Objects in C++

Objects, with dynamic lookup of virtual functions

Patrizia Scandurra
scandurra@dti.unimi.it
Info3

C++ Object System

- Object-oriented features
  1. Classes and Data Abstraction
  2. Encapsulation
  3. Inheritance
     - Single and multiple inheritance
     - Public and private base classes
  4. Objects, with dynamic lookup of virtual functions
  5. Subtyping
     - Tied to inheritance mechanism

Polymorphism in C++

- runtime polymorphism (virtual functions)
- compile-time polymorphism (templates)

Run-time Polymorphism

- **Run-time polymorphism**: implemented with **dynamic lookup of virtual functions**
- ***Dynamic lookup***: a method is selected dynamically, at run time, according to the implementation of the object that receives a message
  - not some static property of the pointer or variable used to name the object
- The important property of dynamic lookup is that **different objects may implement the same operation differently**

Virtual functions

- Member functions are either
  - Virtual, if explicitly declared or inherited as virtual
  - Non-virtual otherwise
- Virtual members
  - Are accessed by indirection through ptr in object
  - May be *overridden* in derived (sub) classes
- Non-virtual functions
  - Are called in the usual way. *Just ordinary functions*.
  - May be redefined in derived classes (overloading through *redefining*)
- Pay overhead only if you use virtual functions

Sample class: one-dimen. points

```
class Pt {
  public:
    Pt(int xv);
    Pt(Pt* pv);
    int getX();
    virtual void move(int dx);
  protected:
    void setX(int xv);
  private:
    int x;
  };
```

Overloaded constructor

Public read access to private data

Virtual function

Protected write access

Private member data

Sample derived class

```
class ColorPt: public Pt {
  public:
    ColorPt(int xv,int cv);
    ColorPt(Pt* pv,int cv);
    ColorPt(ColorPt* cp);
    int getColor();
    virtual void move(int dx);
    virtual void darken(int tint);
  protected:
    void setColor(int cv);
  private:
    int color;
 };
```

Overloaded constructor

Non-virtual function

Virtual functions

Protected write access

Private member data

Sample derived class

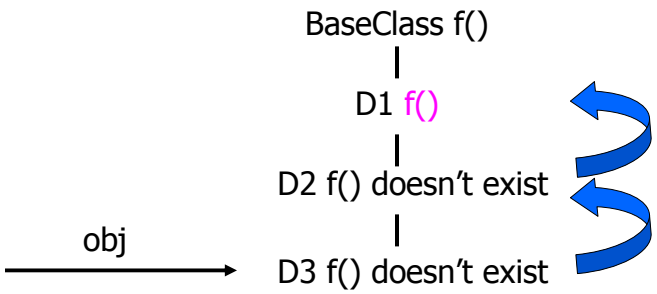
```
/* ---------- Definitions of Member Functions --------*/

void ColorPt::darken(int tint) { color += tint; }

void ColorPt::move(int dx) {
Pt::move(dx); this->darken(1);
}
```

Virtual functions and *indirection (1)*

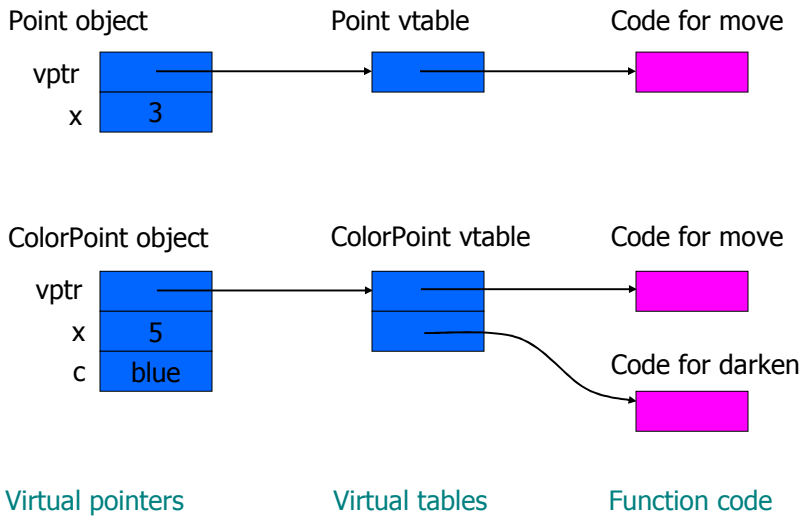
- C++ allows a base class pointer to point to a derived class object
- Upon method invocation, the method of the derived object is called
- This leads to generic alghoritms using base class pointers

```
Pt*  ptr = new ColorPt;
ptr->move();
delete(ptr);
```

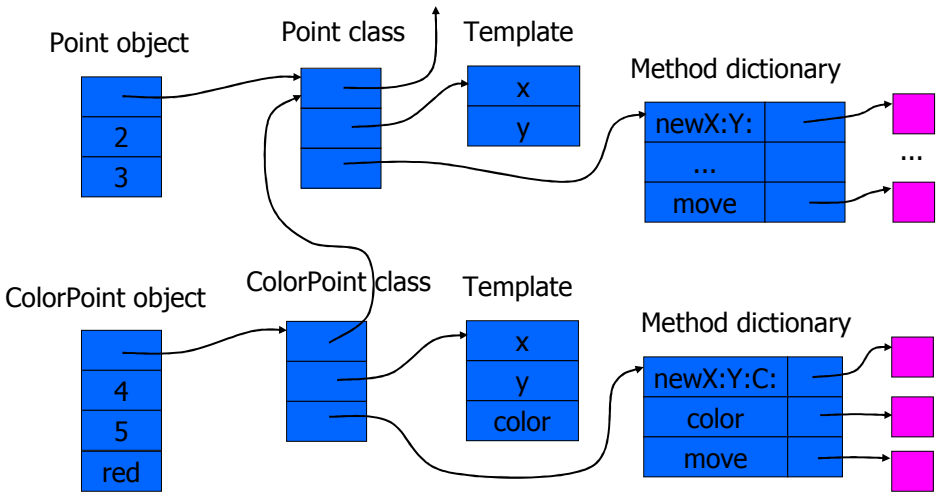
Virtual functions and *indirection (2)*



Run-time representation



Compare to Smalltalk



Why is C++ lookup simpler?

- Smalltalk has no static type system
  - Code p message:pars could refer to any object
  - Need to find method using pointer from object
  - Different classes will put methods at different place in method dictionary
- C++ type gives compiler some superclass
  - Offset of data, fctn ptr same in subclass and superclass
  - Offset of data and function ptr known at compile time
  - Code p->move(x) compiles to equivalent of (*(p->vptr[1]))(p,x) if move is first fctn in vtable.

data passed to member function; see next slide

Calls to virtual functions

- One member function may call another

```
class A {
    public:
        virtual  int  f (int x);
        virtual  int  g(int y);
};
int A::f(int x) { … g(i) …;}
int A::g(int y) { … f(j) …;}
```

- How does body of f call the right g?
  - If g is redefined in derived class B, then inherited f must call B::g

"This" pointer

- Code is compiled so that member function takes "object itself" as first argument

  Code `int A::f(int x) { … g(i) …;}`

  compiled as `int A::f(A *this, int x) { … this->g(i) …;}`

- "this" pointer may be used in member function
  - Can be used to return pointer to object itself, pass pointer to object itself to another function, ...

Non-virtual functions

- How is code for non-virtual function found?
- Same way as ordinary "non-member" functions:
  - Compiler generates function code and assigns address
  - Address of code is placed in **symbol table**
  - At call site, address is taken from symbol table and placed in compiled code
  - *But* some special scoping rules for classes
- Overloading
  - Remember: overloading is resolved at compile time
  - This is different from run-time lookup of virtual function

Scope rules in C++

- Scope qualifiers
  - binary :: operator, ->, and .
  - class::member, ptr->member, object.member
- A name outside a function or class,
  - not prefixed by unary :: and not qualified refers to global object, function, enumerator or type.
- A name after X::, ptr-> or obj.
  - where we assume ptr is pointer to class X and obj is an object of class X
  - refers to a member of class X or a base class of X

Virtual vs Overloaded Functions

```
class parent { public:
    void printclass() {printf("p ");};
    virtual void printvirtual() {printf("p ");};   };
class child : public parent { public:
    void printclass() {printf("c ");};
    virtual void printvirtual() {printf("c ");};    };
main() {
    parent p;  child c; parent *q;
    p.printclass(); p.printvirtual(); c.printclass(); c.printvirtual();
    q = &p;  q->printclass(); q->printvirtual();
    q = &c;  q->printclass(); q->printvirtual();
}
```

Output: p p c c p p **p** c

Function call binding

- early binding (C, C++)
  - at compile time
- late binding (C++)
  - at runtime
  - mighty, but a bit less efficient
    - 1 more assembler statement per call,
    - slight memory overhead due to VPTRs