Object-oriented programming: Polymorphism in C++

Topics:
- Virtual functions in C++
- Implementation of virtual and non-virtual functions

Motivation

We should be able to write functions that invoke an operation on an object, rather than invoking a particular method.

In C++, this is accomplished by declaring a function member to be virtual.

Example: declaring virtual function

class Employee {
private:
  string first_name, last_name;
  short department;
public:
  virtual void print() const;
};
class Manager : public Employee {
private:
  list<Employee*> group;
public:
  void print() const;

int main() {
  Employee doe("John", " Doe", 23);
  Manager howell("Charles", " Howell", 23, 3);
  doe.print(); // invokes Employee::print()
  howell.print(); // invokes Manager::print()
  Employee* ePtr = &howell;
  ePtr->print(); // invokes Manager::print()
}

Example: Virtual functions

int main() {
  Employee doe("John", " Doe", 23);
  Manager howell("Charles", " Howell", 23, 3);
  doe.print(); // invokes Employee::print()
  howell.print(); // invokes Manager::print()

  Employee* ePtr = &howell;
  ePtr->print(); // invokes Manager::print()
}

Virtual-function definitions

void Employee::print() const {
  os << " Doe: " << first_name << " " << last_name
       << endl << " Dept: " << department;
}
void Manager::print(os) {
  Employee::print(os);
  os << " Level: " << level;
}

Pop quiz

void myPrint( Employee& emp )
{
  emp.print(cout);
}
int main(void)
{
  Employee doe(" John", " Doe", 23);
  Manager howell(" Charles", " Powell", 23, 3);
  myPrint(howell);
}

Question: Which method is invoked when actual parameter is Manager?
Exercise

We want to store different geometric objects in a list and compute their area. Develop a class hierarchy with classes Shape, Circle, and Rectangle. It should be possible to declare:

```cpp
list<Shape> myShapes;
Circle* circ = new Circle(10);
Rectangle* rect = new Rectangle(20, 40);
myShapes.push_back(circ);
myShapes.push_back(rect);
for (list<Shape>::iterator it = myShapes.begin();
it != myShapes.end(); it++)
    cout << (*it)->area() << endl;
```

Heterogeneous containers

Inheritance and virtual functions useful for implementing containers that can hold different types of elements
- Need a base class from which all element types derive
- Operations need to be virtual, as we won’t know the actual type of the element when we access it from the container.

Question

What is the mechanism by which virtual functions are actually implemented?

Answer: Each object of a class with virtual functions carries with it a pointer to a "table of functions" (called a vtable).
- When a virtual function is invoked, the run-time system "finds the right function" by looking it up in the vtable.
- Because different objects can point to different vtables, this scheme allows a member function to be invoked based on the run-time type of the object.

Virtual-function implementation

```cpp
class C {
    protected:
        unsigned attr1;
        int attr2;
    public:
        void f1();
        virtual void f2();
        virtual void f3();
    };

class C1 : public C {
    protected:
        unsigned attr3;
    public:
        void f1();
        void f2();
        virtual void f3();
    };

class C2 : public C {
    public:
        void f1();
        void f2();
        void f3();
    };

class D : public C {
    protected:
        unsigned attr3;
    public:
        void f1();
        void f2();
        virtual void f3();
    };

class C {
    protected:
        unsigned attr1;
        int attr2;
    public:
        void f1();
        void f2();
        virtual void f3();
    };

class C1 : public C {
    protected:
        unsigned attr3;
    public:
        void f1();
        void f2();
        void f3();
    };

class C2 : public C {
    public:
        void f1();
        void f2();
        void f3();
    };

class D : public C {
    protected:
        unsigned attr3;
    public:
        void f1();
        void f2();
        virtual void f3();
    };

class C {
    protected:
        unsigned attr1;
        int attr2;
    public:
        void f1();
        void f2();
        virtual void f3();
    };

class C1 : public C {
    protected:
        unsigned attr3;
    public:
        void f1();
        void f2();
        void f3();
    };

class C2 : public C {
    public:
        void f1();
        void f2();
        void f3();
    };

class D : public C {
    protected:
        unsigned attr3;
    public:
        void f1();
        void f2();
        virtual void f3();
    };
```

Exercise

Observe: A data parameter is a value, not a reference.

```cpp
void myPrint(Employee e) {
    e.print();
}

int main(void) {
    Manager howell(...);
    myPrint(howell);
}
```
Question

A compiler translates a program in one language into an equivalent program in another language, which is "closer to the machine".

How would the compiler-generated code for function myPrint differ when Employee::print is virtual vs. non-virtual?

Code for myPrint (print NOT virtual)

```
pushl $ebp
movl $esp, $ebp
movl 8(%ebp), %eax
pushl $_cout
pushl $eax
call _print__Employee7stream
movl $ebp, $esp
popl $ebp
ret
```

Code for myPrint (print virtual)

```
pushl $ebp
movl $esp, $ebp
movl 8(%ebp), %edx
pushl $edx
movl 4(%edx), %ecx ; pt return in ecx
movl 12(%ecx), %eax ; pt address at offset
movl $eax, %eax ; call function pointed to
movl %ebp, %esp
popl %ebp
ret
```

Observations

Lots of things happen when we define an operation to be virtual
- Better left to the compiler
- No significant inefficiency (one extra indirection per virtual function invocation)

Terminology

C++ allows three different kinds of polymorphism:
- Ad hoc (using function overloading)
- Parametric (using template classes/functions); and
- Inclusion (using inheritance and virtual functions)

In OOP, the unqualified term polymorphism refers to inclusion polymorphism.

Class that defines virtual functions is called a polymorphic type