C++ Workshop — Day 2 out of 5

Object-Orientation

Thierry Géraud, Roland Levillain, Akim Demaille
`{theo,roland,akim}@lrde.epita.fr`

EPITA — École Pour l'Informatique et les Techniques Avancées
LRDE — Laboratoire de Recherche et Développement de l'EPITA

2015–2016
(v2015-75-g3c99c09, 2015-12-02 11:02:33 +0100)

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
  $\rightarrow$ adding $=$ a **non-intrusive** process
- we do not break the "type-safe" property
  $\rightarrow$ a new type is not really an unknown type!

Program features

Expected features:

- both circles and rectangles can be translated (moved)
- 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 remember 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
  Be honest to your friends... When you lie, they get revenge

Abstract Class and Abstract Method

Definitions

An **abstract class**. . .

- is a class that represents an abstraction
- cannot be instantiated
- has at least one abstract method

An **abstract method** is

- a method whose code 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 represent 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 code invoked by `shape::print` depends on which actual object we have to print; a circle? a rectangle? at that point we do not know.

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 (1/3)

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

1. shape has an interface
   a public accessibility area
2. a constructor
   initializing attributes is a safe behavior
3. a destructor
   just write it (no explanations here sorry...)
4. a translation method
   it will be defined in `shape.cc`
5. a printing method
   just to *say* that we want to *print* shapes
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 (2/3)

To make a method abstract in C++, its declaration

- starts with "`virtual`"
- ends with "= 0"

Calling `print` on a `shape` is then valid:

```
#include "shape.hh"

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

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

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

```
#include "shape.hh"

shape::shape(float x, float y)
  : x_{x}, y_{y}
{}

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

Definitions + playing with words

“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* for `circle`
- `circle` extends `shape`

Class Hierarchy

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

- usually a tree
- depicted upside-down
  (superclasses at the top, subclasses at the bottom)

Practicing (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)

Subclassing

Circle as a C++ subclass

```
#include "shape.hh"                    // 8
class circle : public shape            // 9
{
public:                                // 10
  circle(float x, float y, float r);  // 11
  void print() const override;        // 12
private:
  float r_;                            // 13
};
```

8 knowing the class from which `circle` inherits is required

9 the inheritance relationship is translated by “`: public`”

10 “`public:`” starts the class interface

11 a constructor

12 a `print` definition, tagged with the “`override`” keyword.

13 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

- a `circle` can be translated
- `circle` has *three* attributes
  indeed: `sizeof(circle) == 3 * sizeof(float) + sizeof(void*)`
  (the '`void*`' is related to type identification...)

When "inheritance" makes sense (2/4)

If inheritance were explicit in the 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"
#include <cassert>

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

void circle::print() const // kw 'override' in .hh only
{
  assert(0.f < r_);  // invariant
  std::cout << '(' << x_ << ", " << y_ << ", " << 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
  as if they were defined in the `circle` class
- the "`virtual`" keyword must not appear in source file
  only in the declaration of the method
- likewise with "`override`"
  but `override` is not a keyword!
  Yet, don't use it as a variable name, *please!*

Transtyping

1. Rationale for inheritance
2. Inheritance in C++
3. Playing with types
   - Transtyping
   - Accessibility
   - Conclusion
4. Smart Pointers: Part I

An object has two types!

Let us take a variable that contains an object.

The static type of the object
: is the type of the variable that contains the object.
Always known at compile-time.

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

Take a guess... (1/2)

In the following piece of code:

```
#include "shape.hh"

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

what is the static type of the object in `s`?

and what is its dynamic type?

Take a guess... (2/2)

and with:

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

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

can you answer?

Valid transtyping (1/2)

Since a circle is a shape, you can write:

```
circle* c = new circle{1, 6, 64};
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 constness:

```
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()
{
  foo(circle{1, 6, 64});
}
```

- which method is called by `foo`?
- which method is actually performed at run-time?
- why? (a "vtable" equips this hierarchy...)

Accessibility

Three Kinds of Accessibility

- `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_`

These are called "access specifiers". It's about accessibility.
Please, don't use the word "visibility", it's something else.

`final`

- Sometimes you do not want to be derived from
- Even though you are a derived class
- `final` allows to flag such cases
- Sometimes, you'd like to help the compiler optimize your code
- Help it know a method will not be overriden

Final (1/2)

```
class A {
  // ...
  virtual void foo() = 0;
};

class B : public A {
  // ...
  void foo() override final;  // <- final impl
};

class C : public B {
  // ...
  // B::foo cannot be overridden here
};
```

Like for `virtual` and `override`, use only in declarations.

Final (2/2)

```
class A final {  // <- now the class is final
  // ...
};

class B : public A {
  // ...
  // does NOT compile because A cannot be derived
};
```

Conclusion

Dynamic Allocation & Deallocation

From C to C++:

C	`circle* c = (circle*)malloc(sizeof(circle));` `init_circle(c, 1, 6, 64);`
C++	`circle* c = new circle{1, 6, 64};`
C	`free(c);`
C++	`delete c;`
C	`int* buf = (int*)malloc(n * sizeof(int));`
C++	`int* buf = new int[n];`
C	`free(buf);`
C++	`delete[] buf;`

Memory management is not easy.

An exercise from the real world

Printing a page means printing every shapes of this page:

```
void print(const page& p)
{
  for (const shape& s: p) // each shape s of p
    print(s);
}
```

How to make "`print(s)`" work properly?

Hint for beginners

You can avoid many problems by following this advice:

- an abstract class can derive from an abstract class
- a concrete class should not derive from a 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://cristal.inria.fr/~remy/poly/mot/`
- *Object-Oriented Software Construction*, second edition by Bertrand Meyer, Prentice Hall, 1997.

Smart Pointers: Part I

(Raw) Pointers

Why Pointers?

- Pointers in C are a powerful means to play tricks with memory
  - Forget about forging an address from an integer
  - Forget about pointer arithmetic
- Pointers are an important means to refer to another place
  - They are "retargetable" references
  - These are "non-owning pointers"
- Pointers are 0/1 containers
  - `nullptr` for empty
  - Unclear ownership
  - C++ 17 promotes `std::optional` instead
- Pointers manage dynamically allocated memory
  - `new` "returns" a pointer
  - Clearly an owning pointer
  - However, in C++ we prefer value semantics
  - So this should be seldom used?

Why Pointers?

- Pointers in C are a powerful means to play tricks with memory
  - Forget about forging an address from an integer
  - Forget about pointer arithmetic
- Pointers are an important means to refer to another place
  - They are "retargetable" references
  - These are "non-owning pointers"
- Pointers are 0/1 containers
  - `nullptr` for empty
  - Unclear ownership
  - C++ 17 promotes `std::optional` instead
- Pointers manage dynamically allocated memory
  - `new` "returns" a pointer
  - Clearly an owning pointer
  - However, in C++ we prefer value semantics
  - So this should be seldom used?

Why Pointers?

- Pointers in C are a powerful means to play tricks with memory
  - Forget about forging an address from an integer
  - Forget about pointer arithmetic
- Pointers are an important means to refer to another place
  - They are "retargetable" references
  - These are "non-owning pointers"
- Pointers are 0/1 containers
  - `nullptr` for empty
  - Unclear ownership
  - C++ 17 promotes `std::optional` instead
- Pointers manage dynamically allocated memory
  - `new` "returns" a pointer
  - Clearly an owning pointer
  - However, in C++ we prefer value semantics
  - So this should be seldom used?

Why Pointers?

- Pointers in C are a powerful means to play tricks with memory
  - Forget about forging an address from an integer
  - Forget about pointer arithmetic
- Pointers are an important means to refer to another place
  - They are "retargetable" references
  - These are "non-owning pointers"
- Pointers are 0/1 containers
  - `nullptr` for empty
  - Unclear ownership
  - C++ 17 promotes `std::optional` instead
- Pointers manage dynamically allocated memory
  - `new` "returns" a pointer
  - Clearly an owning pointer
  - However, in C++ we prefer value semantics
  - So this should be seldom used?

Why Pointers?

- Pointers in C are a powerful means to play tricks with memory
  - Forget about forging an address from an integer
  - Forget about pointer arithmetic
- Pointers are an important means to refer to another place
  - They are “retargetable” references
  - These are “non-owning pointers”
- Pointers are 0/1 containers
  - `nullptr` for empty
  - Unclear ownership
  - C++ 17 promotes `std::optional` instead
- Pointers manage dynamically allocated memory
  - `new` “returns” a pointer
  - Clearly an owning pointer
  - However, in C++ we prefer value semantics
  - So this should be seldom used?

  Wrong!

Runtime Polymorphism

- We use pointers to get a "uniform handle" to objects
- But then again, what about ownership?
  - point to (or "reference to")
  - holds some `new`'d object
- Note that many OO languages offer *only* reference semantics
- So everything is actually a pointer
- Java, C#, etc.
- And the GC deals with the details

Runtime Polymorphism

- We use pointers to get a “uniform handle” to objects
- But then again, what about ownership?
  - point to (or “reference to”) do not `delete` it!
  - holds some `new`’d object
- Note that many OO languages offer *only* reference semantics
- So everything is actually a pointer
- Java, C#, etc.
- And the GC deals with the details

Runtime Polymorphism

- We use pointers to get a "uniform handle" to objects
- But then again, what about ownership?
  - point to (or "reference to") do not `delete` it!
  - holds some `new`'d object do `delete` it!
- Note that many OO languages offer *only* reference semantics
- So everything is actually a pointer
- Java, C#, etc.
- And the GC deals with the details

The Problem with Pointers

The only question is:

`delete`, or not `delete`

The Problem with Pointers

The only question is:

`delete`, or not `delete`

owner, or not owner

Smart Pointers

- look like pointers
- behave like pointers
- manage ownership
- they make your programs more robust!

Shared Pointers

Pointers and Containers

```
#include <iostream>
#include <vector>

#define PING() std::cerr << __PRETTY_FUNCTION__ << '\n'

struct shape
{
  virtual ~shape() { PING(); };
  virtual void print() const = 0;
};

struct circle: shape
{
  void print() const override { PING(); }
};

struct square: shape
{
  void print() const override { PING(); }
};
```

Pointers and Containers

```cpp
int main()
{
  using shape_ptr
    = const shape*;
  auto ss
    = std::vector<shape_ptr>{};
  ss.emplace_back(new circle{});
  ss.emplace_back(new square{});
  for (auto s: ss)
    s->print();
}
```

- don't worry about `std::vector`
- we'll see that tomorrow
- a dynamic (resizable) array of `shape_ptr`
- `emplace_back` means "build and append"

Pointers and Containers

```
int main()
{
  using shape_ptr
    = const shape*;
  auto ss
    = std::vector<shape_ptr>{};
  ss.emplace_back(new circle{});
  ss.emplace_back(new square{});
  for (auto s: ss)
    s->print();
}
```

- don't worry about `std::vector`
- we'll see that tomorrow
- a dynamic (resizable) array of `shape_ptr`
- `emplace_back` means "build and append"

```
virtual void circle::print() const
virtual void square::print() const
```

Pointers and Containers

```cpp
int main()
{
  using shape_ptr
    = std::shared_ptr<const shape>;
  auto ss
    = std::vector<shape_ptr>{};
  ss.emplace_back(new circle{});
  ss.emplace_back(new square{});
  for (auto s: ss)
    s->print();
}
```

Pointers and Containers

```cpp
int main()
{
  using shape_ptr
    = std::shared_ptr<const shape>;
  auto ss
    = std::vector<shape_ptr>{};
  ss.emplace_back(new circle{});
  ss.emplace_back(new square{});
  for (auto s: ss)
    s->print();
}
```

```
virtual void circle::print() const
virtual void square::print() const
virtual shape::~shape()
virtual shape::~shape()
```

Pointers and Containers

```
int main()
{
  using shape_ptr
    = const shape*;
  auto ss
    = std::vector<shape_ptr>{};
  ss.emplace_back(new circle{});
  ss.emplace_back(new square{});
  for (auto s: ss)
    s->print();
}
```

```
int main()
{
  using shape_ptr
    = std::shared_ptr<const shape>;
  auto ss
    = std::vector<shape_ptr>{};
  ss.emplace_back(new circle{});
  ss.emplace_back(new square{});
  for (auto s: ss)
    s->print();
}
```

```
virtual void circle::print() const
virtual void square::print() const
```

```
virtual void circle::print() const
virtual void square::print() const
virtual shape::~shape()
virtual shape::~shape()
```

Shared Ownership

```
for (unsigned i = 0; i < 10; ++i)
  {
    unsigned n = rand() % 10; // bad quality random, but quick to write
    if (n < ss.size())
      ss.emplace_back(ss[n]);
    else if (n % 2)
      ss.emplace_back(new circle{});
    else
      ss.emplace_back(new square{});
  }
for (auto s: ss)
  s->print();
```

Shared Ownership

```
for (unsigned i = 0; i < 10; ++i)
  {
    unsigned n = rand() % 10; // bad quality random, but quick to write
    if (n < ss.size())
      ss.emplace_back(ss[n]);
    else if (n % 2)
      ss.emplace_back(new circle{});
    else
      ss.emplace_back(new square{});
  }
for (auto s: ss)
  s->print();
```

Good luck with memory management...

Shared Ownership

```
virtual void circle::print() const
virtual void circle::print() const
virtual void circle::print() const
virtual void square::print() const
virtual void circle::print() const
virtual void circle::print() const
virtual void circle::print() const
virtual void square::print() const
virtual void square::print() const
virtual void circle::print() const
virtual shape::~shape()
virtual shape::~shape()
virtual shape::~shape()
virtual shape::~shape()
virtual shape::~shape()
virtual shape::~shape()
```

Avoid `new`, prefer `make_shared`

- `shared_ptr<Foo>{new Foo{arg}}` don't
  - exception unsafe
  - two allocations
  - redundancy (twice `Foo`)
  - contains a `new` without its `delete`
- `std::make_shared<Foo>(arg)` do

Part I

Appendix

Some Sugar

Introducing auto and decltype:

```
auto p = std::make_shared<test>();
p->noop();

decltype(p) p2 = p;
std::cout << p.get() << ' ' << p2.get() << '\n'; // same addr
std::cout << p.use_count() << '\n'; // 2
```

auto is often for

you_dont_want_the_write_a_type_because_it_is_too_long_and_or_obvious

auto and decltype are also great to rely on the compiler.

A Hierarchy in C

5 A Hierarchy in C

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 feature

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 factored some code

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

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

and

- our program relies on casts such as: `circle*` $\rightarrow$ `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`)