Object-oriented languages combine a confusing collection of motivations and features. Swiss-army knife approach: complex structure that does almost everything. Lego approach: simple pieces that fit together well.
C++ was invented because Vogon poetry wasn’t destructive enough.

Anonymous

Douglas Adams, *The Hitchhiker’s Guide to the Galaxy*
C++ Is Highly Pervasive

C++ is popular, I think for a lot of bad reasons.

One good reason for using C++ is the Standard Template Library (STL).
I find OOP technically unsound ... philosophically unsound ... [and] methodologically wrong.

Alexander Stepanov, developer of the C++ STL
Motivations

- SIMULA — [See Lounden and Lambert, Ch 5] templates, independent object; see heap allocation.
- C++ — originally C macros for information hiding
- SmallTalk — computational building blocks; see lambda calculus
- Java, Modula-3, Ada, OCaml, Rust — seeking, experimenting, innovating . . .
Object-Oriented Programming

“Before the industrial revolution, the firearms industry was a loose coalition of individual craftsmen. Each firearm was crafted by an individual gunsmith. The revolution was sparked when Eli Whitney received a large manufacturing contract to build muskets for the government. Whitney’s innovation was to divide the work so that each part was produced by a specialist to meet a specified standard. Each gunsmith focused on a single part, using sophisticated tools to optimize that task. The government, quickly realized that the standards would allow parts to be interchanged, greatly simplifying their firearm repair problems. The importance of object-oriented programming is comparable to that of Whitney’s interchangeable part innovation, and for many of the same reasons. Both redefine the unit of modularity so that workers produce subcomponents instead of complete solutions. The subcomponents are controlled by standards and can be interchanged across different products.”

What did *that* mean?
What I take from it is: interfaces, decomposition, and modularity. (This is a discussion in a separate unit on modularity.)

Reflecting on the wide-scale use of OO, one possible explanation for its appeal is code reuse (DRY) for data that is related.
We can identify several basic features of object-oriented languages:

• dynamic (at runtime) instantiation of templates
• encapsulation of state
• inheritance
• overriding
• dynamic dispatch

(And so many variant and interactions with other features!)
Encapsulation of State

Encapsulation of data is a given in modern programming languages. Encapsulation of data and operations is also a given. Every language has some degree of modularity.

Is the data mutable or not is a major issue. Because mutable data is inherently error prone some language to not allow state. This is less suitable for simulation of real-word objects.

Representing a train, for instance, as a data value in a program might be most natural by having the same train be at different geographic positions over the course of the program.
class/BankAccount.java
In *Effective Java*, Joshua Bloch makes this important recommendation:

*Classes should be immutable unless there’s a very good reason to make them mutable. If a class cannot be made immutable, limit its mutability as much as possible.*

This is very important advice about the design of data structures in your program whether or not the language is OO or not.
Immutability
Mutability
Mutability Leads To Disaster

HUGE MOLASSES TANK EXPLODES
IN NORTH END; 11 DEAD, 50 HURT

Giant Wave of 2,300,000 Gallons of Molasses, 50 Feet High, Sweeps
Everything Before It—100 Men, Women and Children Caught in
Sticky Stream—Buildings, Vehicles and L Structure Crushed

35 STATES ON DRY LAW LIST

Amendment Ratified by Five Yester-
day—One More Needed—Predict Nation Day July 1

SECRET IN PEACE CONGRESS

Search for More Victims During
the Night

No Escape From Gigantic Wave
of Fluid

INTERNAL EXPLOSION WAS
CAUSE, SAYS STATE CHEMIST
Sharing is cheap, but buggy; copying (large objects) is expensive, but safe. Immutable objects can be shared without problems, Always design for immutability; optimize later.

*premature optimization is the root of all evil*

Knuth, 1974 Turing Award lecture
Sharing Mutable

sharing

mutable
Sharing Immutable

sharing

immutable
Let’s move on to something more intrinsically OO-like: inheritance.
class A {
    private int x = 0;
}

class B extends A {
}

The class B has the field x (even though it cannot access it directly). We say B inherits x. (The code `new B().x` does not compile because access is denied.)
A class can be instantiated to create new objects and subclassed to create new classes.

```java
class A {
    private int x = 0;
    int m() { x=x+1; return x; } // mutation!
    int n() { x=x-2; return x; } // mutation!
}

class B extends A {
    int o() { x=x*3; return x; } // Error: private!
    int p() { return 3*n(); } // return 3x-6
}
```

The code `(new B()).m()` is legal as an instance of B does have a method m. We say the class B inherits the methods m and n, and the field x.
Subtype Polymorphism

interface A {
    int m();
}
class B implements A {
    private int i = 0;
    void m() { i=i+1; }
}
class C implements A {
    private float x = 0.0f;
    void m() { x=x+1.0f; }
}
class D {
    void p (A a) { a.m(); }
}

The method p can be invoked on objects of different classes, e.g.,
(new D()).p(new B()) and (new D()).p(new C()), because of their interface
is just as good as the interface required of the argument to p.
An interface sometimes implies just the set of operations defined on an object (as in Java), but the polymorphism extends to fields as well.

We called the subtype polymorphism in an earlier unit.

“B is an A” and “C is an A”.
We can substitute B or C for A.
Liskov substitution principle.
BARBARA LISKOV
Developed the Liskov substitution principle
Barbara Liskov (–)

When she was still a young professor at the Massachusetts Institute of Technology, she led the team that created the first programming language that did not rely on goto statements. The language, CLU (short for “cluster”), relied on an approach she invented — data abstraction — that organized code into modules. Every important programming language used today, including Java, C++ and C#, is a descendant of CLU.

In 2008, Liskov won the Turing Award.
Object-Oriented Programming

Organize data structures into a hierarchy

```java
class Point {
    int x, y;
}

class Rectangle extends Point {
    int h, w;
}

class Circle extends Point {
    int radius;
}
```
Overriding

Completely ignore the inheritance.

override/Animals.java
```java
class A {
    int x = 0;
    void incr() { x=x+1; }
}

class B extends A {
    int y = 1;
    @Override
    void incr() { super.incr(); y=y+1; }
}
```
class A {
    int x = 0;
    int m() { x=x-1; return x; }
    int n() { x=x+2; return this.m(); }
}

class B extends A {
    int m() { x=x*3; return super.m(); }
}

Consider (new B()).n(); 5 is returned.
Inheritance and dynamic dispatch in Java:

class/Shapes.java
Calling Procedure

- `call P(A)` – compiler looks up address of `P` and jumps to instruction
- `call P(A)` – (overloading) compiler chooses from among several procedures based on the static types of arguments
- `o.P()` – (dynamic dispatch) the runtime system chooses from among several procedures based on the subtype of object `o`

Note that static type checking is possible in all cases. Also, dynamic dispatch is not polymorphism—it is a strategy to resolve which of disparate functions (all of the same form) to call.
Narrowing

Coercions can be classified into those that preserve information (widenings) and those that lose information (narrowings). The coercion `int` to `long` is a widening; `int` to `short` is a narrowing. See the table of Java coercions at

```
/ ryan/java/language/java-data.html
```

The terms apply to the OO hierarchy as well. OO programming often use narrowing which defeats the purpose of strong typing. See the Java program example:

```
class/Widening.java
```
Dynamic Dispatch

In Java overriding is default (not static); in C++ overriding is not default (virtual).

Inheritance and overriding in C++

class/inheritance.cc
Object-Oriented Programming in Ada

OO language use classes for encapsulation and subtyping. Ada, Modula-3, Oberon have modules and extensible records.

objects/shape_main.adb
Vtable

Scott, Figures 10.4, pages 558.
Scott, Figures 10.5, pages 559.
Subtyping Subtleties

Subtype polymorphism is not without its subtleties.

- Easy to confuse overloading with overriding.
  
  ```java
  equals/Mistake.java
  ```

- Favor aspects over inheritance

- Subtypes and arrays. Arrays are neither co- nor contra-variant as we saw in the unit on typing.
Subtyping and Aspects

Equality, for example, does not mix well with subtypes.

```java
aspect/Point.java
aspect/SubPoint.java
aspect/Aspect.java
```

See *Effective Java Programming Language Guide* by Joshua Bloch. Chapter 3: Methods Common to All Objects.

BTW Java SE 14 (March 2020) introduced records as a “preview” feature.

```java
record Point(int x, int y) {} 
```

is a final class with immutable components and appropriate accessors, constructors, equals, hashCode, and toString implementations.
Assignment and subtyping do not mix sometimes (as we saw earlier).

misc/Sub.java
misc/ASE.java
Contravariance

Vegetable <: Food
Bat     <: Mammal

interface I {
    Mammal f (Vegetable v);
}

class C implements I {
    Bat f (Food d) { return new Bat(); }  
}

Works in Java 1.5, but not Java 1.4: no contravariance.

class/Contra.java class/Vegetarian.java
Design Patterns

Idioms for using object-programming languages for things other than what it was designed for.
Factory Pattern, Singleton Pattern, Visitor Pattern, ...
Singleton Design Pattern

aspect/Singleton.java
Visitor Design Pattern

The Gang of Four defines the visitor as:

Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
The visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A result of this separation is the ability to add new operations to existing object structures without modifying those structures.

In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function.
Object-Oriented

Each interpretation must be applied to each kind; if we add a new kind, we must implement each interpretation for it; and if we add a new interpretation, we must implement it for each kind.

Appel 2nd, Figure 4.6, page 94. Kinds versus Interpretations, orthogonal axes of modularity.
Double Dispatch

1. When the accept method is called in the program, its implementation is chosen based on both: the dynamic type of the element and the static type of the visitor.

2. Then the associated visit method is called, its implementation is chosen based on both: the dynamic type of the visitor and the static type of the element.
// Each instance of visitor is a different
// suite of virtual methods for the structure.

class Visitor {
  visit (SomeStructure n) {
    // Do stuff on this structure 'n', then continue
    // and select right visitor code for a subpart by
    // dynamic dispatch on the specific kind subpart.
    instanceSubPartOfN.accept (this);
  }
}

class SubPart {
  accept (Visitor v) {
    // Statically select action based on the
    // kind of this structure.
    v.visit(this);
  }
}

With the object-oriented style, each interpretation is just a method in all the classes. It is easy and modular to add a new kind: All the interpretations of that kind are grouped together as methods of the new class. But it is not modular to add a new interpretation: A new method must be added to every class.

Appel, 2nd, page 95-96.
Each structure merely implements `accept` and calls overloaded `visit` with itself as argument.

The visitor does all the work for each structure and controls the traversal by calling `accept` (dynamic dispatch).
Visitor Design Pattern

Dynamic dispatch on the `accept` method in the tree structure with the visitor as an argument will call the correct, overloaded, instance method in the visitor with all the instance members of the tree node. The visitor may then choose to traverse the substructure by calling `accept` on the instances of the subtree.
Math Has the Abstractions

In category theory, the concept of catamorphism (from Greek: κατά = downwards or according to; μορφή = form or shape) denotes the unique homomorphism from an initial algebra into some other algebra.

In functional programming, catamorphisms provide generalizations of folds on lists to arbitrary algebraic data types, which can be described as initial algebras. Consider the Haskell class Foldable.
Some OO Variations

- The programming language Eifel
- Multiple inheritance
Eiffel

Eiffel is not type-safe.

1. Replace result type by a subtype of the one occurring in the superclass
2. Hide inherited methods in a subclass

A routine is a CAT (Changing Availability or Type) if some redefinition changes its export status or the type of one of its arguments. A call is a catcall if some redefinition of the routine would make it invalid because of a change of export status or argument type.

Not implemented, may not solve the problem, and may be so restrictive nothing interesting passes.
The term “mixin” was first coined in the Flavors community [Weinreb81]. It has been used in a number of contexts; see, e.g., [Keene88], [Bracha90], [Booch94], [Taligent94], [VanHilst96]. In the context of multiple-inheritance languages like C++, a mixin is a partial class that implements a small part of the functionality of a large class (See [Booch94]). A mixin is distinguished from the other partial classes by the intent of its designers. A typical, partial class is like a mostly completed puzzle, with deferred methods representing a few missing pieces. A mixin is like a single puzzle piece, with final methods representing the tabs on a puzzle piece and deferred methods representing the indentations into which tabs of other pieces fit.

Multiple Inheritance

In Ada, generics are even more useful, because you can identify explicitly the set of operations a formal type parameter is supposed to have. Instantiating a generic is the most flexible kind of “interface” inheritance, particularly in Ada, where the operations on a type can be effectively renamed as part of the instantiation. Overall, our belief is that robust support for generics, plus the “multiple namespace inheritance” already provided by Ada’s “with” and “use” clauses, obviates much of the need for “linguistic” support for multiple inheritance. Whatever is left can be readily and flexibly handled by the sophisticated type composition facilities provided by Ada 95, including the concept of an “access discriminant” which allows one object to refer to another object, including possibly its enclosing object.

Tucker Taft, 1997, forum post