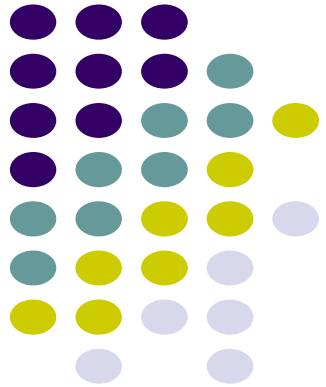
How to "OO" in C++

July 19, 2022
CPP North

Mike Spertus

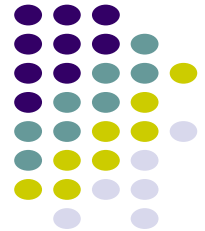
mike@spertus.com



What we will discuss

- When faced with OO problems, many C++ programmers have the perception that they need to use inheritance and virtual functions, but that's far from the only way to do it in C++
- These "OO" techniques are often a better choice in practical programs
- This talk will cover three approaches that cover most use cases, and I find myself using again and again
  - Inheritance and Virtual functions, Concepts, and Variant-based polymorphism

What is "OO"?



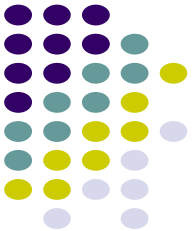
Spoiler

"Object-Oriented"

What is "Object-Oriented"?

Spoiler

No consensus



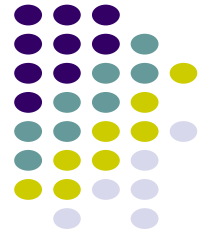
> Most arguments about “What is the essence of…?” do more to reveal the prejudices of the participants than to uncover any objective truth about the topic of discussion. Attempts to define the term “object-oriented” precisely are no exception.

— Benjamin Pierce
Types and Programming Languages. MIT Press.

We'll take it at face value

- "Object-oriented" means
  - "oriented around objects"
- When we need to operate on an object, we will ask the object to perform the operation
  - What and how it does it is none of our business
- To do this, call a method on the object

Methods



- Methods are defined as part of the object's class
-
```
class A {
  string foo() { return "foo"s; }
  virtual string bar() { return "bar"s; }
};
/* ... */
A foobar;
cout << foobar.foo() << foobar.bar();
```

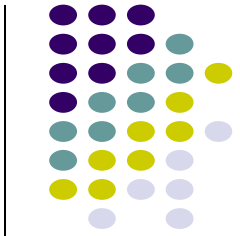
But what is the object's class?

- I think this gets to the heart of object-orientation
- An expression in C++ has two types
  - The type of the expression
    - Static/Compile-time type
  - The type of the actual object in memory
    - Dynamic/Run-time type
- These can differ (polymorphism), mainly due to inheritance
  - We'll see an example below
- Which class' method is used?

The philosophy of most languages

- **The language chooses**
- Most languages always use one or the other
- Java describes itself as an object-oriented language, so it always uses the dynamic type
  - The type of the object in memory at run-time
- C is not object-oriented, so it always uses the static type
  - The type of the expression at compile-time

The philosophy of C++



- **The programmer chooses**
- Since both are useful, C++ supports each
- Non-virtual methods use the static type
- Virtual methods use the run-time type

Virtual vs. Non-virtual method

```
struct Animal {
  void f() { cout << "animal f"; }
  virtual void g()  { cout << "animal g"; }
};
struct Gorilla : public Animal{
  void f() { cout << "gorilla f"; }
  void g() { cout << "gorilla g"; }
  void h() { cout << "gorilla h"; }
};
void fun() {
   Gorilla g;
   Animal &a = g;
   a.f(); // Not virtual: Animal's f
   a.g(); // Virtual: Gorilla's g
   a.h(); // Ill-formed: h is not in animal
   g.f(); g.g(); g.h(); // Gorilla's f, g, and h
}
```

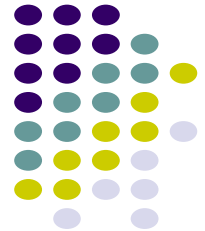
Why not always use virtual methods?

- In Object-oriented languages like Java, all methods are virtual
- I'll use OO without quotation marks to refer to this familiar OO style of using inheritance and virtuals
- Are there any downsides?

Why not always use virtuals? Performance

- You may have heard that virtual functions only have one more indirection, and you don’t need to worry about the performance
- This is true most of the time
- But not always (this is not well-known)
- Let’s look at an example

Benchmark 1: Performance unchanged if f is virtual



```
class A {
public:
   virtual int f(double i1, int i2) {
       return static_cast<int>(i1*log(i1))*i2;
   }
};
int main()
{
   A *a = new A();
   int ai = 0;
   for(int i = 0; i < 100'000'000; i++) {
           ai += a->f(i, 10);
   }
   cout << ai << endl;
}
```

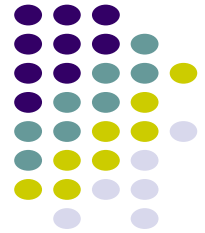
Benchmark 2: Non-virtual ten times faster

```
class A {
public:
   virtual int f(double d, int i) {
       return static_cast<int>(d*log(d))*i;
   }
};
int main()
{
   auto a = new A();
   int ai = 0;
   for(int i = 0; i < 100'000'000; i++) {
           ai += a->f(10, i); // Only change!
   }
   cout << ai << endl;
}
```

What happened?

- Changing one line made virtuals 10 times slower than regular methods!
- The main performance cost of virtual functions is the loss of inlining and function level optimization
  - Not the overhead of the indirection
  - In the second benchmark, log(10) only needed to be calculated once in the non-virtual case.
- Usually not significant, but this case is good to understand
  - The more virtual functions you use, the less the compiler can understand your code to optimize it

Why not always use virtuals? Polar bears



- Java programs use 50% more energy than C++ programs
  - https://greenlab.di.uminho.pt/wp-content/uploads/2017/09/paperSLE.pdf
- While modern Java JITs use advanced devirtualization techniques
- Deferring all dispatching until runtime is likely a big reason why

Why not always use virtuals? Interoperability

- Code that needs to interoperate with other languages needs datatypes with a fixed layout in memory
- C++ provides an `is_standard_layout` type trait to check this
- The presence of any virtual methods breaks this due to the "vtable pointer"

Why not always use virtuals? Verifiability

- Since virtual method calls are not dispatched at runtime, it is much harder to verify its correctness
- Code coverage tools, static analyzers, model checkers, etc. all have a harder time in the presence of virtual functions
  - This is similar to how virtual functions stressed the optimizer in the example above

Why not always use virtuals? Spaghetti inheritance

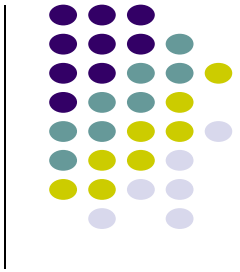
- In inheritance/virtual based object-orientation
- A class needs to inherit from every possible thing it needs to override
- If you are familiar with the SOLID principles of OO best practices, the Interface Segregation principle says:
  - Many client-specific interfaces are better than one general-purpose interface
  - For more on the SOLID principles, see Tony Van Eerd's great talk
- This quickly leads to messy unmaintainable inheritance hierarchies in practice
- We'll see some concrete examples shortly

Why not always use virtuals? No client-side extensibility

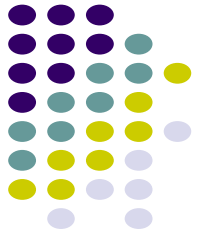
- Suppose you are writing a game that uses a library of animal classes written by someone else
- You wish each animal had a virtual `hitPoints()` method, but of course they don't because the designer of the animal hierarchy doesn't know about your app
- This happens a lot? What do you do?
- If you are familiar with the SOLID principles of OO best practices, this is an example of the Open-Closed principle
  - You want to extend the classes behavior (Open)
  - Without modifying the existing classes (Closed)

Virtual methods: Best Practices

- Usually it is fine to make methods virtual, the performance cost is "almost always" minimal and the availability of runtime dispatch is powerful in object-oriented code
- But don't use virtual functions gratuitously if there is no reason to do so. Gratuitous use of virtual functions can…
  - …impede compiler code optimization
  - …unpredictably modify memory layout
  - …impede program analysis and reasoning tools
  - …lead to unmaintainable "spaghetti inheritance" hierarchies
  - …fail to meet the needs of client-side extensibility
  - …kill polar bears



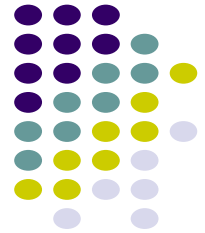
TEMPLATES AND OO CAN SOLVE SIMILAR PROBLEMS



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
  - With template inference replacing virtual dispatch
- Let’s consider a real-world example

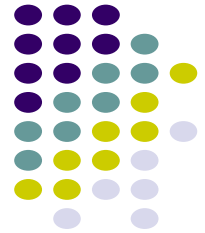
A real world example

- 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

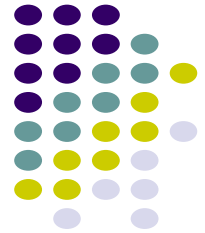
```
struct lockable {
  virtual void lock() = 0;
  virtual void unlock() = 0;
};
struct mutex: public lockable {
  void lock() override;
  void unlock() override;
};
struct lock_guard {
  lock_guard(lockable &m) : m(m) { m.lock(); }
  ~lock_guard() { m.unlock(); }
  lockable &m;
};
```

Using our Java-style mutex
Just like standard C++!



```
mutex m;

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

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

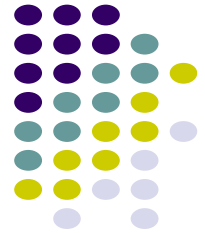


- C++ mutexes do not inherit from a lockable base class
- The C++ committee decided to use templates instead of the virtual override approach taken by Java
- Since mutexes are frequently used in performance-critical code, this was undoubtedly the right choice
- Let's take a look

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

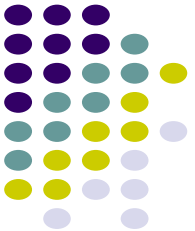


```
// Exactly the same as before!
mutex m;

void f() {
  lock_guard lk(m); // CTAD auto-deduces
  // do stuff
}
```

Concepts make it even better

- In the spirit of Bjarne’s quote, new C++ features, like C++20 Concepts, were intentionally designed to be familiar to users of "normal" object-oriented code

Animal example with virtuals



- An Animal example like this is often given as the "hello world" of OO programming

```
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?

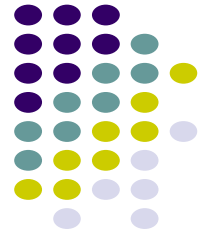
- Not really

```
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

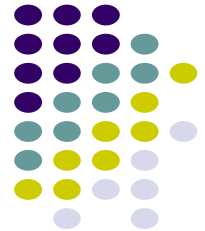
Concepts



- Concepts play an analogous role for generic programming that base classes do for object oriented programming
- A concept explains what operations a type supports

```
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/h9r3c4j8b
- As you can see, the definitions of `Cat` and the client code in `main()` are very similar
  - "Generic Programming should just be Normal Programming"
  - Bjarne was right!

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;`

What to choose?

- As we just saw, 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 a better understanding of the problems

Client-side extensibility

- Suppose you are using a class hierarchy, and you wish the classes had a virtual method specific to the needs of your application
- For example, you are writing a game, and you want to give each animal a `hitPoints()` method
- Unfortunately, the class designer doesn't understand your application so there is no `hitPoints()` virtual
- 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

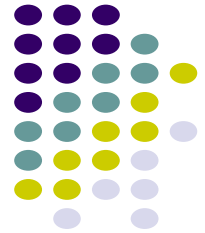
The Visitor Pattern

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

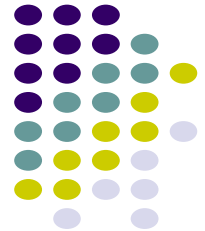
Make Your Classes Extensible

- 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); }
    /* … */
  };
  ```

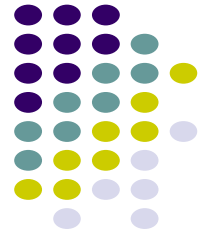
Class User: Create a visitor

- Now, I create a visitor that implements the virtual methods I wish were there
- ```
  struct HPVisitor
    : public AnimalVisitor {
   HPVisitor(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 hit points of the `Animal a` I created above
- ```
  int hp;
  a->accept(HitPointsVisitor(hp));
  cout <<  ((hp > 10) ? "safe" : "in danger");
  ```
- https://godbolt.org/z/86M7GPsrv
  ```

Best practice

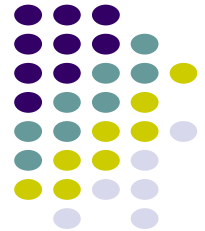


- If you are designing a class hierarchy where the best interface is unclear, add an `accept()` method as a customization point



USING DUCK-TYPED VARIANTS FOR PERFORMANT, EXTENSIBLE OO

Duck Typing



- 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
  - This is sometimes called "nominal typing" in the literature
- 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
  - This is sometimes called "structural typing" in the literature

Templates use duck typing

- `T square(auto x) { return x*x; }`
- `square(5); // OK`
- No need for `x` to inherit from a `HasMultiplication` class
- Enough that `operator*` makes sense to use here

Concepts improve duck typing

- We could create a `HasMultiplication` concept whenever we need one without needing to maintain or modify complex inheritance hierarchies

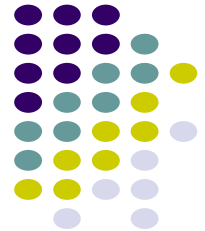
Dynamic dispatch

- C++ templates have always been duck typed, but are used for compile-time dispatch
  - Remember, `set<Animal> zoo` is not possible if `Animal` is a concept
- Traditionally, you would need an abstract `Animal` base class and virtuals to create a zoo with `set<unique_ptr<Animal>>`
- 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
  - More dynamic than our `Animal` concept (we can have a zoo with runtime dispatch)
  - Everything is *value-based*, no need to worry about `unique_ptr`, RAII, lifetimes, …
  - Often much better performance since we don't need to perform a heap allocation (A and B live directly in the variant)

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
  - It's related, but just enough to be confusing
- If `v` is a variant, and `c` is a callable, `visit(c, v)` 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
- Usually we encapsulate this in a helper function
- How does it compare to using virtuals or templates?
- https://godbolt.org/z/e4boMnzcc
  ```

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
- No brittle inheritance hierarchy while maintaining a reasonable level of static type safety
- As dynamic as traditional OO
  - A `set<Animal>` works great
  - Unlike our Concepts version
  - It's even simpler than the `set<unique_ptr<Animal>>` approach required by the traditional virtual method OO

Malleable (mallard?) typing

- Virtual function overrides are very tightly coupled, need ingto 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
  ```

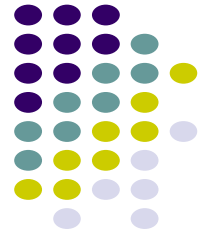
Users can add methods

- 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 template (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 `hitPoints()` "method" like we did before
  - ```
    overloaded hitPoints([](Cat &) { return 12 },
                         [](Dog &) { return 10});
    // Get hit points without knowing whether
    // a is a Cat or a Dog
    cout << visit(hitPoints, a); // Prints 12
    ```
- Note that this idiom relies on CTAD and aggregates deducing the template arguments for you
- https://godbolt.org/z/r43W5qjaT

Problems with Duck-Typed variants



- The notation is much uglier than calling a virtual function
- While the flexibility is nice, duck typing slightly 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

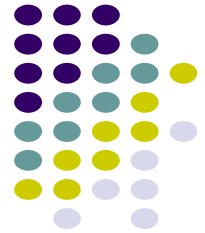
- Because it is ugly and not well-known, only prefer variant-based polymorphism over virtuals or concepts when there is a clear benefit
  - In practice, I use it a lot, but less than I do virtual functions or concepts

What to use

- The following table is admittedly overly simplistic
- But I usually find it helps me choose

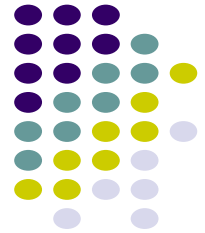
	Native notation	Performance	Dispatch	Duck typing	Value-based	Extension
Virtuals	✔		Dynamic			accept()
Concepts	✔	✔	Static	✔	✔	Duck provides
Variants		✔	Dynamic	✔	✔	overloaded

What about other techniques?



- We have learned about 3 approaches to writing "OO" code in C++
  - Inheritance/Virtuals
  - Concepts
  - Variant-based polymorphism
- And two client-side extensibility mechanisms
  - The visitor pattern (accept)
  - The overloaded pattern
- Are there more?
- In particular, you may be familiar with other techniques like static polymorphism or pre-C++20 template techniques like SFINAE and `void_t`

For the most part, C++20 offers better solutions



- While you may need to learn them to understand legacy code
- In my experience, C++20 techniques like concepts almost always solutions that are easier and better
- If you learn the approaches here, you should be well-equipped for most OO programming and learn other techniques when needed
  - E.g., for maintaining a legacy program, or if you really need nominally typed static dispatching, you might need to learn techniques static polymorphism, but they are much more complex and mostly unnecessary now

For the most part? Teaser

- There is one other technique called Type Erasure that we don't have time to cover today but deserves to be part of your toolkit
- I like to think of it as a way to have virtual functions even when there is no common base class (it substitutes something called a "type erasure model" for the base class)
- Or a way to have variants even if you don't know all the possibilities in advance
- The price you pay is that it requires a lot of complex boilerplate to set up
- There is a great writeup at https://davekilian.com/cpp-type-erasure.html
- Also, Rob Douglas has a great presentation on Type Erasure at https://www.youtube.com/watch?v=ZPk8HuyrKXU

Thank you!

Questions?