Class inheritance

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Topics

- Class inheritance - fundamental notions:
  - base class and sub-class
  - inherited functionality from base class to derived class
  - `protected` access modifier
- Member function redefinition
  - redefinition and polymorphism
  - use of `virtual`, `override` and `final`
  - pure `virtual` functions puras and abstract classes
- Consolidating example
  - a class hierarchy for 2D geometric shapes

Code examples are available online at GitHub.

Declaring subclasses

Inheritance is normally declared as follows:

```
class SubClass : public BaseClass {
    ...
};
```

`SubClass` is called a **subclass** of the `BaseClass` **base class**. Other common designations: the base class is also called the **parent class** or **super-class**, the subclass is also called the **derived class** or **child class**.

`SubClass` inherits the functionality of `BaseClass`, and can define additional functionality on its own.

More advanced: `SubClass` may redefine the inherited functionality, and in some cases `BaseClass` may just define abstract functionality to be implemented by `SubClass`.

A first example

```
class person {
public:
  person(int id, const string& name);
  person(const person& p);
  int id() const;
  const string& name() const;
  ...
private:
  int pid;
  string pname;
};
```

`person` will be used as a base class.

A first example (cont.)

```
class teacher : public person {
public:
  teacher(int id, const string& name,
          const string& dept);
  teacher(const teacher& t);
  const string& department() const;
  ...
private:
  string tdepartment;
};
```

`teacher` is a subclass of `person`. It inherits all the definitions of `person` but also defines new functionality, for instance the `department()` member function.

Class hierarchies

A class may have an arbitrary number of classes, and a class hierarchy can have several levels. We could have for instance:

```
class person { ... };

// Subclasses of person
class teacher : public person { ... };
class student : public person { ... };

// Subclasses of student
class erasmus_student : public student { ... };
class working_student : public student { ... };
```

We will see a concrete example later.

Inherited functionality

Over a `teacher` object we can invoke the `department()` member function. Since `teacher` is a subclass of `person` we can also invoke `id()` and `name()` member functions inherited from `person`.

```
person p(123, "Joana Doa");
std::cout << p.id() << ' '
          << p.name() << '\n';

teacher t(124, "John Doe", "Computer Science");
std::cout << t.id() << ' '
          << t.name() << ' '
          << t.department()
          << '\n';
```

Output:

```
123 Joana Doa
124 John Doe Computer Science
```

Inherited functionality (cont.)

```
int person::id() const { return pid; }
const std::string&
person::name() const { return pname; }
...
const std::string&
teacher::department() const { return tdepartment; }
```

`id()` and `name()` are only defined for `person`. The functionality is thus reused for `teacher`. The `teacher` subclass only needs to implement the new functionality it declares on its own such as `department()`.

Class inheritance vs. class composition

Inheritance should reflect an "**is a**" relationship between the abstract notions represented by classes, e.g., a teacher is a person.

A **common design** error is to use class inheritance to model a "**has a**" relationship, for which class composition is more appropriate. For example we could have the `polynomial` class of previous classes formulated as:

```
//  BAD design
class polynomial : public std::vector<fraction> { ... };
```

instead of using a `vector<fraction>` field for the coefficients. A polynomial *is not* a vector of fractions, but it *has a* (it internally employs) vector of fractions.

Subclass constructors

```
person::person
(int id, const string& name)
  : pid(id), pname(name) {

}
...
teacher::teacher(int id,
 const string& name,
 const string& dept) :
   person(id, name), // base class constructor call
   tdepartment(department) {

}
```

A subclass constructor must call the base class constructor. This happens typically through a member initializer list that begins with a call to a constructor in the base class. If such a call is absent, a call to the default constructor of the base class is assumed.

Subclass constructors (const.)

```
person::person(const person &p) :
  pid(p.pid), pname(p.pname) { }
...
teacher::teacher(const teacher &t)
  : person(t), tdepartment(t.tdepartment) { }
```

Another subclass constructor example: the `teacher` copy constructor calls the `person` copy constructor.

`protected` access modifier

```
class person {
protected:
  // directly accessible in subclasses
  int pid;
  std::string pname;
...
};
class teacher : public person { ... };
```

Like `public` and `private`, `protected` is an access modifier. **A `protected` declaration is accessible inside the declaring class but also any of its subclasses**. Code outside the class or subclasses cannot access `protected` declarations (like in the case of `private`)

In the above variant of `person`, code in the `teacher` subclass may directly access member fields `pid` and `pname` in `person`.

`protected` access modifier (cont.)

If we have ...

```
class person {
protected:
  // directly accessible in subclasses
  int pid;
  std::string pname;
...
};
```

then access to `pid` and `pname` would be valid in code such as the following:

```
teacher::teacher(const teacher &t)
  : person(t), tdepartment(t.tdepartment) {
    pid = pid + 1000; // direct access
    pname = "Prof. " + pname;
  }
```

Inheritance and access modifiers

Usually inheritance is declared with a `public` access modifier, as in our example:

```
class teacher : public person ...
```

We will always use `public` in the course. But technically, it is also possible to have ...

```
class teacher : protected person ...
```

or ...

```
class teacher : private person ...
```

- `public`: preserves the accessibility of inherited definitions;
- `protected`: `public` declarations in the base class are changed to `protected`;
- `private`: `public` and `protected` declarations in the base class are changed to `private` in the subclass.

Inheritance and access modifiers (cont.)

For example, suppose we have:

```
class teacher : protected person { ... };
```

Then the following code is not valid, if defined outside class `teacher`:

```
teacher t(123, "John Doe", "Computer Science");
cout << t.id(); // NOT VALID!
```

```
‘int person::id() const’ is inaccessible within this context
```

`id()` has `protected` access for `teacher` objects, even if it remains `public` for `person`-only objects

```
person p(123,"John Doe");
cout << p.id(); // VALID!
```

Pointers and references

```
teacher t(...);
person& r = t;
person* p = &t;
```

Since `teacher` is a subclass of `person`, we can have `person` references or pointers to `teacher` objects, since the `teacher` type is a subtype of the `person` type.

The ability to use base class pointers or references to refer to objects of subclasses can be quite useful, as we will see.

Casting

On the other hand, a `person` reference or pointer does not necessarily refer to a `teacher` object. Pointer or reference assignment requires a **cast**, for example as in

```
person* p = ...;
teacher* t = (person*) p; // cast
```

The above cast is **safe** if `p` really points to a `teacher` object. Otherwise the behavior is undefined ... bad things tend to happen ...

Excluding pointers or references, a cast cannot be used for a direct conversion from base class to subclass (we get a compilation error):

```
person p (...);
teacher t = (teacher) p;
```

```
no matching conversion for C-style cast from
'person' to 'teacher'
```

Object slicing

What happens when we use a derived class object to construct or assign a base class object? For example, as in:

```
teacher t(124, "john doe", "Computer Science");
person p = t;
```

Only the base class "part" of the subclass object is considered. This is known as **object slicing**.

In the example, a "slice" is taken only for the `person` part of `t`: `t` is treated just as a `person` object for the construction of `p`.

Redefinition of member functions

Both `person` and `teacher` declare the `print()` member function. The member function declaration is the same in both classes:

```
class person {
   ...
   void print(std::ostream& out) const;
};
class teacher : public person {
   ...
   void print(std::ostream& out) const;
};
```

`teacher::print()` is in this case called a **redefinition** of `person::print()`.

Redefinition of member functions (cont.)

`teacher::print()` may internally call `person::print()` ….

```
void person::print(std::ostream &out) const {
    // Print id and name
    out << "ID: " << pid << std::endl
        << "Name: " << pname << std::endl;
}
void teacher::print(std::ostream &out) const {
    // Call person::print
    person::print(out);
    // Print department info
    out << "Department: " << tdepartment << std::endl;
}
```

Redefinition of member functions (cont.)

In line with the previous code of `teacher::print` then

```
teacher t(124, "John Doe", "Computer Science");
t.print(std::cout);
```

will print

```
ID: 124
Name: John Doe
Department: Computer Science
```

Redefinition of member functions (cont.)

Since we can have base class pointers or references what will we get in the following case?

```
teacher t(124, "John Doe", "Computer Science");
person* pt = &t;
pt->print(std::cout);
```

`pt` points to an object of type `teacher` but is declared as a pointer to `person`. Which member function is called: `person::print` or `teacher::print`?

Redefinition of member functions (cont.)

```
teacher t(124, "John Doe", "Computer Science");
person* pt = &t;
pt->print(std::cout);
```

The above fragment invokes `person::print()`

```
ID: 124
Name: John Doe
```

because the declared type of `pt` refers to `person`, not `teacher`.

We can however get a different behavior if `print()` is declared to be a **virtual function** in `person`.

Virtual functions

We can declare `print()` as a **virtual function** in `person` using the `virtual` keyword:

```
class person {
    ...
    virtual void print(std::ostream& out) const;
};
class teacher : public person {
    ...
    void print(std::ostream& out) const;
};
```

In this case, `teacher::print` is called a **virtual function override**.

But what changes in practice?

Use of `virtual`

If `person::print` is declared as `virtual`, then a call to `print()` through a `person` reference or pointer will use **dynamic binding**, i.e., take into account the actual **type of the object at runtime** rather than the declared type of the variable ...

Hence

```
teacher t(124, "John Doe", "Computer Science");
person* pt = &t;
pt->print(std::cout);
```

will call `teacher::print` ...

```
ID: 124
Name: John Doe
Department: Computer Science
```

Use of the `override` modifier

Good practice: although optional, it is recommended that a virtual function override is annotated with the `override` keyword.

```
class person {
    ...
    virtual void print(std::ostream& out) const;
};
class teacher : public person {
    ...
    void print(std::ostream& out) const override; // use of ove
};
```

The `override` annotation explicitly signals that a function is a virtual function override. The compiler validates that the override is valid.

Use of the `override` modifier (const.)

Example of invalid override (different return type):

```
class person {
    ...
    virtual void print(std::ostream& out) const;
};
class teacher : public person {
    ...
    // int instead of void for the return type!
    // The function is not a valid override
    // of person::print().
    int print(std::ostream& out) const override;
};
```

The compiler infers that the override is invalid:

```
error: virtual function 'print' has a different return type
('int') than the function it overrides (which has return type
'void')
```

Use of `final`

A virtual function may be declared as `final` to disallow overrides.

For `person::print` declared final

```
class person {
  ...
  virtual void print(std::ostream& out) const final;
};
class teacher : public person {
  ...
  // Not allowed!
  void print(std::ostream& out) const override;
};
```

we get a compilation error for the `teacher::print` override

```
error: declaration of 'print' overrides a 'final' function
```

Use of `final` (cont.)

`final` can also be declared for a class to disallow any subclasses ...

```
class person final { ... };
// Subclasses not allowed
class teacher : public person { ... };
```

```
error: base 'person' is marked 'final'
```

Pure virtual functions and abstract classes

A **pure virtual function**, sometimes also called an abstract function, is declared as `virtual ... func(...) = 0`, e.g.,

```
class person {
  ...
  virtual void print(std::ostream& out) const = 0;
};
```

A class with one or more pure virtual functions is called **abstract**. An abstract class cannot be directly instantiated. **An override of the function must be defined by (concrete) sub-classes**. The function **typically also has no implementation at the base class level**.

Pure virtual functions and abstract classes (cont.)

For

```
class person {
  ...
  virtual void print(std::ostream& out) const = 0;
};
```

we cannot have variables of type `person` ...

```
person p(123, "Joana Doa");
```

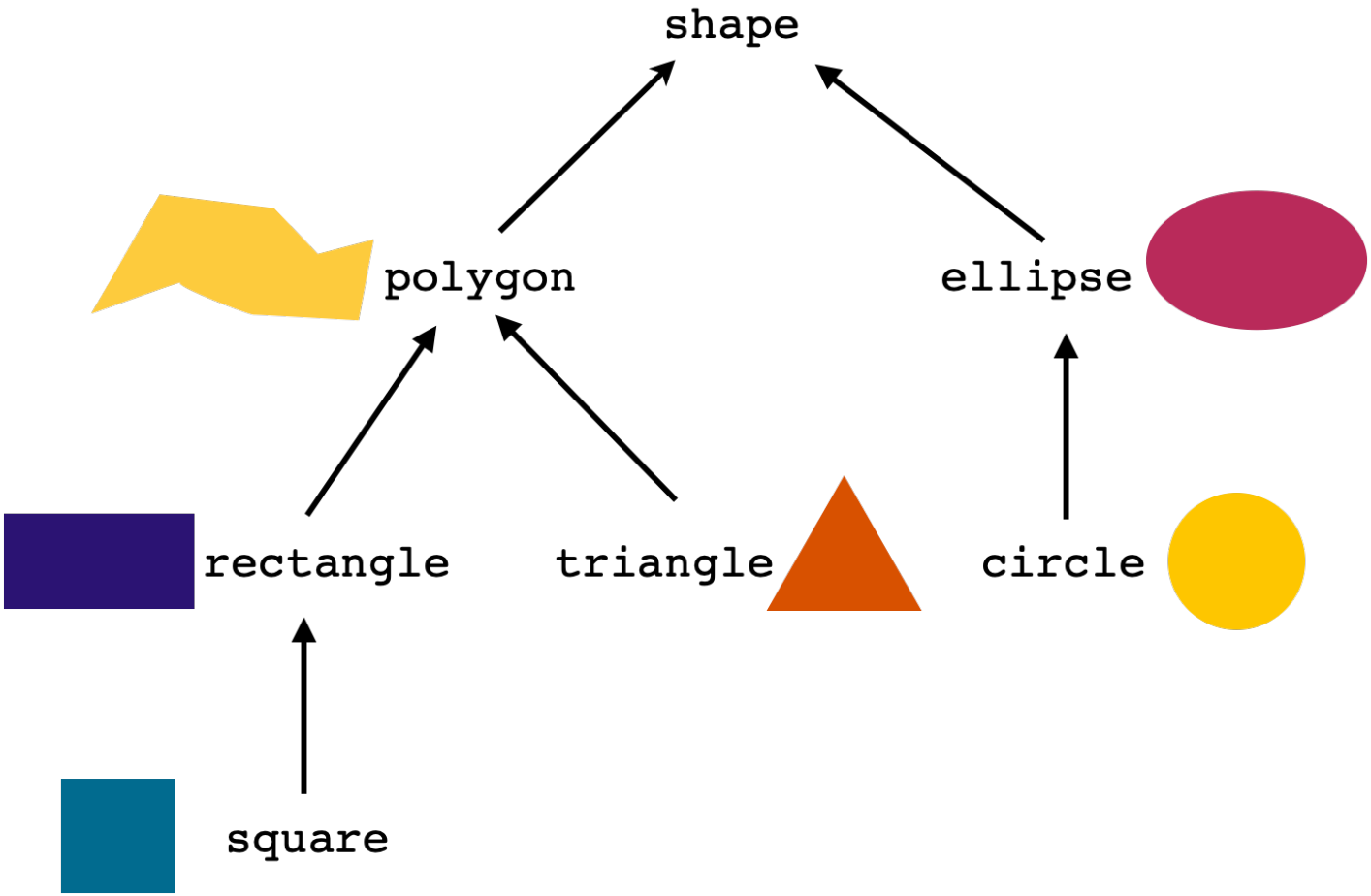
```
error: variable type 'person' is an abstract class
```

However, as `teacher` extends `person`, we can still have

```
teacher t(124, "John Doe", "Computer Science");
```

Consolidating example

A hierarchy of classes for 2D geometrical shapes:



shape abstract class

```
class shape {
public:
    virtual double area() const = 0;
    virtual coord2d center() const = 0;
    virtual void move(const coord2d& direction) = 0;
    virtual ~shape();
};
```

Since shape is abstract (it declares pure virtual functions), we cannot instantiate it directly.

Virtual destructor

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

Good practice: base class defines virtual destrutor so that subclass destructors are properly invoked when using base class pointers, e.g.

```
shape* s =  new polygon ( ...);
...
// guarantees that destructor
// of polygon is called
delete s;
```

Example subclass of shape

```
class ellipse : public shape {
private:
    coord2d ecenter;
    double erx;
    double ery;
public:
    ellipse(const coord2d& c, double rx, double ry)  :
            ecenter(c), erx(rx), ery(ry) { }
    double radius_x() const { return erx; }
    double radius_y() const { return ery; }
    double area() const override final  {
      { return  M_PI * erx * ery; }
    coord2d center() const override final { return ecenter; }
    void move(const coord2d& movement) override final
      {  ecenter += movement; }
};
```

Example subclass of `shape` (cont.)

```
class ellipse : public shape {
...
public:
    ...
    double area() const override final  {
      { return  M_PI * erx * ery; }
    coord2d center() const override final
      { return ecenter; }
    void move(const coord2d& movement) override final
      {  ecenter += movement; }
};
```

`ellipse` defines overrides fort the pure virtual functions declared in `shape`: `area()`, `center()`, and `move()`. In turn, these functions are marked `final` disallowing their override in subclasses of `ellipse` (`circle` discussed next).

One more level in the hierarchy ...

```
class circle final : public ellipse {
public:
    circle(const coord2d& c, double r)
      : ellipse(c, r, r) { }
};
```

`circle` is a subclass of `ellipse`. It cannot override any member functions of `ellipse`, as all of them are marked `final`.

On the other hand, `circle` cannot have subclasses as it is is marked `final`.

The other classes in the `shape` hierarchy ....

(see code online)

```
class polygon : public shape { ... }
class triangle final : public polygon { ... };
class rectangle : public polygon { ... };
class square final : public rectangle { ... };
```

The drawing class

A drawing object can group several shapes:

```
class drawing final {
private:
    std::vector<shape*> shapes;
public:
    drawing() { }
    ~drawing() { ... }
    void add_shape(shape* s) { ... }
    void move_all(const coord2d& movement) { ... }
    std::vector<shape*>& get_shapes() { ... }
};
```

The drawing class (cont.)

Example use of drawing:

```
drawing d;
d.add_shape(new rectangle({3, 5}, 2, 5));
d.add_shape(new circle({1, 1}, 1));
d.add_shape(new triangle({1, 1}, {1, 2}, {2, 1}));

d.move_all({3,-1}); // move all shapes

// Iterate shapes in the drawing
for (shape *s : d.get_shapes())
    std::cout << s->center() << ' '
              << s->area() << std::endl;
}
```

The drawing class (cont.)

```
class drawing final {
private:
    std::vector<shape*> shapes;
public:
    ...
    void add_shape(shape* s) {
        shapes.push_back(s);
    }
    ...
};
```

To store the component shapes, `drawing` uses a vector of `shape*` pointers. We can not have the `vector<shape>` since `shape` is abstract. Even if `shape` was not abstract, we could not use `vector<shape>` to store objects that are subclasses of `shape` (they would be "sliced").

The drawing class (cont.)

```
class drawing final {
private:
    std::vector<shape*> shapes;
public:
    ...
    void move_all(const coord2d& movement) {
        for (shape* s : shapes) {
            s -> move(movement);
        }
    }
    ...
};
```

The abstract functionality of **shape** lets us manipulate all (possible types of) shapes using the same interface.

The drawing class (cont.)

drawing assumes that each contained shapes is dynamically allocated using new. Accordingly, the memory of each shape is released by the class destructor.

```
class drawing final {
private:
    std::vector<shape*> shapes;
public:
    ...
    ~drawing() {
        for (shape* s : shapes) {
            delete s;
        }
    }
    ...
};
```