C++

May 2, 2022

Mike Spertus

spertus@uchicago.edu

MASTERS PROGRAM IN COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO

"OO" DESIGN

Why the quotation marks?

- C++ offers many ways to address the sort of problems covered by OO design
- Not all of these are considered "OO" techniques
- Let's look at an example

TEMPLATES AND OO CAN SOLVE SIMILAR PROBLEMS

Generic Programming Should "Just" Be Normal Programming

— Bjarne Stroustrup

Using templates instead of inheritance and virtuals

- OO and templates can be used for many of the same problems
- In the spirit of Bjarne’s quote, new C++ features, like C++20 Concepts, were intentionally designed to be familiar for replacing object-oriented code
- Let’s review an example we looked at before

OO and Templates can solve similar problems

- Consider the following OO code

```
struct Animal {
   virtual string name() = 0;
   virtual string eats() = 0;

};

class Cat : public Animal {
   string name() override { return "cat"; }
   string eats() override { return "delicious mice"; }

};
// More animals...

int main() {
   unique_ptr<Animal> a = make_unique<Cat>();
   cout << "A " << a->name() << " eats " << a->eats();
}
```

Do we need to use OO?

- Not really

```
struct Cat {
    string eats() { return "delicious mice"; }
    string name() { return "cat";}
};
// More animals...

int main() {
    auto a = Cat();
    cout << "A " << a.name() << " eats " << a.eats();
}
```

That was a lot simpler but…

- We lost the understanding that a is an animal
- a could have the type House or int and we might not find out that something went wrong until much later when we did something that depends on a being an animal
- What we need is a way to codify our expectations for a without all of the overhead and complexity of creating a base class

Concepts

- Concepts play the analogous role for generic programming that base classes do in object oriented programming
- A concept explains what operations a type supports
- The following concept encapsulates the same info as the base class

```
template<typename T>
concept Animal = requires(T a) {
  { a.eats() } -> convertible_to<string>;
  { a.name() } -> convertible_to<string>;
};
```

Now, we can ensure that a represents an animal

- With the above concept defined, we can specify that a must satisfy the Animal concept, and the compiler will not let us initialize it with a non-Animal type like House or int

```
int main() {
   Animal auto a = Cat();
   cout << "A " << a.name() << " eats " << a.eats();
}
```

Let’s compare

- https://godbolt.org/z/cWc6aM
- As you can see, the definition of `Cat` and the client code in `main()` look very similar in both
- This follows a principle enunciated by Bjarne Stroustrup
  - “Generic Programming should just be Normal Programming”

Usage is almost identical to before

```
Animal auto a = Cat();
cout << "A " << a.name()
    << " eats " << a.eats();
```

How does this compare?

- Performance is better
  - Objects created on stack
  - No virtual dispatch
- No inheritance
  - Makes it easier to adapt classes to our code without risking “spaghetti inheritance”
  - On the other hand, it weakens type safety
    - Pacman is not an animal but eats and has a name
- No runtime polymorphism
  - The following is legal if `Animal` is a class but not if it is a concept (Why?)
  - `set<unique_ptr<Animal>> zoo;`

A real world example

- Suppose C++ didn't have mutexes
  - It didn't until C++11
- How would we design them?
- Let's look at how Java does it
- Java uses inheritance and virtual methods

Java-style mutexes

```
struct lockable {
  virtual void lock() = 0;
  virtual void unlock() = 0;
};
struct mutex: public lockable {
  void lock() override;
  void unlock() override;
};
struct lock_guard {
  lock_guard(lockable &m) : m(m) { m.lock(); }
  ~lock_guard() { m.unlock(); }
  lockable &m;
};
```

Using our Java-style mutex

```
mutex m;

void f() {
  lock_guard lk(m);
  // do stuff
}
```

Wait, that’s not how C++ mutexes work!

- C++ mutexes do not inherit from a lockable base class
- The C++ committee decided to use templates instead of the virtual override approach taken by Java
- Since mutexes are frequently used in performance-critical code, this was undoubtedly the right choice
- Let’s take a look

C++-style mutexes

```
struct mutex {
  void lock();
  void unlock();
};


template<typename T>
struct lock_guard {
  lock_guard(T &m) : m(m) { m.lock(); }
  ~lock_guard() { m.unlock(); }
  T &m;
};
```

Using our C++-style mutex is typically unchanged

```
// Exactly the same as before!
mutex m;

void f() {
  lock_guard lk(m);
  // do stuff
}
```

Class Template Argument Deduction

- Note the following line depended on C++17 CTAD
  - `lock_guard lk(m);`
- CTAD infers the template arguments for `lock_guard<mutex>` from the constructor similarly to how Function Template Argument Deduction infers the template arguments for function templates
- CTAD can often be useful in this way when using templates instead of virtuals. E.g., if `tp1` and `tp2` are of type `time_point<C, duration<R>>`
  - `duration d = tp1 - tp2; // duration<R>`

So many choices :/

- As we saw above, there is usually a choice between base classes/virtuals and templates/concepts
- As we will see below, this is only the beginning
- C++ supports many approaches for "object-orientation"
- How can we make sense of this?
- We will need an understanding of OO design best practices

THE SOLID PRINCIPLES
(H/T TONY VAN EERD)

Popularized by "Uncle Bob" Robert C Martin ~2004

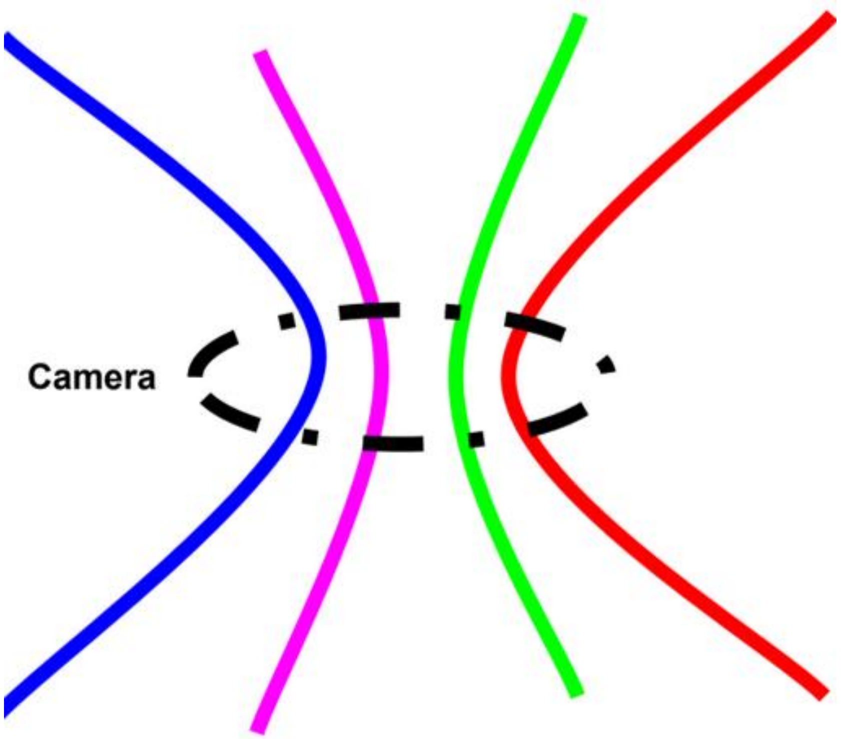
- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle

- How often do you see a class that has members organized into subsets?

```
class Camera
{
  CameraId id;
  DevicePath path;

  int bitdepth;
  Resolution resolution;
  int gain;
  int exposure; // units?

  Pose pose;
  Calibration * calibration;

  int binarizationThreshold;
  int sharpness; // UI only
  int multicamEdgeThreshold;

  Image baseImage;
  Image negativeImage;
  Image maskToUse;
  Image reverseMask;

  Device * device;
  INativeCamera * camera;

  ImagePoller * poller;
};
```

These are separate pieces pulled together by association



What if we wanted to use just one piece?

- Can I pose (position) things other than a camera?
- Does posing really have anything to do with resolution?
- What if a camera supported multiple resolutions but only a single pose?
- We can't do that with this class without breaking the Open-Closed principle
  - Which we'll learn about momentarily, but in this context it means "without breaking existing code"


DON'T CROSS THE STREAMS
memegenerator.net

Better: Give Image its own class

```
struct Image
{
   unsigned char * pixels;
   Format format; // RGB vs RGBA vs Gray vs …
   int width;
   int height;

   Image();
   Image(int width, int height, Format format);
};
```

It's not just classes that mix concerns

- What about functions?
- Many functions are thousands of lines long
  - E.g., https://github.com/llvm/llvm-project/blob/b07aab8fc1088ef66ecbe2befc3ef7e3936a390e/clang/lib/Parse/ParseExprCXX.cpp#L154
-
```
void myFunction(Quux q) {
  // Locals
  Pass p(1);
  Bar b(p);
  Blah l(q);
  // Setup the frabulator
  Frabulator f;
  f.x();
  f.y = "foo";
  // Loop through the sneetches
  for(auto s: f.sneetches)
    ...
```

How can I break a function into single-concern pieces?

- Breaking into several functions is typical
- But is sometimes difficult to share state
- ```
  void myFunction(Quux q) {
    // Locals
    Pass p(1);
    Bar b(p);
    Blah l(q);
    Frabulator f = setupFrabulator(b, p, l, q);
    loopThroughSneetches(f, b, p, l, q);
      ...
  ```
  ```

Option 1: More, smaller functions

- Breaking into several functions is typical
- But is sometimes difficult to share state
- ```
  void myFunction(Quux q) {
    // Locals
    Pass p(1);
    Bar b(p);
    Blah l(q);
    Frabulator f = setupFrabulator(b, p, l, q);
    loopThroughSneetches(f, b, p, l, q);
      ...
  ```
  ```

Option 2: Use a functor

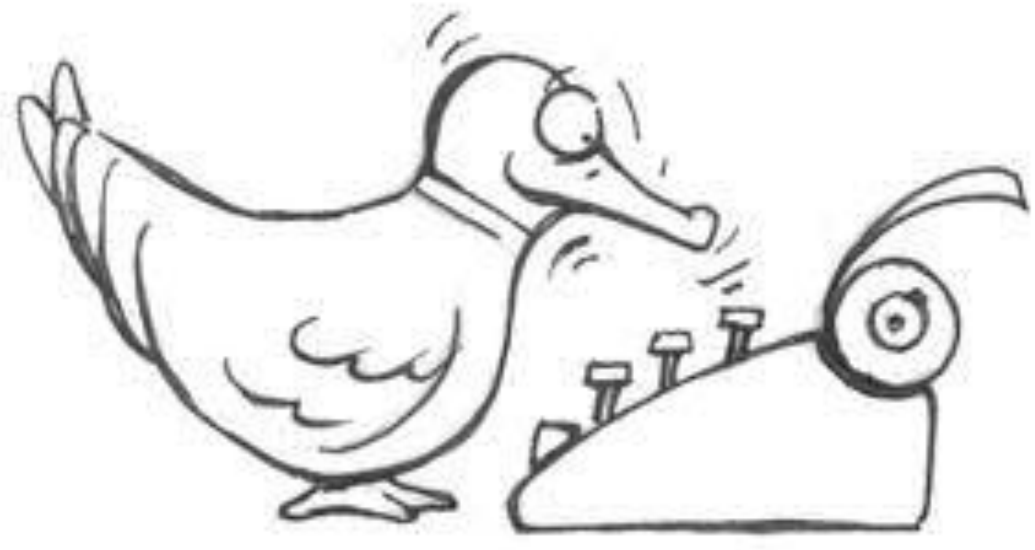
- Functions can be restructured as functors for better organization
- ```
  struct myFunctionHelper {
    myFunctionHelper(Quux q) : q(q), l(q) {}
    Quux q;
    Pass p(1);
    Bar b(p);
    Blah l;
    // Methods now have access to variables
    Frabulator setupFrabulator() { ... }
    void loopThroughSneetches() { ... }
    int operator() {
      Frabulator f = setupFrabulator();
      loopThroughSneetches(f);
      ...
   };
  int myFunction(Quux q) { return myFunctionHelper(q)(); }
  ```
- Good choice for really complex functions
- Supports powerful features like virtual functions and multiple entry points
- but high-friction for simpler cases
  ```

Option 3: Use lambda with capture lists

- While C++ doesn't have local functions *per se*, you can get the same organizing effect with local lambdas
-
```
int myFunction(Quux q) {
Pass p(1);
  Bar b(p);
  Blah l;
  auto setupFrabulator = [&] { ... }
  auto loopThroughSneetches = [&](Frabulator f) { ... }
  Frabulator f = setupFrabulator();
  loopThroughSneetches(f);
    ...
 };
```
- Good for "goldilocks" cases where a class is overkill but multiple responsibilities risk a "spaghetti" function

Open/Closed Principle

- Software constructs should be open for extension but closed for modification
- Open for extension
  - Allows the addition of new capabilities over time
- Closed for modification
  - Don't break existing client code

Inheritance and virtuals

- Of course, inheritance and virtual functions are a great way to extend classes
  - As are the template equivalents we discussed above
- Is it enough?
- Not quite
- It does not cover extremely common use cases for extensions

The Problem: Client-side extension

- Suppose you are using a class hierarchy, and you wish the classes had a virtual method specific to the needs of your application
- Unfortunately, it probably doesn't because the class designer doesn’t understand your application
- You may not be able to add them
  - Maybe they’re not your classes
  - Maybe the virtuals you want only apply to your particular program, and it breaks encapsulation to clutter up a general interface with the particulars of every app that uses them

The Visitor Pattern

- The Visitor Pattern is a way to make your class hierarchies extensible
- Suppose, as a user of the `Animal` class, I wished that it had a `lifespan()` method, but the class designer did not provide one
- We will fix that with the visitor pattern

Class Creator: Make Your Classes Extensible

- Create a visitor class that can be overriden
- ```
  struct AnimalVisitor {
    virtual void visit(Cat &) const = 0;
    virtual void visit(Dog &) const = 0;
  };
  ```
- Add an “accept” method to each class in the hierarchy
- ```
  struct Animal {
    virtual void accept(AnimalVisitor const &v) = 0;
  };
  struct Cat :  public Animal {
   virtual void accept(AnimalVisitor const &v)
    { v.visit(*this); }
    /* … */
  };
  ```

Class User: Create a visitor

- Now, I create a visitor that implements the methods I wish were there
-
```
struct LifeSpanVisitor
  : public AnimalVisitor {
 LifeSpanVisitor(int &i) : i(i) {}
 void visit(Dog &) const { i = 10; }
 void visit(Cat &) const { i = 12; }
 int &i;
};
```

Using the visitor

- Now, I can get the lifespan of the `Animal a` I created above
- ```
  int years;
  a->accept(LifeSpanVisitor(years));
  cout << "lives " << years;
  ```
- https://godbolt.org/z/WbGe4z
  ```

Best practice

- If you are designing a class hierarchy where the best interface is unclear, add an `accept()` method as a customization point

USING DUCK-TYPED VARIANTS FOR PERFORMANT, EXTENSIBLE OO

Duck Typing



Duck Typing

- OK. Not that
  - When I gave a talk on this in Shenzhen, I was worried what the translator would do with the last slide :/
- There is a saying

> If it walks like a duck and quacks like a duck, then it is a duck

- Let's see if we can apply this to types

Inheritance models “isA”

- As we’ve mentioned, inheritance is a model of the “isA” concept
- Duck typing gives a different notion of “isA”
- If a class has a walk() method and a quack() method, let’s not worry about inheritance and call it a duck

If it walks like a duck and quacks like a duck, then it isA duck

Templates use duck typing

- `T square(auto x) { return x*x; }`
- `square(5); // OK`
- We don't require that the type of x inherits from a HasMultiplication class, simply that operator* makes sense to use here

Concepts

- C++20 Concepts improve duck typing
- We could have a HasMultiplication concept that would do the trick without requiring any complex inheritance hierarchies

Dynamic dispatch

- While C++ templates have always been duck typed, templates are used for compile-time dispatch
- By contrast, OO is used for dynamic dispatch
- Because duck typing is flexible and forgiving while remaining statically typesafe, people have asked whether we could use dynamically-dispatched duck typing as an alternative to inheritance

Variants

- C++ has a lightweight "variant" abstraction that generalizes C unions
- A `variant<A, B>` can hold an `A` or a `B` but not both
- These variants will be the basis of an approach to OO that will
  - Often have much better performance than virtuals
  - More dynamic than our `Animal` concept (we can have a zoo with runtime dispatch)
  - Everything is value-based, no need to worry about references, `unique_ptr`, RAII, …
  - Great support for Open-Closed extensibility

Variants: Basic use

- But first, what is a variant?
- A variant is a lot like a tuple, but instead of holding all of its fields at once, it only contains one of them at a time
- Supports a very similar interface to tuples
- ```
  variant<int, double> v = 3; // Holds int
  get<0>(v); // Returns 3
  get<1>(v); // Throws std::bad_variant_access
  v = 3.5;   // Now holds a double
  get<double>(v); // Returns 3.5
  ```
- You can also check what is in it
  ```
  v.index; // returns 1
  holds_alternative<double>(v); // returns true
  holds_alternative<int>(v); // returns false
  ```
  ```

Using Duck-Typed Variants in place of OO

- Suppose we knew (at compile-time) all of the Animal classes
  - E.g., Cat and Dog
- However, we don’t know the type of a particular animal until run-time
- Instead of inheriting from an abstract animal base class, we can have an animal variant
- `using Animal = variant<Cat, Dog>;`

How do I call a method on a variant?

- While `variant<Cat, Dog>` is a great way to store either a `Cat` or a `Dog`
- How can I simulate virtual functions and call the right `name()` method for the type it is holding?

The C++ standard library has a solution: std::visit

- **Warning:** Do not confuse with the Visitor Pattern we discussed earlier
- If `v` is a variant, and `c` is a callable, `visit(v, c)` calls `c` with whatever is stored in `v` as its argument
- Does this solve our problem of making variants behave like virtuals?
- Let’s see

Dynamically calling our `Animal`'s `name()` method

- `Animal a = Cat(); // a is a cat`
- ```
  cout << visit(
    [](auto &x) { x.name(); },
    a);
  ```
- Prints "`Cat`", just like we want
- How does it compare to using virtuals or templates?
  ```

In some ways, it's the best of both world

- Almost as fast as templates
  - Since `Animal` is a single type that can hold a `Cat` or a `Dog`, we can just use animals by value instead of having to do memory allocations
- As dynamic as traditional OO
  - A `set<Animal>` works great
  - Unlike our Concepts version

Malleable (mallard?) typing

- Virtual functions need to exactly match what they override, so we couldn't, say, give `Cat`'s `eat()` a defaulted argument
  - ```
    struct Cat : public Animal {
      void eat(string prey = "mouse") override; // ill-formed
    ```
- With variants, that is not a problem
- As long as `eats()` is callable, we don't care about the rest
- ```
  struct Cat {
    void eat(string prey = "mouse");
  };
  visit([](auto &x) { x.eats(); }, a); // OK. Eats a mouse
  ```

Users can add methods

- Just like we discussed with the Visitor Pattern, users of the Animal type can add their own methods
- To do this, we will use the “overloaded pattern”

The overloaded pattern

- Define an `overloaded` class (you only need to do this once)
- ```
  template<class... Ts>
  struct overloaded : Ts... { using Ts::operator()...; };
  ```
- This inherits the function call operator of everything it is constructed with
- Let's make this clear by creating a `lifeSpan()` "method" like we did before
- ```
  overloaded lifeSpan([](Cat &) { return 12 },
                      [](Dog &) { return 10});
  // Get life span without knowing whether
  // a is a Cat or a Dog
  cout << visit(lifeSpan, a); // Prints 12
  ```
- Note that this idiom relies on CTAD and aggregates deducing the template arguments for you

Problems with Duck Typing

- The notation is much uglier than calling a virtual function
- While the flexibility is nice, duck typing reduces type safety
  - It cannot tell that a Shape's draw() method for drawing a picture is different than a Cowboy's draw() method for drawing a gun
- Variants always uses as much space as the biggest type
- Whenever we create a new kind of Animal, we have to add it to the variant, which can create maintenance problems

Best Practice

- Because it is ugly and not well-known, only prefer variant-based polymorphism over virtuals or templates when there is a clear benefit
  - In practice, I use it a lot, but less than I do virtual functions or templates

Liskov Substitution Principle

Subtype Requirement:

Let Φ (x) be a property provable about objects x of type T. Then Φ (y) should be true for objects y of type S

where S is a subtype of T

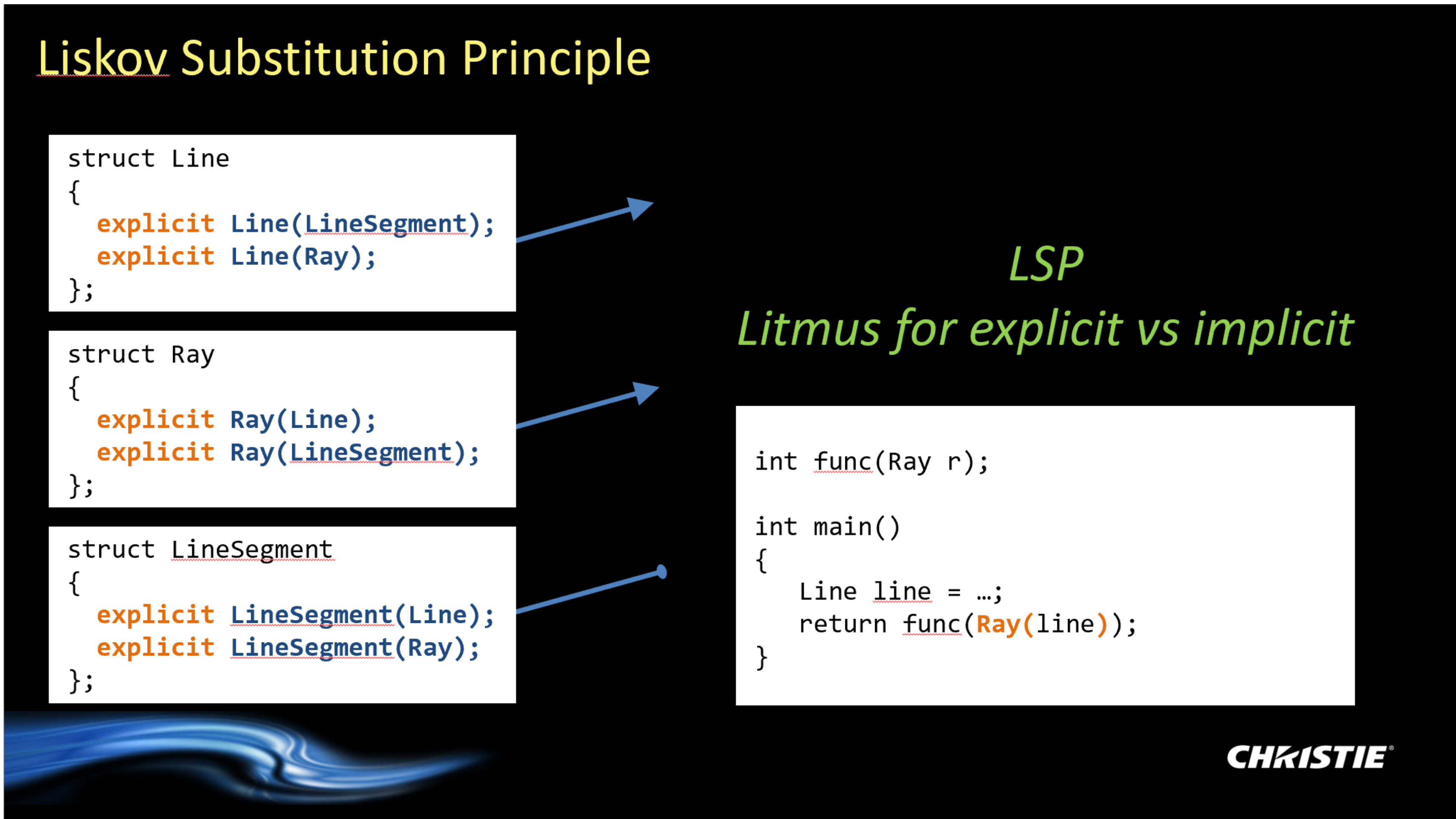
Inheritance models "isA"

- Inheritance is one way of modeling subtyping
- A Deer isA Animal
- One would expect that anything that is true about animals is true about deer

Concepts model isA

- If Animal is a concept instead of a base class, we have seen that the same is true
- One other benefit of concepts is that inheritance only inherits methods, but concepts can specify almost arbitrary $\Phi(x)$ properties
- Tradeoff: Efficiency vs dynamism
  - Generally how to choose the approach

Tony says


Liskov Substitution Principle
struct Line
{
explicit Line(LineSegment);
explicit Line(Ray);
};
struct Ray
{
explicit Ray(Line);
explicit Ray(LineSegment);
};
struct LineSegment
{
explicit LineSegment(Line);
explicit LineSegment(Ray);
};
LSP
Litmus for explicit vs implicit
int func(Ray r);
int main()
{
Line line = …;
return func(Ray(line));
}
CHRISTIE

Interface Segregation Principle

- No code should be forced to depend on methods it doesn't use
- (Martin) Suppose we have a fat "Job" class that has a bunch of methods that are only relevant to print jobs and other methods that are only relevant to stapling jobs
- If the stapling code takes a Job, it will needlessly only work with Jobs that also know about printing

Handling with OO

- This is often given as a motivation for using abstract base classes (interfaces)
- The concrete `Job` class implements the `PrintJob` and `StapleJob` interfaces
- This can be taken so far, getting into spaghetti inheritance and excessive multiple inheritance complexity

Concepts also handle this nicely

- The stapling code can only require what it needs to staple without exploding the type hierarchy
- However, you could also go too far with this too as an incoherent set of functions that each make different requirements of each job that is passed in
- Both of these are good reminders that architecture is more art than science

Dependency Inversion Principle

- This is sometimes paraphrased as "All programming problems can be solved with an extra layer of indirection"
- ☺

Basic idea

- *" the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions"*

Example

- Suppose you have a thumbnail service class that looks for pictures in S3 folders

```
class ThumbnailService {
  S3Folder inputFolder;
  …
};
```

- It is now coupled with the concrete S3 service instead of an abstract idea of a storage service, which is probably sufficient for this use
- Again, the indirection can be introduced either through inheritance/virtuals or template/concepts
- Usual performance/dynamism tradeoff

Solving with inheritance and virtuals

- ```
  Class S3Folder : public Folder { … };
  class ThumbnailService {
    Folder &inputFolder;
    …
  };
  ```
  ```

Solving with templates and concepts

- ```
  template<Folder F> // Folder is a concept
  class ThumbnailService {
    Folder &inputFolder;
    …
  };
  ThumbnailService<S3Folder> ts;
  ThumbnailService ts2(myS3Folder); // CTAD
  ```
  ```

HW 15-1

- *Static polymorphism* is another common idiom for using templates to achieve "OO-like" behavior at compile-time
- Read the static polymorphism section of https://iamsorush.com/posts/static-polymorphism-cpp/
- Reimplement our "animal example" from https://godbolt.org/z/cWc6aM using static polymorphism
- What is your opinion of this approach? Do the SOLID principles shed any light on that?

HW 15-2

- Convert our "train example" on Canvas to use the variant/visit approach instead of inheritance
  - E.g., `ModelLocomotive` will no longer inherit from `Locomotive`
  - **Note:** Just adapt the train_factory example, not the advanced_train_factory
- **Hint:**
  - First just get the trains working without the factory (6/10 pts)
  - Next, figure out how to make factory templates that work with variant/visitor (4/10)

HW 15-3: Extra Credit

- Choose some C++ program (e.g., from GitHub)
- Analyze it according to the SOLID principles
- What does it do well?
- What does it do badly?
- Can you suggest improvements?