“Atelier C++” — Day 2 out of 5

Thierry Géraud

EPITA Research and Development Laboratory (LRDE)

2007

Outline

Outline

Outline

1. Rationale for inheritance
   - Modularity strikes back
   - C shape
2. Inheritance in C++
   - Abstract class and abstract method
   - Definitions + playing with words
   - Subclassing
3. Playing with types
   - Transtyping
   - Accessibility
   - Conclusion

Outline

After day 1

We have

- a `circle` class
- nice features
  - encapsulation
  - information hiding
  - class / object
- a toy-like piece of software

We want rectangles!

After day 1

We want to **extend** our program (to add some new feature).

We would like to ensure that

- extending does not lead to *modify* code
  → adding = a **non-intrusive** process :)
- we do not break the “type-safe” property
  → a new type is not really an unknown type! :)

Program functionalities

Expected functionalities are the following:

- both circles and rectangles can be translated
- both circles and rectangles can be printed

So we want to handle *shapes*.

Shapes?

We can say:

- that a shape is *either* a circle *or* a rectangle
- that both circles and rectangles *are* shapes
- so every shapes can be processed

Please remind that!

Once again:

- a circle *is* a shape
- a rectangle *is* a shape
- if you hold/know/have a shape, it is *either* a circle *or* a rectangle
- actually a set of circles and rectangles is a set of shapes
- OK?

Conclusion

There is a shape **module** in our program:

- sub-modules are *particular* kinds of shapes
- this module can be extended with new sub-modules (what about triangles?)
- extension should be non-intrusive

There is a **type** (“*shape*”) to represent shapes:

- our context is a language with some kind of typing
- “good” typing leads to “good” programs
- compiler is our best friend

  but we have to help it understand what we want

Outline

A first big problem

Think about the couple of sentences:

a *shape* is either a *circle* or a *rectangle*
and
an entity has exactly *one* type

In C that sounds like:

- we should use three types
- we have to resort to the C “cast” feature...

Shape type

First we need shapes, so:

```
typedef enum { circle_id = 0, rectangle_id = 1 } shape_id;

typedef struct {
  shape_id id;
  float x, y;
} shape;
```

Circle and rectangle types

With:

```
typedef struct {
  shape_id id; // == circle_id
  float x, y;
  float r;     // radius
} circle;
```

```
typedef struct {
  shape_id id; // == rectangle_id
  float x, y;
  float w, h;  // width and height
} rectangle;
```

we can write something like:

```
circle* c = // malloc + init
shape* s = (shape*)c;
(void)printf("my shape: id=%d x=%f y=%f\n",
             s->id, s->x, s-> y);
```

Shape procedures (1/2)

We do not need `circle_translate(..)`-like routines since you have this one:

```
void shape_translate(shape* s, float dx, float dy)
{
  s->x += dx;  s->y += dy;
}
```

and a sample use is: `shape_translate(s, 16, 64);`
or: `shape_translate((shape*)c, 16, 64);`

Shape procedures (2/2)

Printing a shape depends on what the shape to be printed is:

```
void shape_print(const shape* s)
{
  assert(s != NULL);
  switch (s->id) {
    case circle_id:
      circle_print((const circle*)s);
      break;
    case rectangle_id:
      rectangle_print((const rectangle*)s);
      break;
    default:
      assert(0);
  }
}
```

What have we done? (1/3)

Given a circle `s` (the same goes for a rectangle):

- you can call `shape_print(s)` instead of `circle_print(s)`
- so you can use a single routine per functionality

From a client (user of the *shape* module) point of view:

- she does not know that circles and rectangles exist
- she does not care about new types (triangle, etc.)

What have we done? (2/3)

You can write this sexy piece of code:

```
typedef struct
{
  shape** s;
  unsigned ns;
  /* ... */
} page;
```

```
void page_print(const page* p)
{
  assert(p != NULL);
  unsigned i;
  for (i = 0; i < p->ns; ++i)
    shape_print(p->s[i]);
}
```

What have we done? (3/3)

- we have introduced a new kind of type: *shape*

  is it a “concrete” type?
- we can extend the shape module, yet in an intrusive way

  just look at `shape_print`...
- we have factor some code

  `shape_translate` is valid for any shape
- we have also factor some data

  `x` and `y` are common to every shapes

and

- our program relies on casts such as: `circle*` $\longrightarrow$ `shape*`

Think different

Actually we have formed:

```
typedef struct
{
  shape s;
  float r;
} circle;
```

```
typedef struct
{
  shape s;
  float w, h;
} rectangle;
```

So that any shape (e.g., a circle) is:

- first a shape
- an extension of a shape with its own features (`r`)

Outline

Definitions

An **abstract class** is

- a class that represents an abstraction
- a class that cannot be instantiated
- a class with at least an abstract method

An **abstract method** is

- a method the code of whom cannot be given
- a method that is just declared
- a method that will be defined in other classes (some sub-classes)

Anti-definition

A **concrete class** is

- a class that does not represents an abstraction thus not an abstract class!
- a class that can be instantiated
- a class with no abstract method

Abstractions

`shape` is an **abstraction** for both `circle` and `rectangle`—an abstract type that represents several **concrete** types.

The method `shape::print` depends on which effective object we have to print; a circle? a rectangle? at that point we do not know.

In the corresponding C source, `shape_print` just dispatches towards `circle_print` or `rectangle_print`.

However:

- an abstract class can have attributes
  a shape have a center located at (`x`, `y`)
- an abstract class can provide methods with their definitions
  attributes ⇒ a constructor
  `shape::translate` can be written

Shape as a C++ abstract class (1/4)

In `shape.hh`:

```
class shape
{
public:                                    // 1
  shape(float x, float y);                 // 2
  void translate(float dx, float dy);      // 3
  virtual void print() const = 0;          // 4
  virtual ~shape() {}                      // 5
protected:                                 // 6
  float x_, y_;                            // 7
};
```

Shape as a C++ abstract class (2/4)

1. `shape` has an interface

   a public accessibility area with three methods

2. a constructor

   initializing attributes is a safe behavior

3. a translation method

   it will defined in `shape.cc`

4. a printing method

   just to *say* that we want to *print* shapes

4. a destructor

   just write it (no explanations here sorry...)

6. a "protected" accessibility area

   details are given later...

7. a couple of hidden attributes

   so they are suffixed by _

Shape as a C++ abstract class (3/4)

An abstract method is declared:

- starting with the “`virtual`” keyword
- and ending with “`= 0;`”

Calling `print` on a shape is then valid:

```
shape* s = // ...
s->print(); // OK
            // conforms to the declaration of 'shape::print'
```

Shape as a C++ abstract class (4/4)

In `shape.cc` nothing to be surprised of:

```
#include "shape.hh"

shape::shape(float x, float y)
{
  this->x_ = x;
  this->y_ = y;
}

void shape::translate(float dx, float dy)
{
  this->x_ += dx;
  this->y_ += dy;
}
```

Outline

"is-a"

The "**is-a**" relationship between classes is known as **inheritance** or **sub-classing**.

A circle "*is-a*" shape so:

- `circle` *inherits* from `shape`
- `circle` is a *sub-class* of `shape`

  `shape` is a *super-class* of `circle`

We also say that:

- `circle` derives from `shape`

  `circle` is a *derived class* of `shape`

  `shape` is a *base class* of `circle`
- `circle` extends `shape`

Class hierarchy

A set of classes glued together by the “is-a” relationship is called a class **hierarchy**.

- A hierarchy is usually a tree
- In this tree a class is depicted above its sub-classes

Practising (not just for fun)

OK:

- a rabbit is-an animal
- a wine is-a drink
- a tulip is-a flower
- (as an exercise find more examples)

OK as anti-examples:

- a guinea pig is-not-a pig
- a piece of cake is-not-a cake
- a program is-not-a language
- (find more)

Outline

Circle as a C++ subclass (1/X)

In `circle.hh`:

```
# include "shape.hh"                    // 7

class circle : public shape             // 8
{
public:                                 // 9
  circle(float x, float y, float r);    // 10
  virtual void print() const;           // 11
private:
  float r_;                             // 12
};
```

Circle as a C++ subclass (2/X)

7. knowing the class from whom `circle` inherits is required
8. the inheritance relationship is translated by "` : public`"
9. "`public:`" starts the class interface
10. a constructor
11. a printing method
    hint: explicitly tag it with the "`virtual`" keyword to remind the reader that this method is abstract in a super-class!
12. a single attribute in a private area

When “inheritance” makes sense (1/4)

Actually the class `circle` has *really* inherited from `shape`:

- the `translate` method
- the couple of attributes `x_` and `y_`

except that it is *implicit*

so we can say

- a `circle` has the ability of being translated
- `circle` has *three* attributes
  indeed:
  `sizeof(circle) == 3 * sizeof(float) + sizeof(void*)`

When “inheritance” makes sense (2/4)

If inheritance were explicit in class body, we would have:

```
class circle : public shape
{
public:
  circle(float x, float y, float r);
  virtual void print() const;
  void translate(float dx, float dy); // inherited!
private:
  float r_;
protected:
  float x_, y_;                       // inherited!
};
```

Circle as a C++ subclass (3/4)

In `circle.cc`:

```
#include "circle.hh"

circle::circle(float x, float y, float r) :
  shape(x, y)
{
  assert(r > 0.f);          // precondition
  this->r_ = r;
}

void circle::print() const
{
  assert(this->r_ > 0.f);  // invariant
  std::cout << '('
            << this->x_ << ", "
            << this->y_ << ", "
            << this->r_ << ')';
}
```

Circle as a C++ subclass (4/4)

A few remarks:

- the constructor of `circle` first calls the one of `shape` having a new circle first means having a new shape...
- the attributes `x_` and `y_` can be accessed just as they were defined in the `circle` class
- the "`virtual` keyword does not appear in source file

Outline

An object has two types!

Let us take a variable that represents an object.

The **static type** of the object is the type of the variable that represents the object. The static type is always known at compile-time.

The **dynamic type** of the object—or **exact type**—is its type at instantiation. The dynamic type is usually *unknown* at compile-time (but known at run-time).

Take a guess... (1/2)

In the following piece of code:

```
void foo(const shape& s)
{
  s.print();
}
```

what is the static type of the object represented by `s`?

and what is its dynamic type?

Take a guess... (2/2)

and with:

```
void foo(const shape& s)
{
  s.print();
}

int main()
{
  circle c // ...
  foo(c);
}
```

can you answer?

Valid transtyping (1/2)

Since a circle is a shape, you can write:

```
circle* c = new circle(1, 6, 64);  // dynamic allocation
shape* s = c;
```

A pointer to a shape is expected (`s`), you have a pointer to a circle (`c`); the assignment is valid.

The same goes for references (see the previous slide).

Valid transtyping (2/2)

What you can do:

- promote constant:

```
circle* c = // init
const circle* cc = c;
```

```
circle& c = // init
const circle& cc = c;
```

- changing static type from a derived class to a base class:

```
circle* c = // init
shape* s = c;
```

```
circle& c = // init
shape& s = c;
```

- both at the same time:

```
circle* c = // init
const shape* s = c;
```

```
circle& c = // init
const shape& s = c;
```

Resolving a method call

In this program:

```
void foo(const shape& s) { s.print(); }

int main()
{
  circle* c = new circle(1, 6, 64);
  foo(*c);
  delete c;  // memory deallocation
  c = 0;     // safety; nota bene: in C++ ''0'' replaces C's ''NULL''
  // ...
}
```

- which method is called by `foo`?
- which method is actually performed at run-time?
- why?

Outline

3 kinds

`public:`
accessible from everybody everywhere
example: `circle::r_get() const`

`private:`
only accessible from the current class
example: `circle::r_`

`protected:`
accessible from the current class *and* from its sub-classes
example: `shape::x_`

Outline

An exercise from the real world

Printing a page means printing every shapes of this page:

```
std::ostream& operator<<(std::ostream& ostr, const page& p)
{
  for // each shape s of p
    ostr << s << std::endl;
  return ostr;
}
```

How to make "`ostr << s`" work properly?

Hint for beginners

You can avoid many problems by following this advice:

- an abstract class can derive from another abstract class
- a concrete class should not derived from another concrete class

sorry that's not argued in this material...

Much further readings

- *Modularité, Objets et Types* by Didier Rémy. Lecture Material; available from

  `http://www.enseignement.polytechnique.fr/profs/informatique/Didier.Remy/mot/0/index.htm`

- *Object-Oriented Software Construction*, second edition by Bertrand Meyer, Prentice Hall, 1997.