Object-Oriented Programming

Dispatch

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Dispatch

Selecting which implementation of an operation (method or function) to call

- *static dispatch*

- *dynamic dispatch*

Static Dispatch

Selecting which implementation of an operation (method or function) to call at compile time

- Which operation (method or function) will be called is determined at compile time

- As fast as a call can be made

- Preferred by the compiler for these reasons

C++ Static Dispatch

```
void f() {}

class C {
public:
    void f() {}
    virtual void m() {}
    static void s() {}
};

    f();
    C c;
    c.f();
    c.m();
    c.s();

    C* pc = &c;
    pc->f();
//    pc->m();
    pc->s();

    C& rc = c;
    rc.f();
//    rc.m();
    rc.s();
```

- The compiler uses static dispatch whenever it can:

  free functions

  non-virtual methods

  static methods

  method calls from non-pointer and non-reference variables

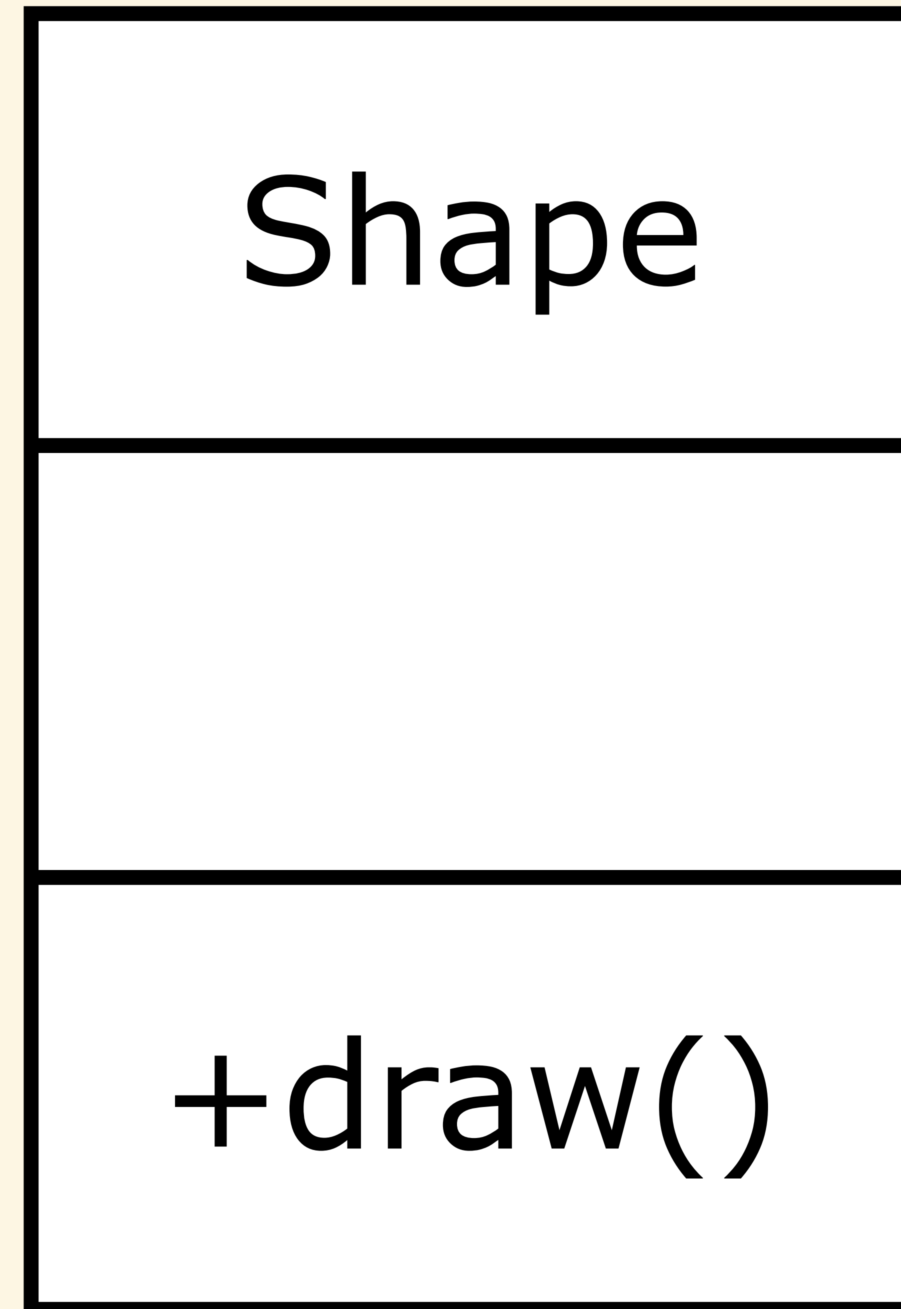
- Why? Less code to generate (program smaller)
- Why? Faster to call

Dynamic Dispatch

Selecting which implementation of an operation (method) to call at run time

- For virtual method calls whose object is a *pointer* or *reference*
- Requires more code than static dispatch
- Slightly slower to call
- Compilers have optimized dynamic dispatch, so the speed disadvantage is smaller than it used to be

Original class Shape




```
class Shape {
public:
    virtual void draw();
};
```

Problem for Dynamic Dispatch

```
class Shape {
public:
    virtual void draw();
};

void apply(Shape& shape) {
    // NOTE: virtual method call
    shape.draw();
}

void apply(Shape* pshape) {
    // NOTE: virtual method call
    pshape->draw();
}

Shape shape;
apply(shape);
apply(&shape);
```

- Methods through the pointer parameter `pshape` and the reference parameter `shape` calls **may not exist yet**
- The functions `apply()` are compiled so that for **any class that inherits from the class Shape, the function calls the proper method**
- Even though the parameter type at the time of compilation was a `Shape*` or a `Shape&`, the `apply()` functions must call the Shape methods for Shape objects and derived methods for objects of derived classes

Add class Circle

```
class Circle : public Shape {
public:
    void draw() override;
};
```

Problem for Dynamic Dispatch

```cpp
// Shape files
class Shape {
public:
    virtual void draw();
};

void apply(Shape& shape) {
    // NOTE: virtual method call
    base.draw();
}

void apply(Shape* shape) {
    // NOTE: virtual method call
    pbase->draw();
}

// Circle files
class Circle : public Shape {
public:
    void draw() override;
};

Circle circle;
apply(circle);
apply(&circle);
```

- New methods now exist
- Original `apply()` code **existed and was compiled long before class Circle was created**
- The original `apply()` code in this case **must call `Circle::draw()`**

How? vtable

```
// Shape files
class Shape {
public:
    virtual void draw();
};

void apply(Shape& shape) {
    // NOTE: virtual method call
    base.draw();
}

void apply(Shape* shape) {
    // NOTE: virtual method call
    pbase->draw();
}

// Circle files
class Circle : public Shape {
public:
    void draw() override;
};

Circle circle;
apply(circle);
apply(&circle);
```

- At compile time, the *virtual table* or *vtable* is created to store the information required for the dynamic dispatch of virtual methods for a class
- For every object of a class that has a *vtable*, the object has a pointer to the class vtable
- Method calls go through the vtable for the class of an object
- We will examine the *vtable* structure in detail at a later date