

# Section Overview

---

## Polymorphism and Inheritance

- What is Polymorphism?
- Using base class pointers
- Static vs. dynamic binding
- Virtual functions
- Virtual destructors
- The override and final specifiers
- Using base class references
- Pure virtual functions and abstract classes
- Abstract classes as interfaces

# Polymorphism

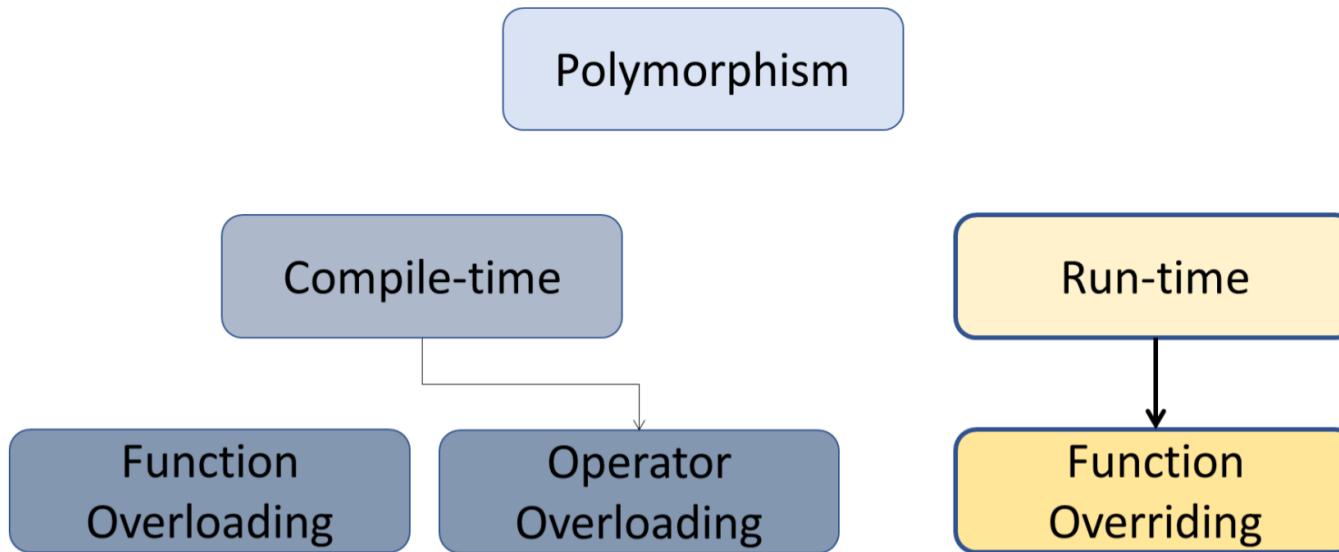
---

## What is Polymorphism?

- Fundamental to Object-Oriented Programming
- Polymorphism
  - Compile-time / early binding / static binding
  - **Run-time / late binding / dynamic binding**
- Runtime polymorphism
  - Being able to assign different meanings to the same function at run-time
- Allows us to program more abstractly
  - Think general vs. specific
  - Let C++ figure out which function to call at run-time
- Not the default in C++, run-time polymorphism is achieved via
  - Inheritance
  - Base class pointers or references
  - virtual functions

# Polymorphism

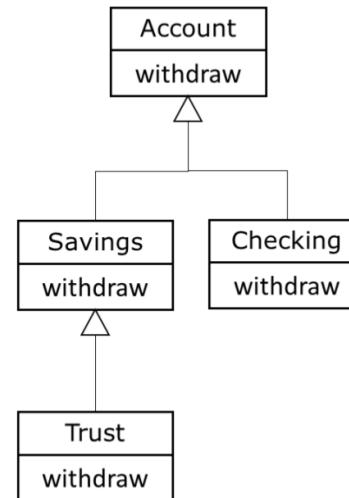
Types of Polymorphism in C++?



# Polymorphism

An non-polymorphic example – Static Binding

```
Account a;  
a.withdraw(1000); // Account::withdraw()  
  
Savings b;  
b.withdraw(1000); // Savings::withdraw()  
  
Checking c;  
c.withdraw(1000); // Checking::withdraw()  
  
Trust d;  
d.withdraw(1000); // Trust::withdraw()  
  
Account *p = new Trust();  
P->withdraw(1000); // Account::withdraw()  
// should be  
// Trust::withdraw()
```



---

BEGINNING C++ PROGRAMMING  
What is Polymorphism?

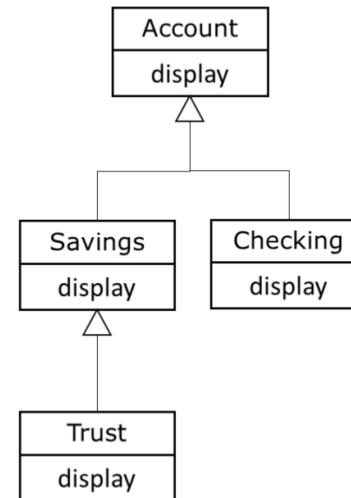


POLYMORPHISM  
S16-CPP-0040-3

# Polymorphism

## An non-polymorphic example – Static Binding

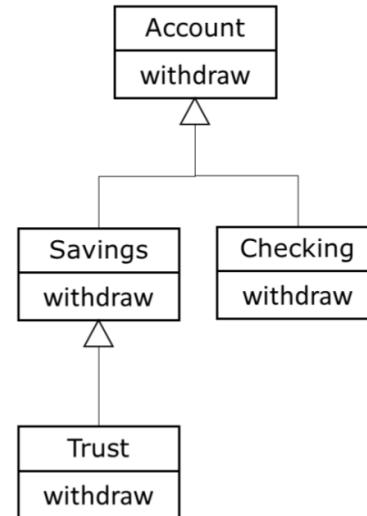
```
void display_account(const Account &acc) {  
    acc.display();  
    // will always use Account::display  
}  
  
Account a;  
display_account(a);  
  
Savings b;  
display_account(b);  
  
Checking c;  
display_account(c);  
  
Trust d;  
display_account(d);
```



# Polymorphism

## A polymorphic example – Dynamic Binding

```
Account a;  
a.withdraw(1000); // Account::withdraw()  
  
Savings b;  
b.withdraw(1000); // Savings::withdraw()  
  
Checking c;  
c.withdraw(1000); // Checking::withdraw()  
  
Trust d;  
d.withdraw(1000); // Trust::withdraw()  
  
Account *p = new Trust();  
p->withdraw(1000); // Trust::withdraw()
```



withdraw method is virtual in Account

BEGINNING C++ PROGRAMMING  
What is Polymorphism?



POLYMORPHISM  
S16-CPP-0040-5

# Polymorphism

## A polymorphic example – Dynamic Binding

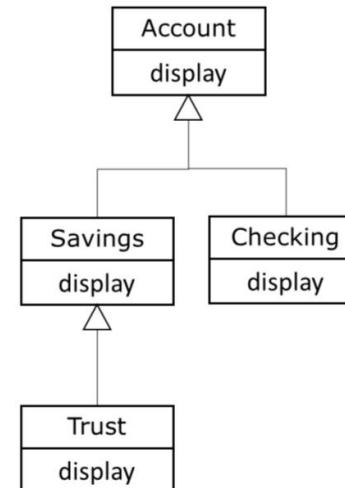
```
void display_account(const Account &acc)
{
    acc.display();
    // will always call the display method
    // depending on the object's type
    // at RUN-TIME!
}

Account a;
display_account(a);

Savings b;
display_account(b);

Checking c;
display_account(c);

Trust d;
display_account(d);
```



display method is virtual in Account

# Polymorphism

---

Using a Base class pointer

- For dynamic polymorphism we must have:
  - Inheritance
  - Base class pointer or Base class reference
  - virtual functions

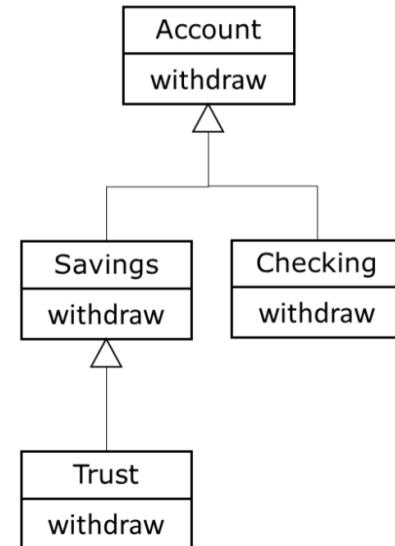
# Polymorphism

Using a Base class pointer

```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

p1->withdraw(1000);      //Account::withdraw
p2->withdraw(1000);      //Savings::withdraw
p3->withdraw(1000);      //Checking::withdraw
p4->withdraw(1000);      //Trust::withdraw

// delete the pointers
```



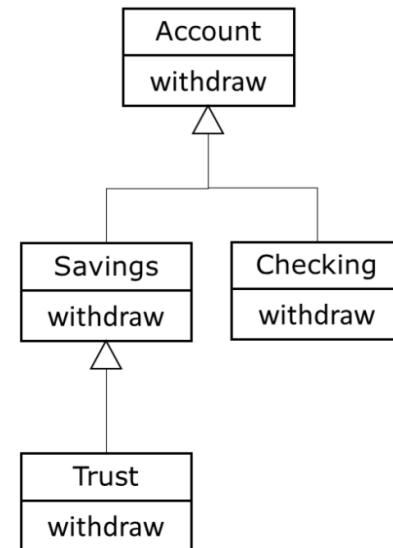
# Polymorphism

Using a Base class pointer

```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

Account *array [] = {p1,p2,p3,p4};

for (auto i=0; i<4; ++i)
    array[i]->withdraw(1000);
```



# Polymorphism

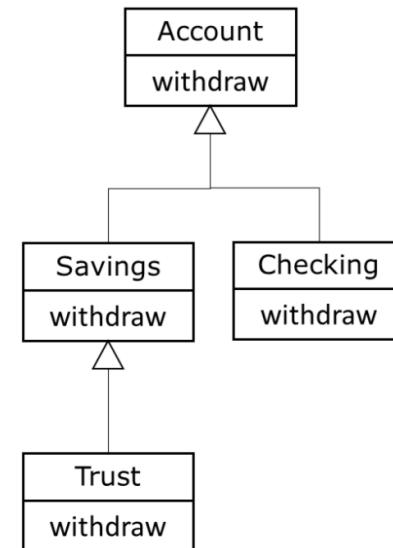
Using a Base class pointer

```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

vector<Account *> accounts
    {p1, p2, p3, p4 };

for (auto acc_ptr: accounts)
    acc_ptr->withdraw(1000);

// delete the pointers
```



# Polymorphism

---

## Virtual functions

- Redefined functions are bound statically
- Overridden functions are bound dynamically
- Virtual functions are overridden
- Allow us to treat all objects generally as objects of the Base class

# Polymorphism

---

Declaring virtual functions

- Declare the function you want to override as virtual in the Base class
- Virtual functions are virtual all the way down the hierarchy from this point
- Dynamic polymorphism only via Account class pointer or reference

```
class Account {  
public:  
    virtual void withdraw(double amount);  
    . . .  
};
```

# Polymorphism

---

Declaring virtual functions

- Override the function in the Derived classes
- Function signature and return type must match EXACTLY
- Virtual keyword not required but is best practice
- If you don't provide an overridden version it is inherited from its base class

```
class Checking : public Account {  
public:  
    virtual void withdraw(double amount);  
    . . .  
};
```

# Polymorphism

---

## Virtual Destructors

- Problems can happen when we destroy polymorphic objects
- If a derived class is destroyed by deleting its storage via the base class pointer and the class a non-virtual destructor. Then the behavior is undefined in the C++ standard.
- Derived objects must be destroyed in the correct order starting at the correct destructor

# Polymorphism

---

## Virtual Destructors

- Solution/Rule:
  - If a class has virtual functions
  - ALWAYS provide a public virtual destructor
  - If base class destructor is virtual then all derived class destructors are also virtual

```
class Account {  
public:  
    virtual void withdraw(double amount);  
    virtual ~Account();  
    . . .  
};
```

# Polymorphism

---

The override specifier

- We can override Base class virtual functions
- The function signature and return must be EXACTLY the same
- If they are different then we have redefinition NOT overriding
- Redefinition is statically bound
- Overriding is dynamically bound
- C++11 provides an override specifier to have the compiler ensure overriding

# Polymorphism

---

## The override specifier

```
class Base {
public:
    virtual void say_hello() const {
        std::cout << "Hello - I'm a Base class object" << std::endl;
    }
    virtual ~Base() {}
};

class Derived: public Base {
public:
    virtual void say_hello() { // Notice I forgot the const - NOT OVERRIDING
        std::cout << "Hello - I'm a Derived class object" << std::endl;
    }
    virtual ~Derived() {}
};
```

# Polymorphism

---

The override specifier

Base:

```
virtual void say_hello() const;
```

Derived:

```
virtual void say_hello();
```

# Polymorphism

---

The override specifier

```
Base *p1 = new Base();
p1->say_hello();      // "Hello - I'm a Base class object"

Base *p2 = new Derived();
p2->say_hello();      // "Hello - I'm a Base class object"
```

- Not what we expected
- say\_hello method signatures are different
- So Derived **redefines** say\_hello instead of overriding it!

# Polymorphism

---

## The override specifier

```
class Base {
public:
    virtual void say_hello() const {
        std::cout << "Hello - I'm a Base class object" << std::endl;
    }
    virtual ~Base() {}
};

class Derived: public Base {
public:
    virtual void say_hello() override { // Produces compiler error
        // Error: marked override but does not override
        std::cout << "Hello - I'm a Derived class object" << std::endl;
    }
    virtual ~Derived() {}
};
```

# Polymorphism

---

The `final` specifier

- C++11 provides the `final` specifier
  - When used at the class level
  - Prevents a class from being derived from
- When used at the method level
- Prevents virtual method from being overridden in derived classes

# Polymorphism

---

final class

```
class My_class final {  
    . . .  
};  
  
class Derived final: public Base {  
    . . .  
};
```

# Polymorphism

---

final method

```
class A {  
public:  
    virtual void do_something();  
};  
  
class B: public A {  
    virtual void do_something() final; // prevent futher overriding  
};  
  
class C: public B {  
    virtual void do_something(); // COMPILER ERROR - Can't override  
};
```

# Polymorphism

---

## Using Base class references

- We can also use Base class references with dynamic polymorphism
- Useful when we pass objects to functions that expect a Base class reference

# Polymorphism

---

Using Base class references

```
Account a;  
Account &ref = a;  
ref.withdraw(1000); // Account::withdraw
```

```
Trust t;  
Account &ref1 = t;  
ref1.withdraw(1000); // Trust::withdraw
```

# Polymorphism

---

Using Base class references

```
void do_withdraw(Account &account, double amount) {  
    account.withdraw(amount);  
}
```

```
Account a;  
do_withdraw(a, 1000); // Account::withdraw
```

```
Trust t;  
do_withdraw(t, 1000); // Trust::withdraw
```

# Polymorphism

---

## Pure virtual functions and abstract classes

- Abstract class

- Cannot instantiate objects
- These classes are used as base classes in inheritance hierarchies
- Often referred to as Abstract Base Classes

- Concrete class

- Used to instantiate objects from
- All their member functions are defined
  - Checking Account, Savings Account
  - Faculty, Staff
  - Enemy, Level Boss

# Polymorphism

---

## Pure virtual functions and abstract classes

- Abstract base class
  - Too generic to create objects from
    - Shape, Employee, Account, Player
  - Serves as parent to Derived classes that may have objects
  - Contains at least one pure virtual function

# Polymorphism

---

## Pure virtual functions and abstract classes

- Pure virtual function
  - Used to make a class abstract
  - Specified with “=0” in its declaration

```
virtual void function() = 0; // pure virtual function
```

- Typically do not provide implementations

# Polymorphism

---

## Pure virtual functions and abstract classes

- Pure virtual function
  - Derived classes MUST override the base class
  - If the Derived class does not override then the Derived class is also abstract
  - Used when it doesn't make sense for a base class to have an implementation
    - But concrete classes must implement it

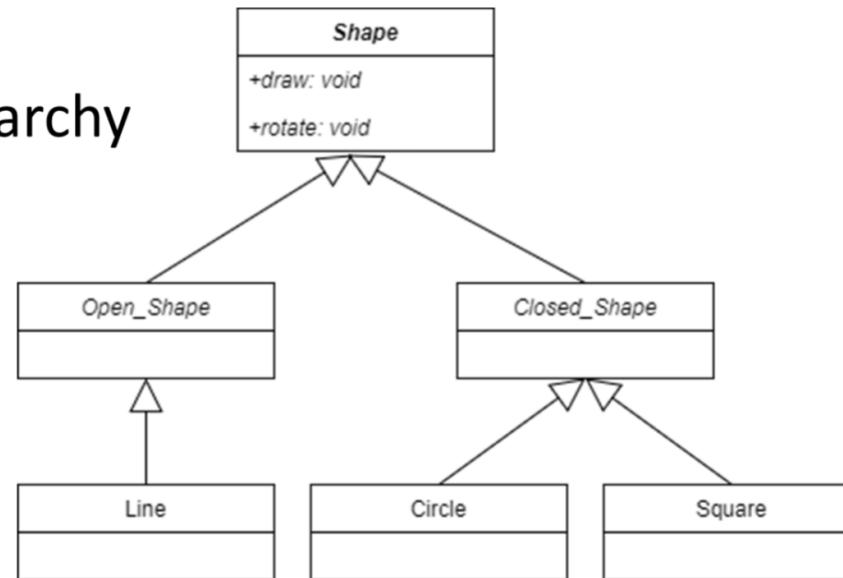
```
virtual void draw() = 0;      // Shape  
virtual void defend() = 0;    // Player
```

# Polymorphism

---

Pure virtual functions and abstract classes

## Shape Class Hierarchy



# Polymorphism

---

Pure virtual functions and abstract classes

```
class Shape {                      // Abstract
private:
    // attributes common to all shapes
public:
    virtual void draw() = 0;        // pure virtual function
    virtual void rotate() = 0;      // pure virtual function
    virtual ~Shape();
    . . .
};
```

# Polymorphism

---

Pure virtual functions and abstract classes

```
class Circle: public Shape {           // Concrete
private:
    // attributes for a circle
public:
    virtual void draw() override {
        // code to draw a circle
    }
    virtual void rotate() override {
        // code to rotate a circle
    }
    virtual ~Circle();
    . .
};

};
```

# Polymorphism

---

Pure virtual functions and abstract classes

Abstract Base class

- Cannot be instantiated

```
Shape shape;           // Error
Shape *ptr = new Shape(); // Error
```

- We can use pointers and references to dynamically refer to concrete classes derived from them

```
Shape *ptr = new Circle();
ptr->draw();
ptr->rotate();
```

# Polymorphism

---

What is using a class as an interface?

- An abstract class that has only pure virtual functions
- These functions provide a general set of services to the user of the class
- Provided as public
- Each subclass is free to implement these services as needed
- Every service (method) must be implemented
- The service type information is strictly enforced

# Polymorphism

---

## A Printable example

- C++ does not provide true interfaces
- We use abstract classes and pure virtual functions to achieve it
- Suppose we want to be able to provide `Printable` support for any object we wish without knowing its implementation at compile time

```
std::cout << any_object << std::endl;
```

- `any_object` must conform to the `Printable` interface

# Polymorphism

---

## An Printable example

```
class Printable {
    friend ostream &operator<<(ostream &, const Printable &obj);
public:
    virtual void print(ostream &os) const = 0;
    virtual ~Printable() {};
    . . .
};

ostream &operator<<(ostream &os, const Printable &obj) {
    obj.print(os);
    return os;
}
```

# Polymorphism

---

An Printable example

```
class Any_Class : public Printable {  
public:  
    // must override Printable::print()  
    virtual void print(ostream &os) override {  
        os << "Hi from Any_Class" ;  
    }  
    . . .  
};
```

# Polymorphism

---

## An Printable example

```
Any_Class *ptr= new Any_Class();
cout << *ptr << endl;

void function1 (Any_Class &obj) {
    cout << obj << endl;
}

void function2 (Printable &obj) {
    cout << obj << endl;
}
function1(*ptr);           // "Hi from Any_Class"
function2(*ptr);           // "Hi from Any_Class"
```

# Polymorphism

---

A Shapes example

```
class Shape {  
public:  
    virtual void draw() = 0;  
    virtual void rotate() = 0;  
    virtual ~Shape() {};  
    . . .  
};
```

# Polymorphism

---

A Shapes example

```
class Circle : public Shape {  
public:  
    virtual void draw() override { /* code */ };  
    virtual void rotate() override { /* code */ };  
    virtual ~Circle() {};  
    . . .  
};
```

# Polymorphism

---

A Shapes example

```
class I_Shape {  
public:  
    virtual void draw() = 0;  
    virtual void rotate() = 0;  
    virtual ~I_Shape() {};  
    . . .  
};
```

# Polymorphism

---

## A Shapes example

```
class Circle : public I_Shape {  
public:  
    virtual void draw() override { /* code */ };  
    virtual void rotate() override { /* code */ };  
    virtual ~Circle() {};  
    . . .  
};
```

- Line and Square classes would be similar

# Polymorphism

---

## A Shapes example

```
vector< I_Shape * > shapes;

I_Shape *p1 = new Circle();
I_Shape *p2 = new Line();
I_Shape *p3 = new Square();

for (auto const &shape: shapes) {
    shape->rotate();
    shape->draw();
}

// delete the pointers
```