

MS-3832 – Adv. C++ Object-Oriented Programming

Polymorphism

- ❖ What is it?
 - Multiple fns with the SAME name
 - Different from overloading!
 - ◆ Static binding rules don't apply!!
- ❖ Achieved via "Virtual" functions
- ❖ Useful when expanding classes

Binding

- ◆ Determines which function is invoked by a function call statement.
- ❖ Static
 - Determined at compile time, by object's data type.
- ❖ Dynamic
 - Determined at run time.

Static Binding

- ❖ Follows the normal rules of strong data typing of objects
- ❖ Uses the normal data typing rules of the language to determine which class functions go with the object.

Dynamic Binding

- ❖ Uses the object's data type to determine, at run time, which function body is invoked by a message (ie. function call).
- ❖ Supported by the "VTABLE".

Polymorphism

```

graph TD
    Base[Base] --> Derived[Derived]
    
```

- ❖ Given Inheritance . . .
 - Derived object "is a" base object
- ❖ . . . and a common function . . .
 - Present in both base and derived class
- ❖ . . . which function to invoke?
 - If derived accessed as a base (upcast)

Two Possibilities

- ❖ Static Scoping with Upcasting
 - one object having two different behaviors to the SAME message
- ❖ Polymorphism using Virtual Fns
 - one object can only have one behavior for each message (consistency!)

MS-3832 – Adv. C++ Object-Oriented Programming

Upcast Example

Static Binding

```
class Sensor
{ public:
  void calibrate()
    { cout << "S::cal" << endl; }
};
class TempSensor : public Sensor
{ public:
  void calibrate()
    { cout << "TS::cal" << endl; }
};
```

7

Upcast Example

Same function in base & derived

```
class Sensor
{ public:
  void calibrate()
    { cout << "S::cal" << endl; }
};
class TempSensor : public Sensor
{ public:
  void calibrate()
    { cout << "TS::cal" << endl; }
};
```

8

Upcast Example

```
int main(void)
{
  TempSensor ts1;
  ts1.calibrate();

  Sensor& s1 = ts1;
  s1.calibrate();

  return (0);
}
```

9

Upcast Example

Declare derived object

```
int main(void)
{
  TempSensor ts1;
  ts1.calibrate();

  Sensor& s1 = ts1;
  s1.calibrate();

  return (0);
}
```

10

Upcast Example

Invoke function

```
int main(void)
{
  TempSensor ts1;
  ts1.calibrate();

  Sensor& s1 = ts1;
  s1.calibrate();

  return (0);
}
```

Result:
Calls derived class function

11

Upcast Example

Make reference of base type; call function again

```
int main(void)
{
  TempSensor ts1;
  ts1.calibrate();

  Sensor& s1 = ts1;
  s1.calibrate();

  return (0);
}
```

12

MS-3832 – Adv. C++ Object-Oriented Programming

Upcast Example

```
int main(void)
{
    TempSensor ts1;
    ts1.calibrate();

    Sensor& s1 = ts1;
    s1.calibrate();

    return (0);
}
```

Make reference of base type; call function again

Upcast occurs here!

13

Upcast Example

```
int main(void)
{
    TempSensor ts1;
    ts1.calibrate();

    Sensor& s1 = ts1;
    s1.calibrate();

    return (0);
}
```

Make reference of base type; call function again

Result:
Calls base class function!

14

Upcast Example

```
int main(void)
{
    TempSensor ts1;
    ts1.calibrate();

    Sensor& s1 = ts1;
    s1.calibrate();

    return (0);
}
```

This doesn't seem right!

After all, it is still the same object.

15

Early/Late Binding

- ❖ The problem - early binding
 - also called Static Binding
 - ◆ Function version associated with class
 - ◆ Function call bound at compile time
- ❖ The solution - late binding
 - also called Dynamic Binding
 - ◆ Choose function at run time
 - ◆ Actual object type: even if base pointer

16

Virtual Functions

- ❖ Function declared in base class
 - With virtual keyword
- ❖ Override function in derived cls.
 - No virtual needed (but OK)
- ❖ Function call
 - Invokes most-derived version

17

Virtual Functions

```
class Sensor
{ public:
    virtual void calibrate()
        { cout << "S::cal" << endl; }
};
class TempSensor : public Sensor
{ public:
    void calibrate()
        { cout << "TS::cal" << endl; }
};
```

18

MS-3832 – Adv. C++ Object-Oriented Programming

Virtual Functions

Virtual function in base class

```

class Sensor
{
public:
    virtual void calibrate()
    { cout << "S::cal" << endl; }
};
class TempSensor : public Sensor
{
public:
    void calibrate()
    { cout << "TS::cal" << endl; }
};
    
```

19

Virtual Functions

Overridden in derived class

```

class Sensor
{
public:
    virtual void calibrate()
    { cout << "S::cal" << endl; }
};
class TempSensor : public Sensor
{
public:
    void calibrate()
    { cout << "TS::cal" << endl; }
};
    
```

20

Virtual Example

```

int main(void)
{
    TempSensor ts1;
    ts1.calibrate();

    Sensor& s1 = ts1;
    s1.calibrate();

    return (0);
}
    
```

This time, both function calls invoke the derived version of the function.

Polymorphism!

21

How Does It Work?

- ❖ Each class specifies version
 - Explicitly: overrides base version
 - Implicitly: accepts base version
- ❖ List of function pointers
 - One for each class: VTABLE
- ❖ Pointer (vptr) in each object

22

Taxonomy

Another name for class hierarchy

```

graph TD
    TSensor --> Sensor
    PSensor --> Sensor
    
```

Sensor
+cal()

TSensor
+cal()
+res()

PSensor
+res()

23

VTABLE Contents

	Sensor	TSensor	PSensor
cal()	Yes	Yes	No
res()	No	Yes	Yes

Each column is a vtbl for a class.
Each row is a virtual function.

24

MS-3832 – Adv. C++ Object-Oriented Programming

VTABLE Contents

	Sensor	TSensor	PSensor
cal()	Yes	Yes	No
res()	No	Yes	Yes
cal()	S::cal()	T::cal()	S::cal()
res()	None	T::res()	P::res()

25

vptr Usage

TSensor TS1, TS2;
PSensor PS1;

cal()	S::cal()	T::cal()	S::cal()
res()	0	T::res()	P::res()

26

Why Virtual Fns?

- ❖ Virtual functions harmless?
 - If not overridden, base versions used
- ❖ Efficiency Detractors:
 - Pointer (vptr) takes up space
 - Extra table lookup: function address
 - Rules out in-line function expansion

27

Overriding Functions

- ❖ Invariant over specialization
 - Non-virtual function in base
 - Not in derived: don't override non-virtual
- ❖ Variant over specialization
 - Virtual function in base
 - Override in derived (or accept base version)

28

Pure Virtual Function

- ❖ Base function meaningless
 - Common behavior (function)
 - But implemented differently for all
- ❖ Declare interface only
 - Virtual syntax plus "= 0"
 - ◆ virtual void calibrate() = 0;

29

Abstract Base Classes

- ❖ Is there really a generic Sensor?
 - No, can't buy one!
 - Meant to describe common details
- ❖ Prevent instantiation:
 - Can't create object of class
 - At least one pure virtual function

30

MS-3832 – Adv. C++ Object-Oriented Programming

Object Slicing

```
int main(void)
{
    TempSensor ts1;
    ts1.calibrate();

    Sensor s2;
    s2 = ts1;
    s2.calibrate();
    ...
}
```

Assignment to base class copies only base part!

Result is base function call.
(Also pass by value.)

31

Object Slicing

- ❖ Moral of this story:
 - Never upcast by value
 - ◆ else object slicing occurs!
 - Upcast by pointer or by reference
 - If all functions in base class are virtual, object slicing generates a compile-time error message!

32

Virtual Constructors??

- ❖ Never!!
 - each ctor in hierarchy must be called in top-down order.
 - ctors shouldn't call virtual functions!
 - dtors shouldn't either!!
- ❖ But, virtual dtors do make sense!

33

Virtual Destructors

- ❖ Destructor clean-up for class
 - E.g., free heap memory
- ❖ What if base class pointer?
 - (can happen, due to upcasting)
 - Calls only base destructor, not derived
- ❖ Solution: `virtual` destructor!

34

Virtual Destructors

- ❖ All destructors are called
 - From most-derived to original base
- ❖ Declare if any virtual function
- ❖ Can be pure virtual (“= 0”)
 - Must still define destructor body!!
 - Sufficient for abstract base class

35

