Software Design (C++)
5. OOP and class hierarchies

Juha Vihavainen
University of Helsinki

Preview

- Object-oriented programming and polymorphism
- OOP features in C++
  - inheritance, base classes, derived classes
  - virtual member functions
  - object layout and vtables
  - management of layered objects
  - run-time class info
- Technicalities: virtual destructors, private assignment, etc.
- Pros and cons of OOP
Object-oriented programming

- OOP thinking: "programming is simulation"
  - in fact, the first application area (of Simula 67, Norway) was discrete-event simulation
- a kind of "anthropomorphism" is central to the OO paradigm
  - thinking in terms of "real-world" objects (or independent agents) that interact with each other to get things done
- "strict" interpretation of the term:
  - uniform data abstraction - "everything" is an object
    - what is "everything" depends on the language
    - also: how strongly is data hiding enforced
  - need inheritance (- or perhaps prototypes + delegation)
  - need dynamic method binding (virtual operations in C++)


What is polymorphism?

- can write reusable code (frameworks) for classes that are not known in advance (not written or even yet designed)
- two (2) separate but related forms of polymorphism
  - inclusion polymorphism: the set of derived instances is (logically) a subset of the base-class instances
  - operation polymorphism = late binding of methods

Late (dynamic) binding

- existing code (the calling code) can change behavior (conform) to appropriately deal with new kinds of objects
- object's exact type need not be know at compile time for a call of a polymorphic (virtual) operation
- call is matched at run time to the exact type of target object

4.12.2014  Juha Vihavainen / University of Helsinki  4
Language mechanisms

- most popular definition of object-oriented programming:
  \[ \text{OOP} = \text{inheritance} + \text{polymorphism} + \text{encapsulation} \]
- base and derived classes \( \text{\# inheritance} \)
  - class Circle : public Shape { ... }; \( \text{\# Circle is-a Shape} \)
  - also called super and subclasses
- virtual functions \( \text{\# "polymorphism"} \)
  - virtual void drawLines () const;
  - also called "run-time polymorphism", "late binding", or "dynamic dispatch"
- private and protected members \( \text{\# encapsulation} \)
  - protected: Shape ();
  - private: std::vector<Point> points;

Calling base-class constructor

The constructor of a derived class calls the constructor of the base class \textbf{Base} using the \textit{member initializer} list

\textbf{Derived (. . ) : Base (ctor arguments), data(. . ), \# note order
\dots

\text{An initializer item for the base class
- uses normal constructor call syntax with appropriate arguments
- invoked (called) before the initialization of local data members
- calling the \textit{correct} constructor is the programmer’s responsibility
- otherwise, the base class’s \textit{default} (zero-arg.) \textit{constructor} is automatically inserted by the compiler and called (whether right or not!)}
Using virtual functions

Suppose a hierarchy of shape classes such as Circle, Text, etc.

- define a base class Shape with a virtual draw method
  
  ```
  virtual void draw() const;  // can override
  ```

- different shapes have their own unique draw operations so we need to override draw in derived classes
- call them by calling on the draw function on the base class Shape
- the target object is provided through a pointer or a reference
- the program determines dynamically (at run time) which function is actually executed

```
Shape * shape; . . . shape = new Circle; . . .
shape->draw();  // calls Circle::draw (possibly)
``` 

- objects need to carry along internal type info (vtable ptr)

What are "polymorphic" objects?

- data members defined for a class are implemented just like for structures (records)
- with (single) inheritance, derived classes have their fields at the end
- a pointer to the base and a pointer to the derived can contain the same address -- the derived pointer just "knows" that its fields go farther than those inherited from the base do (see the next slide)
- member functions are passed one special hidden first parameter: this (called self in Smalltalk and current in Eiffel..)
- non-virtual functions are regular subroutines; the compiler just calls the appropriate version, based on static type of variables/arguments
- C++ philosophy is to avoid run-time overhead whenever possible (holding to the legacy from C) - "don't pay for what you don't use"
- languages like Smalltalk and its "descendants" like Java require (much) more extensive run-time support
Inheritance creates *layered objects*

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

... Base * pBase = ...; Derived * pDerived = ...;
```

- polymorphic objects need to be accessed via pointers and are very often allocated from the heap (but not necessarily)

---

**Two kinds of OO polymorphism** (again)

*Inclusion polymorphism* (subclass polymorphism)

- every instance of a subclass is also an instance of the super classes
  ```
  Base * pBase = new Derived ();
  ```
- the Derived instance will have all Base properties/features so we can operate on it as a Base object

*Operation polymorphism* (late binding/dispatch)

- a (virtual) method is bound according to the run-time type of the target object (in "prewritten" existing reusable code)

```
pBase->foo (); => (* (pBase->vtable [ind])) (pBase)
```

The compiler generates a pointer to function static index this
Late binding = virtual member functions

- polymorphic objects are self-descriptive: carry along info about their type/class (class id) plus overridden behavior
- virtual functions are implemented by creating a dispatch table (vtable) for the class, and putting a pointer to that table in the data of the object (in a hidden field)
  - each derived class have a different dispatch table
- in the dispatch table, functions inherited from the parent (usually) come first, though some of these pointers may point to overridden versions (given in derived classes)
- putting the whole dispatch table into the object itself could save a little time, but waste a LOT of space

Implementation of virtual methods (1/2)

```c
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ... 
    virtual int l( ... 
    virtual void m();
    virtual double n( ... 
    ...
} F;
```

- representation of object $F$ begins with the address of the vtable for class foo
  - all instances (objects) of this class will point to the same vtable
  - the vtable itself consists of an array of addresses, one for each virtual method of the class (note that all this can be implemented in C)
  - the remainder of $F$ consists of the representations of its fields

Of course, the actual layout depends on the implementation.
Implementation of virtual methods (2/2)

class bar : public foo {
    int w;
public:
    void m(); //override
    virtual double s( ... 
    virtual char *t( ... 
    ... 
} B;

- here, too, object B begins with the address of its class’s vtable
- the first four entries represent the members for foo, except that one (m) has been overridden and contains the address of a different subroutine
- additional virtual methods (s, t) follow the ones inherited from foo
- inside B, additional fields of bar (w) follow the ones inherited from foo

Run-time std::type_info in C++

- to query the type of an object, we need to be able to get from the object to its run-time type info
  - the vtable structure already is a kind of meta-data
  - a common implementation technique is to put a pointer to the type info at the beginning of the vtable
  - the type of an object can be queried by the typeid operator
    
    \[
    \text{std::cout} \ll \text{typeid(\*pBase).name()} \ll \text{"\n"}; \quad \text{// print type}
    \]

- the compiler generates a vtable in only if your class has at least one virtual function
- that’s why you can’t do a dynamic_cast<T>() on a pointer whose static type (class) doesn’t have virtual functions
- in most other object-oriented languages, objects (usually) always carry along their type info (plus extensive other data related to the class)
Using late binding: virtual destructors

- a base class has a destructor `Shape::~Shape()` that is implicitly
called by a delete

```cpp
Shape * shapePtr; . . .
delete shapePtr; // refers to some specific shape (say, Circle)
```

- if a derived object is deleted through a base-class pointer (as often
  happens), the C++'s default static binding strategy will cause:
  - the base-class destructor is called and it acts on the object
  - but the potential derived-class resources remain unreleased (!)

- a base-class destructor must be declared as virtual to ensure that the
  right (= most-specific) destructor will always be called
  - that destructor will then call on the base destructors, and the
    destruction proceeds from the subclass to its base classes

Idiom: virtual destructor

- when you design a class to be possibly used as a base class, always
  make its destructor virtual
  - specifically: if a derived instance is deleted via a base pointer

- if the base class has no resources to release, make the destructor's
  body empty (but you must still implement it)
  - the derived-class destructor calls it anyway (!)
  - but the compiler can often optimize away unnecessary calls of
    empty (inline) blocks

- a polymorphic object (operation polym.) must carry along extra
  type info, namely a pointer to its class' vtable (as discussed earlier)
  => overheads

- note that in "plain" data types, the destructors are often not defined
  virtual, e.g. `std::string, std::vector` do not have virtual destructors
Pure virtual functions

Often, a function in a base class can’t be implemented at all

- e.g. the data/state is not available (but “hidden” in derived classes)
- must ensure that a derived concrete class implements that function
- solution: make it a “pure virtual function” (= 0)

How to define truly “abstract interfaces” in C++

class Engine {
  // interface to electric motors
  // no data, and (usually) no constructors (but empty default ones)
  virtual void speedUp (int i) = 0; // defined in a derived class
  // ...
  virtual ~Engine () {} // (empty) virtual destructor
};

Engine eee;

// error: Engine is an abstract class

Pure virtual functions (cont.)

A pure interface can then be used as a base class

class M123 : public Engine {
  // engine model M123
  public:
    M123 (); // initialize, and get resources
    void speedUp (int i) ; ... // overrides Engine::speedUp
    ~ M123 (); // release resources
    ... // representation
};
...
M123 m123; // now OK
Technicality: preventing copying
If you don’t know how to copy an object, prevent such copying!
E.g., classes in a class hierarchy (often) cannot be assigned (say, Person)

```cpp
class X { // ...
  private:
    X (const Shape&); // cannot copy construct
    X& operator = (const X&); // cannot copy assign
};

void f (X& a) {
  X s2 = a; // error: X (const Shape&) = delete;
  a = s2; // error: X& operator = (const X&);
}
```

- should also leave undefined => cannot "accidentally" call at all
- sometimes provide copy construction (for new), but not assignment

Pros and cons of OOP
- useful way to create natural and intuitive conceptual hierarchies
  - consider GUI classes: Shape, Circle, Triangle, Text, etc.
- supports code reuse, extensibility, and "flexibility" at run time
- can cause some overhead (usually don't need to worry..)
  - indirection in calls (via vtable) plus the general call overhead
  - pointer downcasts and type recovery (lots in "pre-generic Java")
- in many cases, templates provide often as flexible but more secure
  - way to achieve (a static version of) polymorphism
  - e.g., STL gives type-parameterized containers and algorithms
- inheritance & late binding are avoided in STL; compile-time
  - checking supports early and more extensive error detection
- can often improve performance (static binding, inlining, etc.)

**Finally:** all can be combined: generics & inheritance & late binding..