstract

```
1 class Shape {
2     // Your derived class "Circle" may forget to write this.
3     virtual void draw(Canvas&) {}
4
5     // Fails at link time because there's no definition.
6     virtual void draw(Canvas&);
7
8     // Pure virtual function.
9     virtual void draw(Canvas&) = 0;
10 };
```

Creating polymorphic objects

- In a language like Java, everything is a pointer
 - This allows for code like on the left
 - Not possible in C++ due to objects being stored inline
 - This then leads to slicing problem
- If you want to store a polymorphic object, use a pointer

```
1 // Java-style C++ here
2 // Don't do this.
3
4 auto base = std::vector<BaseClass>();
5 base.push_back(BaseClass{});
6 base.push_back(SubClass1{});
7 base.push_back(SubClass2{});
```

```
1 // Good C++ code
2 // But there's a potential problem here.
3 // (*very* hard to spot)
4
5 auto base = std::vector<std::unique_ptr<BaseClass>>();
6 base.push_back(std::make_unique<BaseClass>());
7 base.push_back(std::make_unique<Subclass1>());
8 base.push_back(std::make_unique<Subclass2>());
```

Inheritance and constructors

- Every subclass constructor must call a base class constructor
 - If none is manually called, the default constructor is used
 - A subclass cannot initialise fields defined in the base class
 - Abstract classes must have constructors

```
1 class BaseClass {
2     public:
3         BaseClass(int member): int_member_{member} {}
4
5     private:
6         int int_member_;
7         std::string string_member_;
8 }
9
10 class SubClass: public BaseClass {
11     public:
12         SubClass(int member, std::unique_ptr<int>&& ptr): BaseClass(member), ptr_member_(std::move(ptr)) {}
13         // Won't compile.
14         SubClass(int member, std::unique_ptr<int>&& ptr): int_member_(member), ptr_member_(std::move(ptr)) {}
15
16     private:
17         std::vector<int> vector_member_;
18         std::unique_ptr<int> ptr_member_;
19 }
```

Destructing polymorphic objects

- Which constructor is called?
- Which destructor is called?
- What could the problem be?
 - What would the consequences be?
- How might we fix it, using the techniques we've already learnt?

```
1 // Simplification of previous slides code.  
2  
3 auto base = std::make_unique<BaseClass>();  
4 auto subclass = std::make_unique<Subclass>();
```

Destructing polymorphic objects

- Whenever you write a class intended to be inherited from, **always** make your destructor virtual
- Remember: When you declare a destructor, the move constructor and assignment are not synthesized

```
1 class BaseClass {  
2     BaseClass(BaseClass&&) = default;  
3     BaseClass& operator=(BaseClass&&) = default;  
4     virtual ~BaseClass() = default;  
5 }
```

Forgetting this can be a hard bug to spot

Static and dynamic types

- Static type is the type it is declared as
- Dynamic type is the type of the object itself
- Static means compile-time, and dynamic means runtime
 - Due to object slicing, an object that is neither reference or pointer **always** has the same static and dynamic type

Quiz - What's the static and dynamic types of each of these?

```
1 int main() {
2     auto base_class = BaseClass();
3     auto subclass = SubClass();
4     auto sub_copy = subclass;
5     // The following could all be replaced with pointers
6     // and have the same effect.
7     const BaseClass& base_to_base{base_class};
8     // Another reason to use auto - you can't accidentally do this.
9     const BaseClass& base_to_sub{subclass};
10    // Fails to compile
11    const SubClass& sub_to_base{base_class};
12    const SubClass& sub_to_sub{subclass};
13    // Fails to compile (even though it refers to at a sub);
14    const SubClass& sub_to_base_to_sub{base_to_sub};
15 }
```


Static and dynamic binding

- Static binding: Decide which function to call at compile time (based on static type)
- Dynamic binding: Decide which function to call at runtime (based on dynamic type)
- C++
 - Statically typed (types are calculated at compile time)
 - Static binding for non-virtual functions
 - Dynamic binding for virtual functions
- Java
 - Statically typed
 - Dynamic binding

Up-casting

- Casting from a derived class to a base class is called up-casting
- This cast is always safe
 - All dogs are animals
- Because the cast is always safe, C++ allows this as an implicit cast
- One of the reasons to use auto is that it avoids implicit casts

```
1 auto dog = Dog();  
2  
3 // Up-cast with references.  
4 Animal& animal = dog;  
5 // Up-cast with pointers.  
6 Animal* animal = &dog;
```

Down-casting

- Casting from a base class to a derived class is called down-casting
- This cast is not safe
 - Not all animals are dogs

```
1  auto dog = Dog();
2  auto cat = Cat();
3  Animal& animal_dog{dog};
4  Animal& animal_cat{cat};
5
6  // Attempt to down-cast with references.
7  // Neither of these compile.
8  // Why not?
9  Dog& dog_ref{animal_dog};
10 Dog& dog_ref{animal_cat};
```

How to down cast

- The compiler doesn't know if an Animal happens to be a Dog
 - If you **know** it is, you can use **static_cast**
 - Otherwise, you can use **dynamic_cast**
 - Returns null pointer for pointer types if it doesn't match
 - Throws exceptions for reference types if it doesn't match

```
1 auto dog = Dog();
2 auto cat = Cat();
3 Animal& animal_dog{dog};
4 Animal& animal_cat{cat};
5
6 // Attempt to down-cast with references.
7 Dog& dog_ref{static_cast<Dog&>(animal_dog)};
8 Dog& dog_ref{dynamic_cast<Dog&>(animal_dog)};
9 // Undefined behaviour (incorrect static cast).
10 Dog& dog_ref{static_cast<Dog&>(animal_cat)};
11 // Throws exception
12 Dog& dog_ref{dynamic_cast<Dog&>(animal_cat)};
```

```
1 auto dog = Dog();
2 auto cat = Cat();
3 Animal& animal_dog{dog};
4 Animal& animal_cat{cat};
5
6 // Attempt to down-cast with pointers.
7 Dog* dog_ref{static_cast<Dog*>(&animal_dog)};
8 Dog* dog_ref{dynamic_cast<Dog*>(&animal_dog)};
9 // Undefined behaviour (incorrect static cast).
10 Dog* dog_ref{static_cast<Dog*>(&animal_cat)};
11 // returns null pointer
12 Dog* dog_ref{dynamic_cast<Dog*>(&animal_cat)};
```

Covariants

- Read more about covariance and contravariance
- **If a function overrides a base, which type can it return?**
 - If a base specifies that it returns a LandAnimal, a derived also needs to return a LandAnimal
- Every possible return type for the derived must be a valid return type for the base

```
1 class Base {
2     virtual LandAnimal& get_favorite_animal();
3 };
4
5 class Derived: public Base {
6     // Fails to compile: Not all animals are land animals.
7     Animal& get_favorite_animal() override;
8     // Compiles: All land animals are land animals.
9     LandAnimal& get_favorite_animal() override;
10    // Compiles: All dogs are land animals.
11    Dog& get_favorite_animal() override;
12 };
```

Contravariants

- **If a function overrides a base, which types can it take in?**
 - If a base specifies that it takes in a LandAnimal, a LandAnimal must always be valid input in the derived
- Every possible parameter to the base must be a possible parameter for the derived

```
1 class Base {
2     virtual void use_animal(LandAnimal&);
3 };
4
5 class Derived: public Base {
6     // Compiles: All land animals are valid input (animals).
7     void use_animal(Animal&) override;
8     // Compiles: All land animals are valid input (land animals).
9     void use_animal(LandAnimal&) override;
10    // Fails to compile: Not All land animals are valid input (dogs).
11    void use_animal(Dog&) override;
12 };
```

Default arguments and virtuals

- Default arguments are determined at compile time for efficiency's sake
- Hence, default arguments need to use the **static** type of the function
- Avoid default arguments when overriding virtual functions

```
1 class Base {
2 public:
3     virtual ~Base() = default;
4     virtual void print_num(int i = 1) {
5         std::cout << "Base " << i << '\n';
6     }
7 };
8
9 class Derived: public Base {
10 public:
11     void print_num(int i = 2) override {
12         std::cout << "Derived " << i << '\n';
13     }
14 };
15
16 int main() {
17     Derived derived;
18     Base* base = &derived;
19     derived.print_num(); // Prints "Derived 2"
20     base->print_num(); // Prints "Derived 1"
21 }
```

Construction of derived classes

- Base classes are always constructed before the derived class is constructed
 - The base class ctor never depends on the members of the derived class
 - The derived class ctor may be dependent on the members of the base class

```
1 class Animal {...}
2 class LandAnimal: public Animal {...}
3 class Dog: public LandAnimals {...}
4
5 Dog d;
6
7 // Dog() calls LandAnimal()
8 // LandAnimal() calls Animal()
9 // Animal members constructed using initialiser list
10 // Animal constructor body runs
11 // LandAnimal members constructed using initialiser list
12 // LandAnimal constructor body runs
13 // Dog members constructed using initialiser list
14 // Dog constructor body runs
```


Virtuals in constructors

If a class is not fully constructed, cannot perform dynamic binding

```
1 class Animal {...};
2 class LandAnimal: public Animal {
3     LandAnimal() {
4         Run();
5     }
6
7     virtual void Run() {
8         std::cout << "Land animal running\n";
9     }
10 };
11 class Dog: public LandAnimals {
12     void Run() override {
13         std::cout << "Dog running\n";
14     }
15 };
16
17 // When the LandAnimal constructor is being called,
18 // the Dog part of the object has not been constructed yet.
19 // C++ chooses to not allow dynamic binding in constructors
20 // because Dog::Run() might depend upon Dog's members.
21 Dog d;
```

Destruction of derived classes

Easy to remember order: Always opposite to construction order

```
1 class Animal {...}
2 class LandAnimal: public Animal {...}
3 class Dog: public LandAnimals {...}
4
5 auto d = Dog();
6
7 // ~Dog() destructor body runs
8 // Dog members destroyed in reverse order of declaration
9 // ~LandAnimal() destructor body runs
10 // LandAnimal members destroyed in reverse order of declaration
11 // ~Animal() destructor body runs
12 // Animal members destroyed in reverse order of declaration.
```

Virtuals in destructors

- If a class is partially destructed, cannot perform dynamic binding
- Unrelated to the destructor itself being virtual

```
1 class Animal {...};
2 class LandAnimal: public Animal {
3     virtual ~LandAnimal() {
4         Run();
5     }
6
7     virtual void Run() {
8         std::cout << "Land animal running\n";
9     }
10 };
11 class Dog: public LandAnimals {
12     void Run() override {
13         std::cout << "Dog running\n";
14     }
15 };
16
17 // When the LandAnimal constructor is being called,
18 // the Dog part of the object has already been destroyed.
19 // C++ chooses to not allow dynamic binding in destructors
20 // because Dog::Run() might depend upon Dog's members.
21 auto d = Dog();
```

Feedback

