COMP6771
Advanced C++ Programming

7.1 Dynamic Polymorphism

Key Concepts

- Inheritance
  - To be able to create new classes by inheriting from existing classes.
  - To understand how inheritance can promote software reusability.
  - To understand the notions of base classes and derived classes.

- Polymorphism
  - Dynamic: determine which method to call at run-time
  - Static: determine which method to call at compile-time
  - Considerations of Dynamic Polymorphism

Use Cases for Dynamic Polymorphism

- Dynamic polymorphism allows for an “open” type system.
  - Types can be extended with new types.
  - Types can be customised beyond what the original author imagined.
- Natural fit when types have an “is-a” relationship.
- Can reduce code duplication for extremely similar types.

```cpp
#include <iostream>
#include <vector>

struct mammal {
  virtual auto speak() const -> void;
};

struct dog : mammal {
auto speak() const -> void override {
  std::cout << "wan" << std::endl;
  }
};

struct cat : mammal {
auto speak() const -> void override {
  std::cout << "nyaa" << std::endl;
  }
};

int main() {
  dog d;
  cat c;
  std::vector<mammal *> v = {&d, &c};
  for (const mammal *a : v) a->speak();
}

// Output:
// wan
// nyaa
```

Dynamic Polymorphism in C++

- Dynamic polymorphism is achieved by augmenting `classes` and `structs` (and sometimes `unions`).
- Use of **inheritance** to share interface/implementation between a parent class and child classes.
- Use of new keywords `virtual`, `override`, `final` to implement overriding member functions.
- `dynamic_cast<>` to safely cast types up and down the type hierarchy.
- First let's review some classic Object-Oriented Programming concepts.

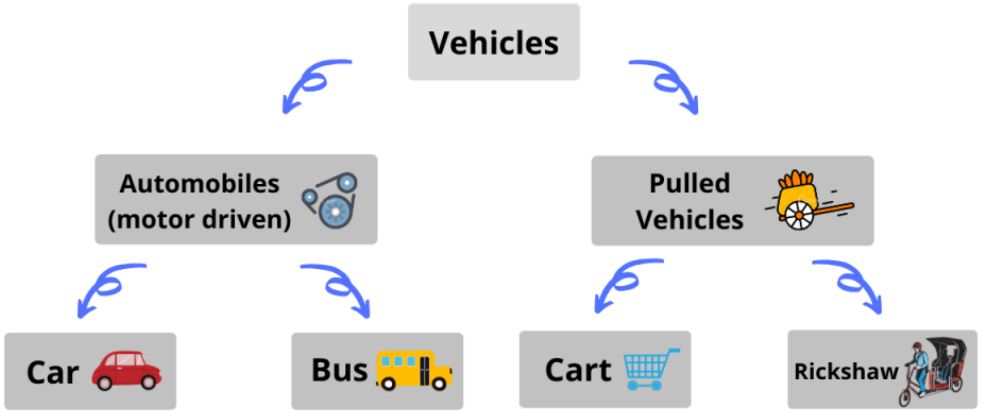
OOP: Composition

- Main idea of OOP: represent concepts as class-types.
- **Composition:** A contains a B but isn't a B itself.
  - ”Has-a" relationship.
  - A person **has a** name;
  - A car **has a** battery;
  - Etc.
- Solves the problem of code duplication by keeping interfaces separate
  - Classes are coupled together by containment, however

```cpp
class wheel {};

class car {
public:
  /* Implementation */

private:
  wheel tl_;
  wheel tr_;
  wheel bl_;
  wheel br_;
};

// A car “has” four wheels
// delegate all operations that require
// wheels to the wheel objects themselves.
```

OOP: Inheritance

- Main idea of OOP: represent concepts as class-types.
- **Inheritance:** A is a B and can do everything B does.
  - "is-a" relationship.
  - A dog **is an** animal.
  - A teacher **is an** employee.
  - Etc.
- Solve the problem of code duplication by sharing implementation and interface.
  - Can lead to incredibly difficult-to-manage type hierarchies, however.



Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

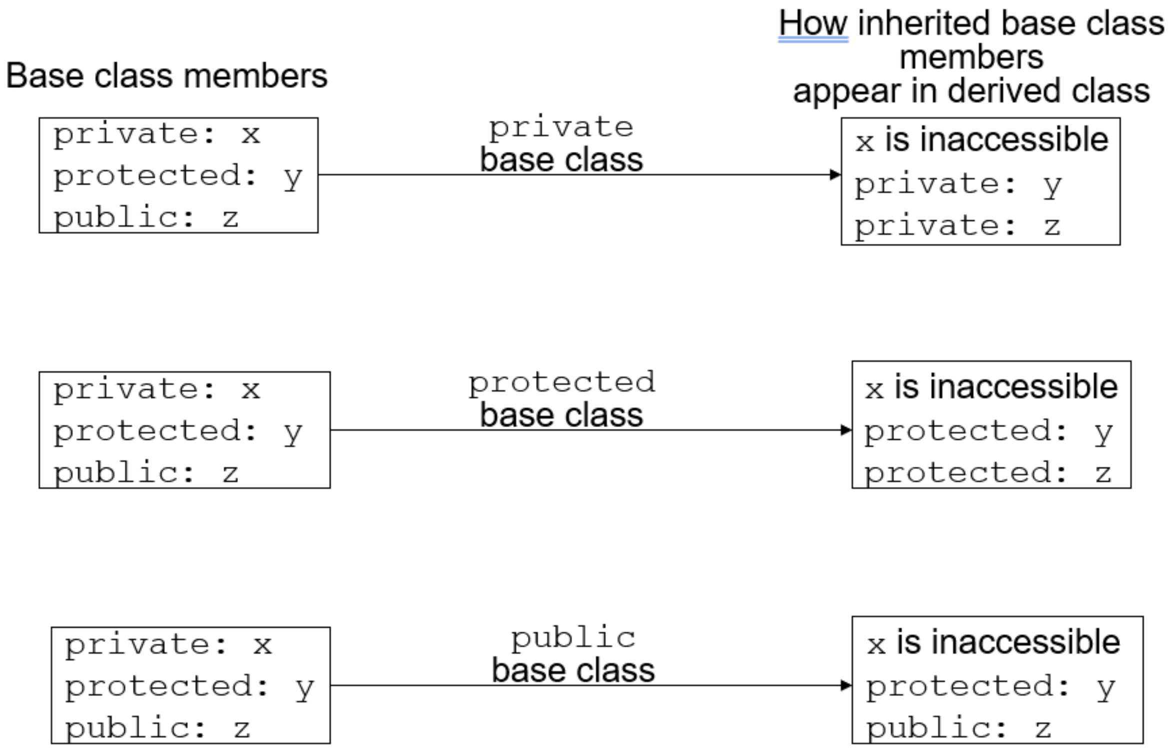
Inheritance in C++

- C++ supports multiple inheritance for all class-types.
  - Construction, destruction, and member look-up rules change when at least one base class exists.
- Inheritance kind depends on the **inheritance access specifier.**
- Implementation vs. Interface inheritance exists.
- Base class / derived class relationship expressed through inheritance.

```cpp
#include <iostream>
#include <string>

struct hello {
  std::string msg1 = "hello!";
};

struct world {
  std::string msg2 = "world!";
};

class child : public hello, public world {
public:
  auto greeting() const -> void {
    std::cout <<
    msg1 + " " + msg2 <<
    std::endl;
  }
};

int main() {
  // prints "hello world!"
  child{}.greeting();
}
```

Inheritance: Member Visibility

- Visibility is the maximum access exported from the deriving class.
- Visibility can be one of:
  - **public**
    - objects of the derived class can be treated as an object of the base class (generally use this unless you have good reason not to)
    - If you don't want public, you should (usually) use composition
  - **protected**
    - Derived class gains access to public and protected members of the parent
  - **private**
    - only accessible in the derived class.



Inheritance: Default Visibility

- The default inheritance visibility for classes is **private**.
- The default inheritance visibility for structs is **public**.

```
struct base {
 int i;
};

class derived_class : base {
  // base was derived from privately.
  // this means that all non-private members
  // in base are now “private” inside
  // of derived_class
};

struct derived_struct : base {
  // base was derived from publically.
  // this means that all public members
  // in base are still “public” in
  // derived_struct, but ”protected” members
  // are still “protected” in derived_struct
  // as well.
};
```

Inheritance: Member Access

- It is possible after inheritance for a class-type to have multiple members with the same name.
- In that case, can use the scope operator (`::`) to access the member of the specific class.
- If a member name is used *unqualified* and there is a collision:
  - For member functions with the same parameter list, will use the most derived class's overload if it exists.
  - Otherwise, if two base classes have the same method, then the call is ambiguous.
  - For data members, name collisions are always ambiguous.

```cpp
#include <iostream>
struct uphill {
  int jack;
  int jill;

  void foo() {}
  void baz() {}
};

struct beanstalk {
  double jack;
  void baz();
};

struct two_worlds_collide : uphill, beanstalk {
  void foo() {
    // ERROR: jack is ambiguous
    std::cout << jack << std::endl;
  }

  void bar() {
    // OK: using uphill's jack
    std::cout << uphill::jack << std::endl;
    foo(); // calls two_worlds_collide::foo()
    baz(); // ERROR: call is ambiguous
  }
};
```

Inheritance: Interface vs. Implementation

- Interface inheritance is when only the interface of methods are intended to be inherited.
  - Does not mean implementation is not inherited also.
- Implementation inheritance is when the implementation are intended to be inherited.
- Inheritance kind depends on the **inheritance access specifier.**
  - Implementation inheritance uses *private* inheritance.
  - Anything other than private is interface inheritance.

```cpp
#include <iostream>>
struct base {
  auto greeting() const -> void {
    std::cout << msg << std::endl;
  }

  std::string msg = "hi!";
};

struct interface : public base {
  // no need to remake greeting()
};

struct implementation : private base {
  // need to re-create the greeting method
  auto greeting() const -> void {
  std::cout << msg + " world!\n";
  }
};

int main() {
  interface{}.greeting(); // prints "hi!"
  implementation{}.greeting(); // prints "hi, world!"
}
```

Inheritance: Construction

- Construction order changes when there is at least one base class.
- Each base class is constructed first (in order of inheritance), then `this`'s data members, then the body of the constructor is run.
  - If a base class's constructor is not specified, then the default constructor is used.
  - A derived class cannot initialise fields in the base class.

```cpp
#include <iostream>

struct A { A() { std::cout << "A "; } };

struct B : A { B() { std::cout << "B "; } };

struct C : A { C() { std::cout << "C "; } };

struct alphabet : B, A, C {
  alphabet() : c{}, a{} {
    std::cout << "alphabetical";
  }

  C c;
  B b;
  A a;
};

int main() {
  { alphabet alpha; }
}

// output:
// A B A A C A C A B A alphabetical
```

Inheritance: Destruction

- Destruction order changes when there is at least one base class.
- First, `this`'s destructor runs.
- Then, the destructors of `this`'s data members run in the reverse order of declaration.
- Then, the destructors of this's base classes run in the reverse order of inheritance.

```
#include <iostream>

struct A { ~A() { std::cout << "A\n"; } };

struct B : A { ~B() { std::cout << "B "; } };

struct C : A { ~C() { std::cout << "C "; } };

struct alphabet : A, B, C {
  ~alphabet() {
    std::cout << "alphabetical ";
  }

  B b;
  C c;
};

int main() {
  { alphabet alpha; }
}

// output:
// alphabetical C A B A C A B A A
```

C++ Object Memory Layout

- Very important to understand how objects are laid out in memory.
- Base classes are laid out first (in order of inheritance).
- Then `this` is laid out.
- For deep type hierarchies, `this` may have data members which are very far apart from a base class's data members.

```cpp
struct b1 { double d; };
struct b2: b1 { unsigned ss[2]; };
struct d0 : b2 { int is[2]; };

d0 d = {};
```

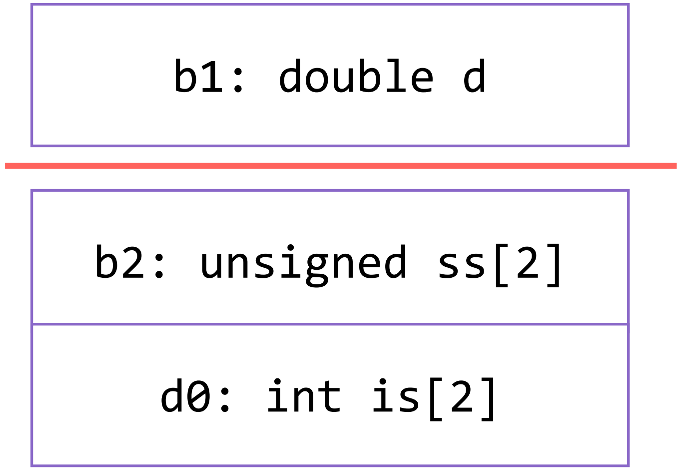
`b1: double d`
`b2: unsigned ss[2]`
`d0: int is[2]`

d's memory layout

Object-Slicing Problem

- If a b1 variable is declared on the stack *by value* how big should it be?
  - The compiler only knows its static type!
- When storing a class with base classes by value, classes lower in the hierarchy will not be stored since the compiler doesn't know about their existence.
- This is called the "object-slicing problem".

```cpp
struct b1 { double d; };
struct b2: b1 { unsigned ss[2]; };
struct d0 : b2 { int is[2]; };

// SLICE! Compiler only copied the b1 part of d0{}
b1 b = d0{};
```



b's memory layout.

You can see how the non-b1 parts are sliced off when copying.

Polymorphic Classes

- **Polymorphism** means that a call to a member function will cause a different function to be executed depending on the runtime type of the object.
- Polymorphism allows reuse of code by allowing objects of related types to be treated the same.
- Polymorphism in C++:
  - Static (compile-time) type vs. dynamic (runtime) type.
  - Overridable methods through the `virtual`, `override`, and `final` keywords.
  - Due to the Object Slicing Problem only **pointers** and **references** to classes can exhibit polymorphic behaviour.

Static vs. Runtime Type of Polymorphic Classes

- Static type is the type it is declared as in the source code.
- Dynamic type is the type of the object at run-time.
- Due to object slicing, value objects **always** have the same static and dynamic type.

```cpp
struct base {};
struct derived : base {};

int main() {
  auto b = base{};
  auto d = derived{};

  base sliced = d; // not good, don't do this!

  // The following could all be replaced with
  // pointers and have the same effect.
  const base &base2base = b;

  // A potential reason to use auto:
  // you can't accidentally do this.
  const base &base2derived = d;

  // Fails to compile
  const derived &derived2base = b;

  const derived &derived2derived = d; // OK!

  // Fails to compile despite a ref to a derived class
  const derived &derived2base2derived = base2derived;
}
```

Static vs. Runtime Binding

- Static binding: Decide which function to call at compile time.
  - Based on static type in the source code.
- Dynamic binding: Decide which function to call at runtime.
  - Based on dynamic type.
- C++ is by default statically typed.
  - Types are specified at compile time.
  - Static binding for non-virtual functions.
  - Dynamic binding for virtual functions.
- Very different from almost all other languages that support OOP!

`virtual` Methods

- How does the compiler know which method to call in a polymorphic class?
- Explicitly tell compiler that a method is meant to be overridden in a subclass.
- Use the `virtual` keyword in the base class:
  - For methods in the base class that expect to be overridden in derived class.
  - Dynamic binding: actual function to call bound at runtime by the compiler.
  - It ensures that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
  - Static binding: Without virtual member functions, actual function to call determined at compile-time.
- Use the `override` keyword in the derived class

```cpp
#include <iostream>

struct cat {
Virtual void speak() const {
    std::cout << "meow\n";
  }
};

struct garfield : cat {
  void speak() const override {
    std::cout << "I want lasagne!\n";
  }
};

int main() {
  garfield g;
  cat c;

  const cat &cg = g;
  const cat &cc = c;

  cg.speak(); // prints "I want lasagne!"
  cc.speak(); // prints "meow"
}

// though cg and cc have the same static type,
// at runtime their types are different.
// The correct function to call is looked up
// at runtime
```

The `override` Keyword

- Tells the compiler this method overrides a virtual function in a base class.
- 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 catching errors related to:
  - Refactoring / typos.
  - `const` / non-`const` methods.
  - Slightly different signatures.

```
struct character {
  virtual int power() const { return 6771; }
};

struct guardian : character {
  // ERROR: this method does not override a
  // virtual method in `character`
  // (missing const qualification)
  int power() override { return 42; }
};

struct vegeta : character {
  // OK: matches int power() const in `character`
  int power() const override { return 9001; }
};
```

The `final` Keyword

- Specifies to the compiler that a method cannot be overridden in a derived class.
  - This means static binding if you have a reference/pointer to a derived class, but dynamic binding for a base class reference/pointer.
- Also specifies to the compiler that a class-type cannot be derived from.

```cpp
struct top {
  virtual void greet() const final {
    std::cout << "hello, world!" << std::endl;
  }
};

struct final middle : top {
  // ERROR: cannot override `greet`:
  // it has declared as final
  void greet() const override {
    std::cout << "heyyy" << std::endl;
  }
};

// ERROR: cannot derive from `middle`:
// it has been declared as "final"
struct bottom : middle {};
```

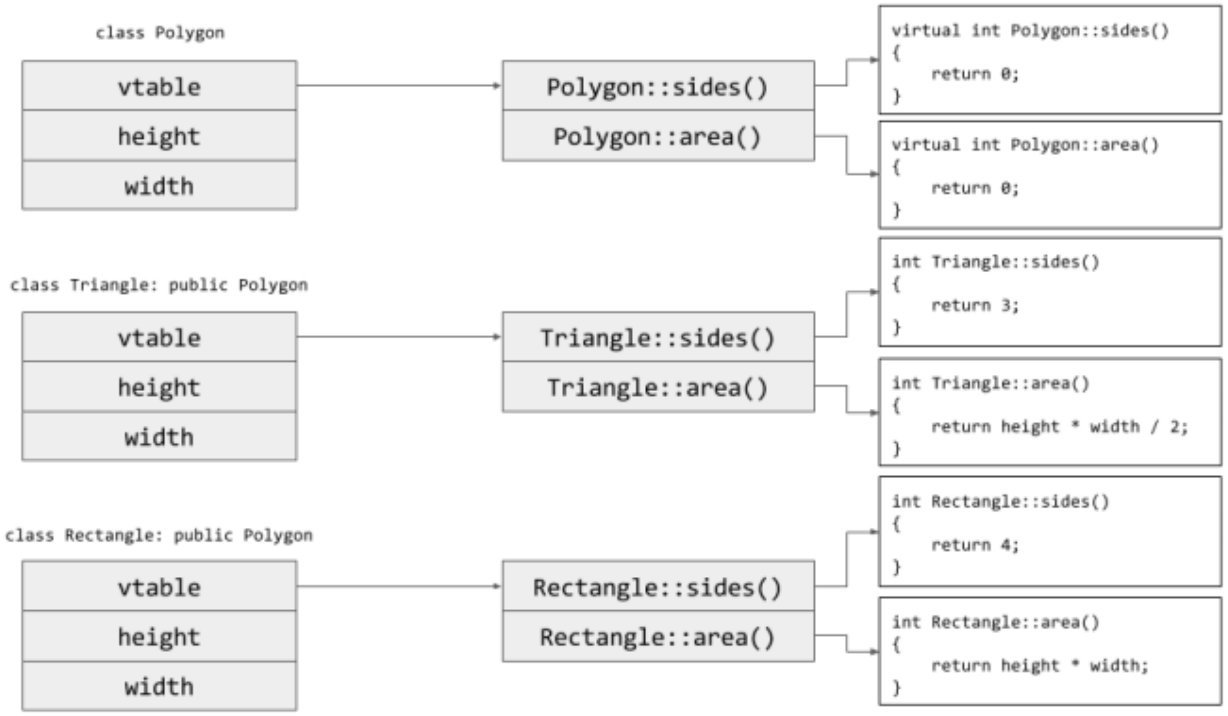
`virtual` Methods & Default Arguments

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

```
struct base {
  virtual void print_num(int i = 1) {
    std::cout << "Base " << i << '\n';
  }
};

struct derived: base {
  void print_num(int i = 2) override {
    std::cout << "Derived " << i << '\n';
  }
};

int main() {
  derived d;
  base *b = &d;

  // Prints "Derived 2"
  d.print_num();

  // Prints "Derived 1" because the default argument
  // was chosen due to b's static type (base)
  // though the actual function to call was chosen due
  // to b's dynamic type (derived)
  b->print_num();
}
```

How `virtual` works (VTables)

- Each polymorphic class has a `vtable` stored in the text segment of the binary.
  - A `vtable` is an array of function pointers.
  - Compiler hashes the names of virtual methods and stores a pointer to this class's *specific* implementation of that virtual or copies the parent's corresponding function pointer if not overridden.
- If the `vtable` for a class is non-empty, then every member of that class has an implicit data member that is a pointer to the `vtable`.
- When a virtual function is called **on a reference or pointer type**, then the program does the following:
  - Follow the `vtable` pointer to get to the `vtable` .
  - Increment by an offset (calculated by the compiler), which is a constant.
  - Call the function pointer pointed to by `vtable[offset]`.



Another example here

Constructing Polymorphic Objects

- Virtual methods cannot be used until a class is fully constructed.
- A base class's virtual methods can be used once its constructor has run.
- Due to objects being stored inline, if you want to store a polymorphic object, use a pointer.
  - Storing references in classes immediately makes the class non-copyable (since references cannot be rebound).
  - If you want to store a reference, use `std::reference_wrapper`

```
// would work in a language like Java
// will NOT work in C++
auto base = std::vector<BaseClass>{};
base.push_back(base{});
base.push_back(derived1{});
base.push_back(derived2{});
```

```
// Idiomatic C++ code
auto base = std::vector<std::unique_ptr<base>>{};
base.push_back(std::make_unique<base>());
base.push_back(std::make_unique<derived1>());
base.push_back(std::make_unique<derived2>());
```

Destructing Polymorphic Objects

- Virtual methods cannot be used if a class is partially destructed.
- Every polymorphic class **must** have a virtual destructor so the resources are destructed in a proper order when you delete a base class pointer pointing to derived class object.
  - If the base class destructor is `virtual`, derived class's destructors are automatically `virtual`.
  - Remember: When you declare a destructor, the move constructor and assignment are not synthesised.
  - Forgetting this can be a hard bug to spot.

```cpp
#include <iostream>
#include <memory>

struct base {
  base() { std::cout << "A "; }

  base(base &&) = default;
  base &operator=(base &&) = default;

  virtual ~base() { std::cout << "B\n"; }
};

struct derived: base {
  derived() { std::cout << "C "; }

  derived(derived &&) = default;
  derived &operator=(derived &&) = default;
  ~derived() override { std::cout << "D "; }
};

int main() {
  std::unique_ptr<base>{new derived{}};
}

// Output: A C D B
```

Pure Virtual Methods

- Virtual functions are good for when you have a default implementation that can be overridden.
  - Sometimes there is no good default behaviour.
- A **pure virtual function** specifies a function that a class **must** override.
  - Potentially the most arcane syntax in all of C++.
  - Non-assessable extra reading: [(Im)pure virtual functions]

```
struct canvas { /* implementation */ };

struct shape {
  // Derived classes may forget to override this.
  virtual void draw(canvas &) {}

  // Fails at link time because
  // there's no definition.
  virtual void draw(canvas &);

  // Pure virtual function.
  // Any derived class must override this.
  // Declare a virtual method as normal
  // and "set" it to 0.
  virtual void draw(canvas &) = 0;
};

struct circle : shape {
  void draw(canvas &c) override { /*...*/ }
};
```

Rules for `virtual` Methods

1. Virtual member functions cannot be `static`.
2. Virtual member functions cannot be `friends`.
3. Virtual member functions only exhibit dynamic binding when used through a pointer or reference.
   - This includes pointers/reference to the most-derived type, or to any base class in the type hierarchy.
4. The prototype of virtual functions must be the same in the base as well as derived class.
   - Ensure this with `override`.
5. They are always declared in the base class and overridden in the derived class. It is not mandatory for the derived class to override virtual methods (unless pure virtual).
   - In that case, the base class version of the method is used.
6. A class must have a virtual destructor but it cannot have a virtual constructor.

Types of Class Methods

Syntax	Name	Meaning
`virtual void fn() = 0;`	Pure virtual	Inherit interface only
`virtual void fn() {}`	Virtual	Inherit interface with optional implementation
`void fn() {}`	Non-virtual	Inherit interface and mandatory implementation

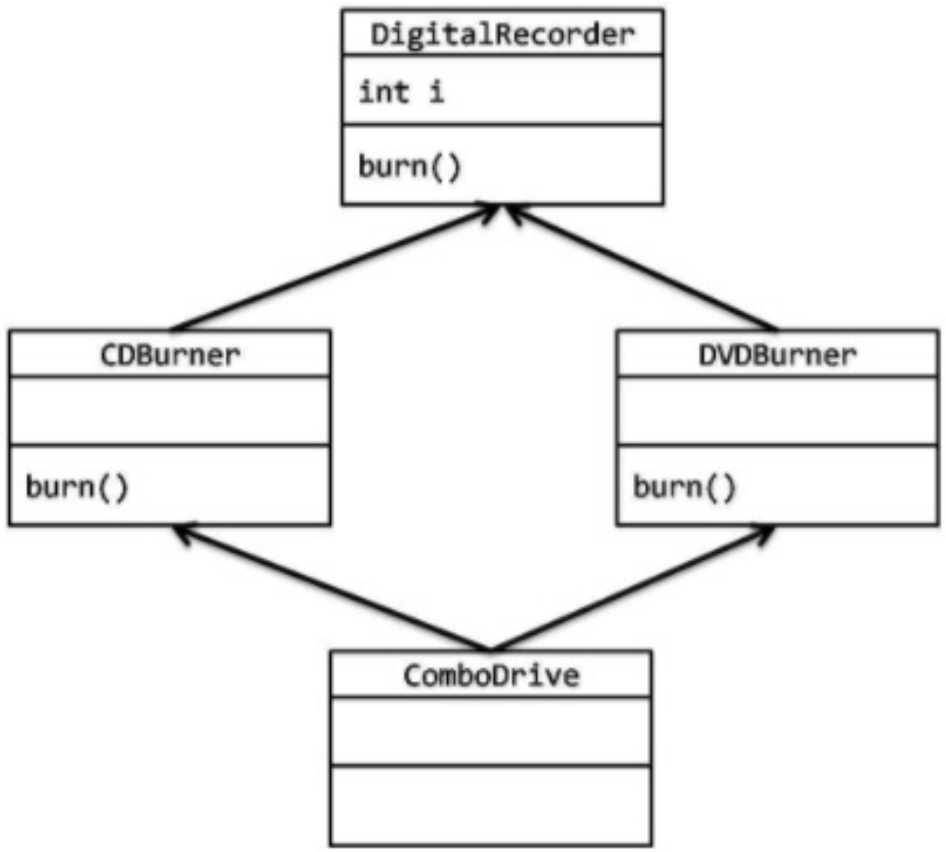
Note: non-virtuals can be hidden by writing a function with the same name in a subclass.
DO NOT DO THIS.

Abstract Base Classes (ABC)

- Might want to deal with a base class, but the base class by itself does not make sense.
  - E.g.: What is the default way to draw a shape? How many sides by default?
- Might want some default behaviour and data, but derived classes should fill out the rest of the behaviour.
  - E.g., all files have a name, but are reads done over the network or from a disk.
- If a class has at least one pure virtual 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.

The Deadly Diamond (of Death)

- C++ supports multiple inheritance.
- This can create a few specific problems:
  - In deep hierarchies, it is possible for a derived class's ancestors to inherit from the same class!
    - The derived class could have two or more copies of the same base class.
  - Calls to unqualified member functions are ambiguous.
- This is known as the "deadly diamond".
  - Not specific to C++.
  - Any OOP language that supports interfaces (Java, Python) also has this problem.
- C++ solves this via `virtual` inheritance.



the Deadly Diamond of Death(DDD)

virtual Inheritance

- If a derived class inherits virtually from a base class, it is guaranteed to only have **one** copy of any shared ancestors.
  - All ancestors in a hierarchy *must also* inherit that base virtually, however.
  - All virtual base class subobjects are initialised first before non-virtual ones.
- Unqualified calls to member functions with name collisions go through overload resolution as if those member functions were declared `virtual`.
  - Without virtual inheritance, the call would be ambiguous.

```
struct B { int n; };
class X : public virtual B {};
class Y : virtual public B {};
class Z : public B {};
  // every A has one X, one Y, one Z, and two B's:
  // - one that is the base of Z
  // - and one that is shared by X and Y
struct A : X, Y, Z {
  A() {
    // modifies the virtual B subobject's member
    X::n = 1;
    // modifies the same virtual B subobject's member
    Y::n = 2;
    // modifies the non-virtual B subobject's member
    Z::n = 3;
    // prints 223
    std::cout << X::n << Y::n << Z::n << '\n';
  }
};

struct M { void f(); };
struct B1: virtual M { void f(); };
struct B2: virtual M {};

struct C : B1, B2 {
  void foo() {
  X::f(); // OK, calls X::f (qualified lookup)
  f(); // OK, calls B1::f (unqualified lookup as if virtual)
  }
};
```

Casting Up a Type Hierarchy

- Casting from a derived class to a base class is called up-casting.
- This cast is always safe.
  - All derived classes *are* base classes, after all.
- Because the cast is always safe, C++ allows this as an implicit cast.
- One potential reason to use auto is that it avoids implicit casts.

```cpp
struct animal { /* ... */ };
struct dog : animal { /* ... */ };

int main() {

  auto doggo = dog();

  // Up-cast with references.
  animal& animalia_r = doggo;

  // Up-cast with pointers.
  animal* animalia_p = &doggo;
}
```

Casting Down a Type Hierarchy

- Casting from a base class to a derived class is called down-casting.
  - This cast is not guaranteed to be safe.
- 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.
- `dynamic_cast` relies on **Runtime Type Information** (RTTI).
  - RTTI is one of the most disabled features of C++ due to its performance cost.

```cpp
struct animal { virtual ~animal() = default; };
struct dog : animal { /* ... */ };
struct cat : animal { /* ... */ };

dog d;
cat c;
animal& ad = d;
animal& ac = c;

int main() {
  // Attempt to down-cast with references.
  dog& dr1 = static_cast<dog&>(ad);
  dog& dr2 = dynamic_cast<dog&>(ad);

  // Undefined behaviour - incorrect static cast.
  dog& dr3 = static_cast<dog&>(ac);

  // Throws exception
  dog& dr4 = dynamic_cast<dog&>(ac);

  // returns null pointer
  dog* dp1 = dynamic_cast<dog*>(&ac);
}
```

OOP: Covariance

- If a derived class overrides a virtual method from a base class, what should the return type be?
- Every possible return type for the derived's overridden method must be a valid return type for the base's original method.

```
struct fruit { virtual ~fruit() = default; };
struct apple : fruit { /* ... */ };
struct granny_smith : apple { /* ... */ };

struct parent {
  apple app = apple{};

  virtual const fruit &get_fruit() const {
    // OK: apple is a fruit!
    return app;
  }
};

struct child : parent {
  granny_smith gs = granny_smith{};

  const apple &get_fruit() const override {
    // OK: granny_smith apples are a fruit too!
    // this method override is covariant
    return gs;
  }
};
```

OOP: Contravariance

- If a derived class overrides a virtual method from a base class, what should the parameter types be?
- An overridden method can accept more general types as the parameters.
  - Every possible argument to the base class's method **must** be a valid argument to the derived's overridden method.
- Not as easy to ensure as covariant methods in C++.

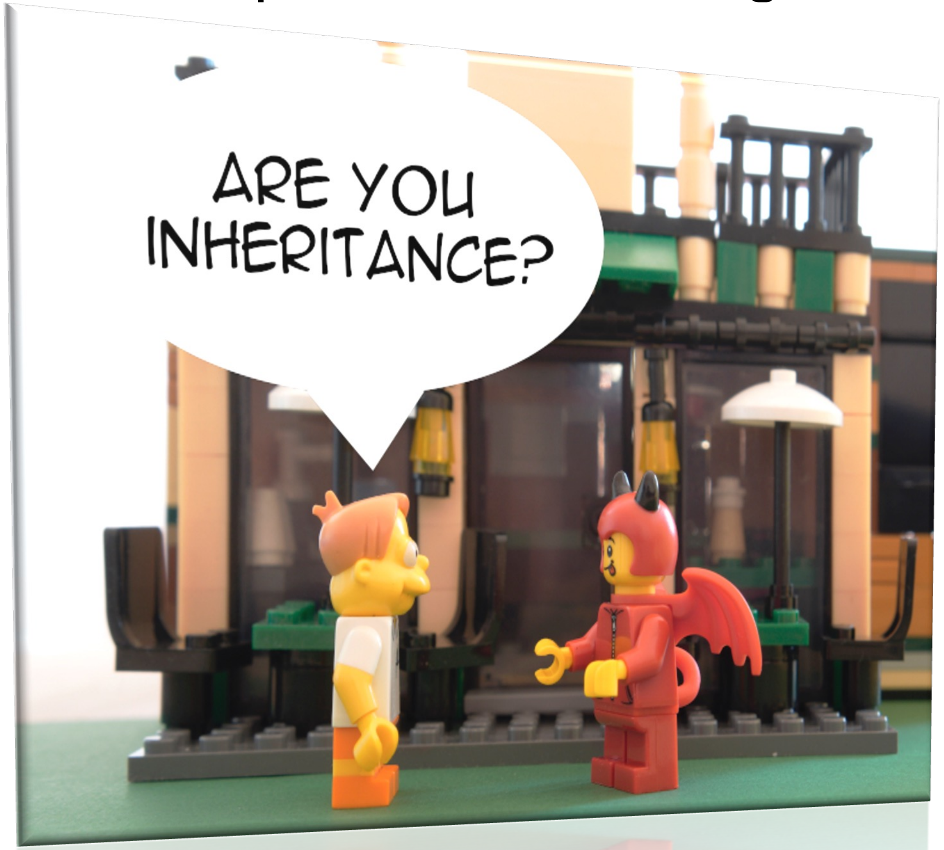
```
struct fruit { virtual ~fruit() = default; };
struct apple : fruit { /* ... */ };

struct parent {
  virtual void eat(const apple &a) const {
    // nom nom nom on that apple
  }
};

struct child : parent {
  // this method override is contravariant!
  // ...but the compiler doesn't accept this
  // because the signatures between the
  // base and derived class don't match.

  // could potentially reuse the base's
  // method if we explicity down-cast
  // with dynamic_cast or static_cast
  void eat(const fruit &f) const override {
    // nom nom nom on that fruit
  }
};
```

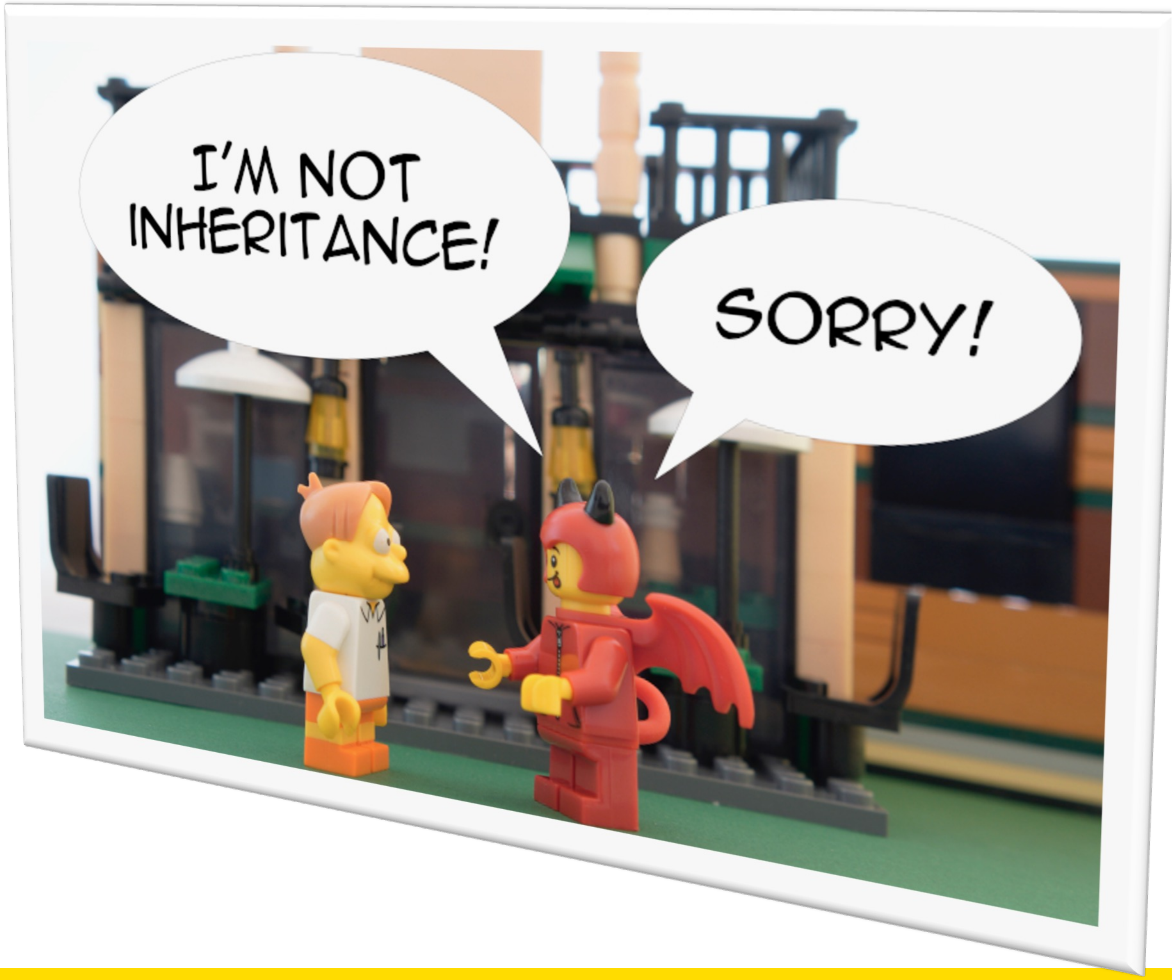
Can Inheritance be dangerous?

[Inheritance-the base class of evil](https://learn.microsoft.com/en-us/events/goingnative-2013/inheritance-base-class-of-evil) Sean Parent)

https://learn.microsoft.com/en-us/events/goingnative-2013/inheritance-base-class-of-evil

https://thevaluable.dev/guide-inheritance-oop/





Images source: https://thevaluable.dev/guide-inheritance-oop/s

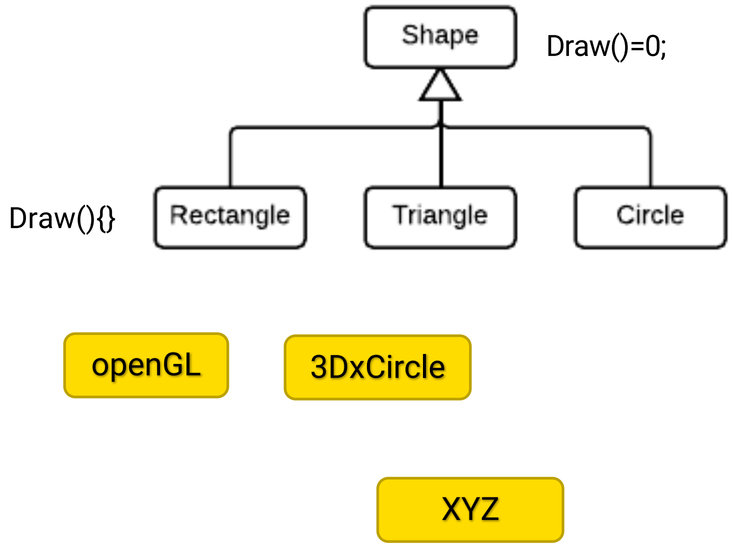
Problem with Inheritance

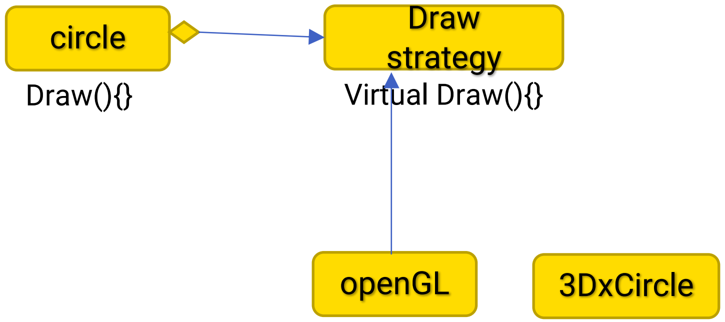
- Often requires dynamic allocation.
- Ownership and nullability semantics become hazy.
- Intrusiveness: requires modifying child classes.
- No more value semantics.
- Can change semantics for algorithms and containers.

Dependencies: Design Principle

Problem-Change: Software must be adaptable

- Using inheritance is not the only way to extend a class behavior, but definitely is the **most dangerous and harmful one.**
- One change in the base class could affect the behavior of the child.





The Gang of Four: book

23 most commonly used design pattern

Some Guidelines

- Use **only** two type of class-level declarations: **interfaces and final classes**;
- Inject interfaces in dependent classes’ constructors;
- **Don’t allow** any class dependency to be injected other than interfaces;
- Use a Dependency Injection Container (or an equivalent method, depending on which language you’r coding with) to handle the creation of my instances;
  - If injecting too many dependencies in a class, rethink design in terms of class responsibilities and using the interface segregation principle;
- When required, split complex behavior in multiple final classes implementing the same interface;
- Use inheritance **only** when it makes sense on a semantic level and only for extension purposes, without any base behavior change;

Feedback (stop recording)