Objects in C++

Objects, with dynamic lookup of virtual functions

C++ Object System

- Object-oriented features

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

Polymorphism in C++

- Runtime polymorphism
- Virtual functions
- Compile-time polymorfism
- (parametric polymorfism)
- Generic programming
- 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
- 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

Virtual members

- Must be explicitly declared as "virtual"
- May be *overridden* in derived (sub) classes
- Dynamic binding is activated
- Are accessed by indirection through **ptr** in object
- Explicitly as pointers or using references

```
class A { public: virtual void vi(){...}};
class B : public A{ public: virtual void vi(){ …}};
int main() {
     A* pa = new A; a -> vi(); // VIRTUAL CALL
     A& ra = b; ra.vi(); // VIRTUAL CALL
     A a = b; a.vi(); // NON VIRTUAL CALL
}
```

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);
}
```

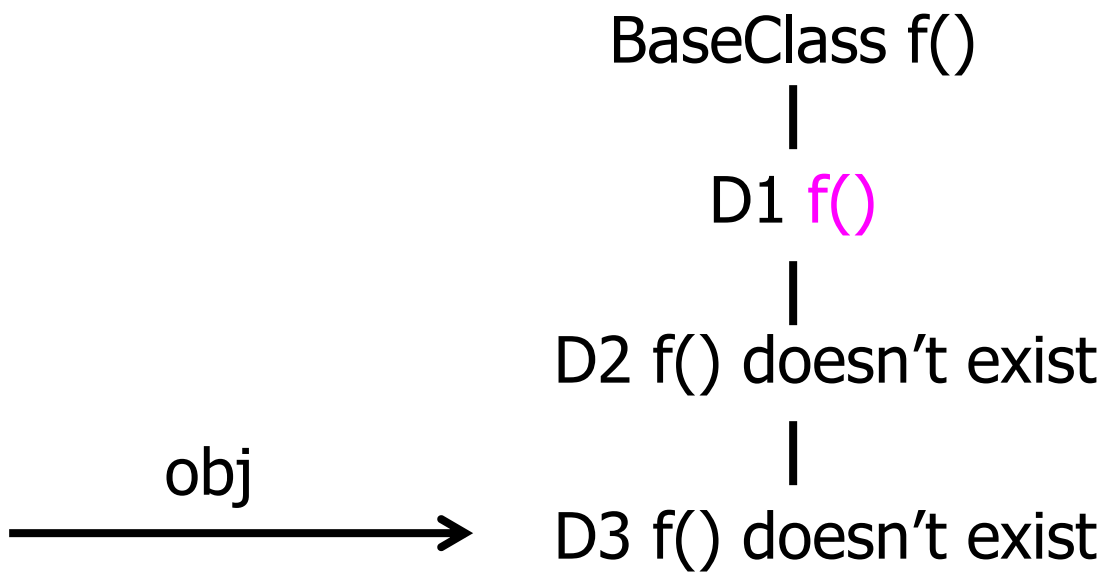
Virtual functions and *indirection (1)*

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

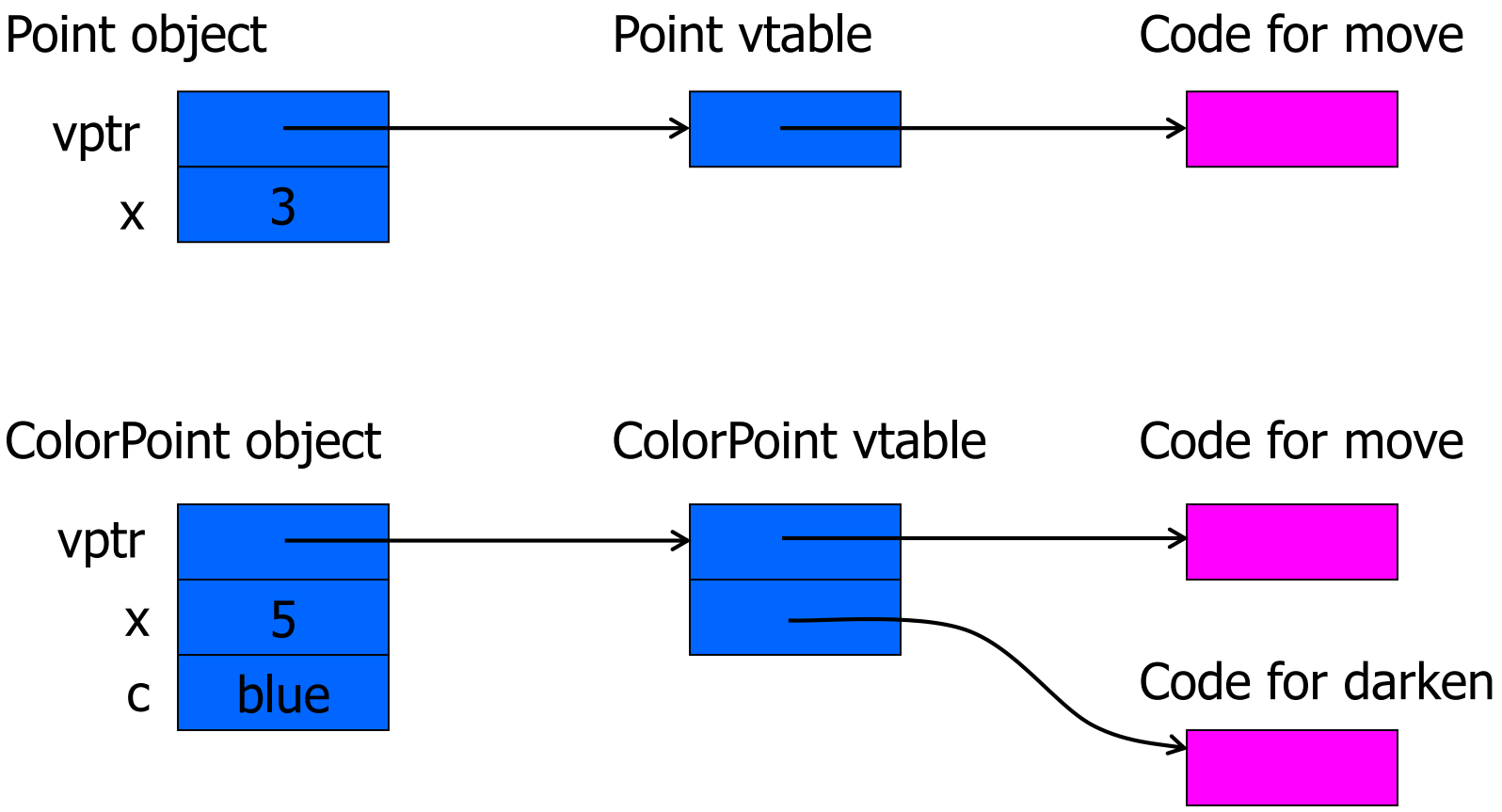
```
Pt*  ptr = new ColorPt;

ptr->move();
```

Virtual functions and *indirection (2)*



Run-time representation



“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, ...

Constructors/destructors and inheritance (2)

- destructors
  - always make destructors virtual in base classes
  - there might be cleanup work to be done in derived classes

```
class Employee {
  //...
  public:
   //...
   virtual ~Employee() {}
};
```

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

Overload

- An **overloaded** function is a function that shares its name with one or more other functions, but which has a different parameter list. The compiler chooses which function is desired based upon the arguments used.

Overridden

- An **overridden** function is a method in a descendant class that has a different definition than a **virtual** function in an ancestor class. The compiler chooses which function is desired based upon the type of the object being used to call the function.
  - Regardless the access modifier (private and so on) of the function
  - Si può fare overriding anche di metodi private
    - Not like Java
    - Vediamo un esempio

redefined

- A **redefined** function is a method in a descendant class that has a different definition than a non-virtual function in an ancestor class. Don't do this. Since the method is not virtual, the compiler chooses which function to call based upon the static type of the object reference rather than the actual type of the object.

Virtual vs redefined 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

Esercizio

- Definiamo una classe A con un metodo virtual che ridefiniamo (overriding) in una sottoclasse B.
- Proviamo a chiamare quel metodo in diversi casi

Function call binding

- Early binding (C,C++)
  - At compile time
- Late binding (C++)
  - At runtime
- Mighty. But less efficient
- 1 more assembler statement per call
- Slight memory consumption due to the VPTRs