# POLYMORPHISM

# OOP CORE FEATURE REVIEW

Encapsulation: **Controlling access to object data and behavior**

Abstraction: **Only exposing relevant object data and behavior per interaction**

Inheritance: **Sharing object data and behavior to eliminate code repetition**

Polymorphism: **Sharing a common interface for related object types**

# POLYMORPHISM

**Polymorphism**

enables run time resolution of data type for derived classes accessed from a common interface of the base class

typically implemented via inheritance and virtual functions base reference or pointer used to access derived objects

**Virtual function**

a redefined function where the calling object determines if the base or derived version of the function is called during runtime

# EARLY/STATIC BINDING

**Concept** type determined at compile time

**Syntax**

```
class Parent {                          // base class
      void output() { }
};
class Child: public Parent{             // derived class
      void output() { }                 // redefined function
};

Child c();                              // child object
Parent &p1 = c;                         // parent reference to child object
Parent *p2 = &c;                        // parent pointer to child object

c.output();                             // call child output function
p1.output();                            // call parent output function
p2->output();                           // call parent output function
```

# LATE/DYNAMIC BINDING

**Concept** type determined at run time (polymorphic effect)

**Syntax**

```
class Parent {
    virtual void output() { }          // virtual function
};
class Child: public Parent{
    void output() override { }         // redefined virtual function
};

Child c();
Parent &p1 = c;
Parent *p2 = &c;

c.output();                            // call child output function
p1.output();                           // call child output function (virtual functions)
p2->output();                          // call child output function (virtual functions)
```

# LATE/DYNAMIC BINDING

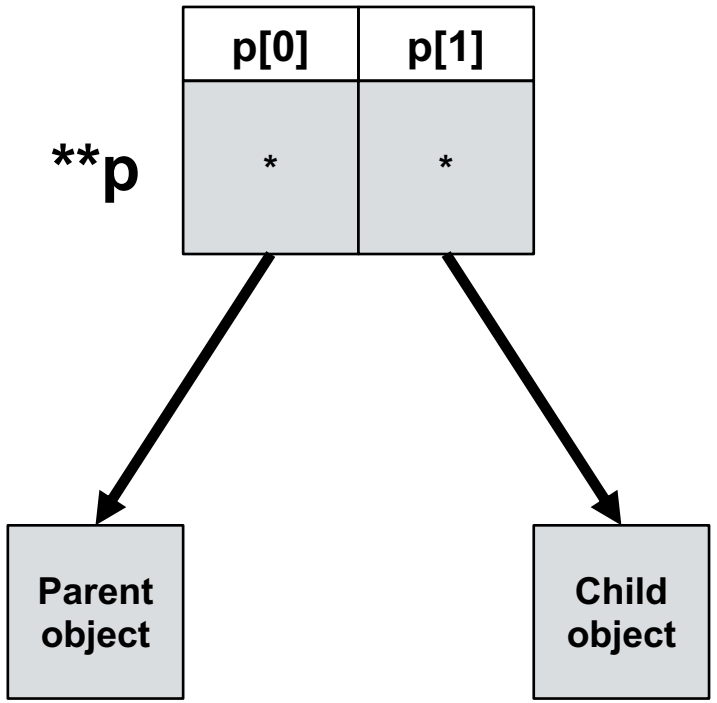
**Concept** type is determined at run time

**Syntax**

```
class Parent {
      virtual void output() { }          // virtual function
};
class Child: public Parent{
      void output() override { }         // redefined virtual function
};

Parent **p = new Parent*[2];             // dynamic array of parent object pointers
p[0] = new Parent();                     // store a pointer to a parent object
p[1] = new Child();                      // store a pointer to a child object

for(int i=0; i<2; ++i) {
      p[i].output();                     // output function will match object type
}
```

# LATE/DYNAMIC BINDING

```
class Parent {
      virtual void output() { }
};
class Child: public Parent{
      void output() override { }
};

Parent **p = new Parent*[2];
p[0] = new Parent();
p[1] = new Child();

for(int i=0; i<2; ++i) {
            p[i].output();
}
```



# COVARIANT RETURN

**Concept** redefined function with child instead of parent return type

**Syntax**

```
class Parent {
    virtual Parent* func() { return this; }     // virtual function with parent return
};
class Child: public Parent{
    Child* func() override { return this; }   // redefined with child return
};
```

# VIRTUAL DESTRUCTOR

**Concept** ensure that the child destructor is called for a child object

**Syntax**

```
class Parent {
      virtual ~Parent;                  // virtual destructor
};

class Child: public Parent{
       ~Child();
};
```

# ABSTRACT CLASSES

**Concept**

implement an interface to be used indirectly through inheritance
an abstract class cannot be instantiated, but it's derived classes can
requires one or more pure virtual functions

a pure virtual function lacks a definition
it is intended to be implemented in a derived class

**Syntax**

```
class Parent {                                  // abstract class
    virtual void output() const = 0;            // pure virtual function
};

class Child: public Parent{
    void output() const override { };           // overridden function has definition
};
```

# ABSTRACT CLASSES

Syntax

```
class Parent {
      virtual void output() const = 0;      // pure virtual function
};
class Child: public Parent{
      void output() const override { };     // overridden function has definition
};

Parent *p;                                  // parent pointers can be created
p = new Child();                            // child objects can be instantiated

p = new Parent();                           // compiler error
                                            // cannot instantiate abstract class objects
```

# SLICING PROBLEM

**Concept** when child objects copied to parent objects data is lost

**Syntax**

```
class Parent { };                    // base class
class Child: public Parent{ };       // derived class

Parent p;                            // parent object
Child c;                             // child object
p = c;                               // copy child object to parent, child data lost


Parent *p;                           // parent pointer
Child *c = new Child();              // dynamic child object
p = c;                               // copy pointers not objects, child data safe
```

# UPCAST VS DOWNCAST

**Upcast** conversion from child pointer to parent pointer (implicit or explicit are legal)
**Downcast** conversion from parent pointer to child pointer (only explicit is legal)

**Syntax**

```
class Parent { };                       // base class
class Child: public Parent{ };          // derived class

Parent p();                             // parent object
Child c();                              // child object

Parent *pp;                             // parent pointer
Child *cp;                              // child pointer

pp = &c;                                // implicit upcast, legal
cp = &p;                                // implicit downcast, illegal
cp = static_cast<Child*>(&p);           // explicit downcast, legal but troublesome
```

# UPCAST VS DOWNCAST

Syntax

```
class Parent { };                                   // base class
class Child: public Parent{                         // derived class
      void childFunction();                         // unique derived class function
};

Parent **p = new Parent*[2];                        // array of parent pointers
p[0] = new Parent();                                // point to a parent object
p[1] = new Child();                                 // point to a child object

for(int i=0; i<2; ++i) {                            // iterate through array
      if( dynamic_cast<Child*>(p[i]) ) {            // return true if p[i] is-a Child*
            Child *c = dynamic_cast<Child*>(p[i]);  // if so, explicit cast to Child*
            c->childFunction();                     // safe to call child function
      }
}
```

# POLYMORPHISM SUMMARY

**Recommendations**

use virtual functions or pure virtual functions for abstract classes

```
virtual output() const {};
virtual output() const = 0;
```

use virtual destructors

```
virtual ~Parent();
```

use base pointers or references to access derived objects

```
Parent *p = new Child();
```

use base pointers in data structures of derived objects

```
Parent **p = new Parent*[1000];
```

copy base pointers or references instead of derived objects

```
Parent *p1, *p2 = new Child();
p1 = p2;
```

# FUNCTION OBJECT (FUNCTORS)

**Concept**

a class that instantiates objects which act like functions
a functor can maintain state between function calls
requires () operator overload

**Syntax**

```
class Add {
      int count = 0;                          // track number of function calls
public:
      int operator() (int x, int y) {         // overloaded () operator
            count++;                          // an example of state (number of operations)
            return x+y;
      }
};

Add a;                                        // instantiate a function object
cout << a(5, 6);                              // use the object as a function call
```