C++ Workshop — Day 2 out of 5

Object-Orientation

Thierry Géraud, Roland Levillain, Akim Demaille
theo@lrde.epita.fr

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

2015–2021
January 27, 2021

Outline

Outline

Outline

Outline

Outline

Outline

Dynamic Allocation & Deallocation

From C to C++:

C	`circle* c = (circle*)malloc(1 * sizeof(circle));`
	or: `circle* c; 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.

Why Pointers?

- Pointers in C are a powerful means to play with memory
  ```
  *p++ = a;
  ```
- Pointers are an important means to refer to another place
  ```
  p = &a; /*...*/ p = &b;
  ```
- Pointers are 0/1 containers
  ```
  if (p != nullptr) p->run();
  ```
- Pointers manage dynamically allocated memory
  ```
  p = new int[n];
  ```

Why Pointers?

- Pointers in C are a powerful means to play with memory
```
*p++ = a;
```
- Pointers are an important means to refer to another place
```
p = &a; /*...*/ p = &b;
```
- Pointers are 0/1 containers
```
if (p != nullptr) p->run();
```
- Pointers manage dynamically allocated memory
```
p = new int[n];
```

Wrong!

Why Pointers?

- Pointers in C are a powerful means to play **tricks** with memory
  - Forget about forging an address from an integer (Can you say why?)
  - Forget about pointer arithmetic

Why Pointers?

- Pointers in C are a powerful means to play **tricks** with memory
  - Forget about forging an address from an integer (Can you say why?)
  - Forget about pointer arithmetic
- Pointers are an important means to refer to another place
  - They are "retargetable" references / These are "non-owning pointers"

    When a pointer dies, it dies alone!

Why Pointers?

- Pointers in C are a powerful means to play **tricks** with memory
  - Forget about forging an address from an integer (Can you say why?)
  - Forget about pointer arithmetic
- Pointers are an important means to refer to another place
  - They are "retargetable" references / These are "non-owning pointers"
    When a pointer dies, it dies alone!
- Pointers are 0/1 containers
  - `nullptr` for empty (Forget about `0` and `NULL`)
  - Unclear ownership
  - C++17 promotes `std::optional` instead

Why Pointers?

- Pointers in C are a powerful means to play **tricks** with memory
  - Forget about forging an address from an integer (Can you say why?)
  - Forget about pointer arithmetic
- Pointers are an important means to refer to another place
  - They are "retargetable" references / These are "non-owning pointers"
    When a pointer dies, it dies alone!
- Pointers are 0/1 containers
  - `nullptr` for empty (Forget about `0` and `NULL`)
  - 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?

Runtime Polymorphism

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

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!

    vs
  - hold some `new`'d object
- Note that many OO languages offer *only* reference semantics
  - So everything is actually a pointer
  - Java, C#, etc.
  - And a Garbage Collector (GC) deals with the details (hopefully for the programmer)

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!
    vs
  - hold 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 a Garbage Collector (GC) deals with the details (hopefully for the programmer)

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

Smart pointers:

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

They are so smart!

Outline

Pointers and Containers

```
struct phoenix
{
  void fly() const {
    std::cout << "fly" << '\n';
  }
  ~phoenix() {
    std::cout << "die!" << '\n';
  }
};

int main()
{
  using phoenix_ptr
    = const phoenix*;
  auto v
    = std::vector<phoenix_ptr>{};
  v.push_back(new phoenix{});
  v.emplace_back(new phoenix{});
  for (auto s : v)
    s->fly();
}
```

`std::vector`

- a dynamic (so resizable) array of `phoenix_ptr`
- both `emplace_back` and `push_back` mean "append"...

The `for` loop reads:

"for each `s` in `v` do"

Result:

```
fly
fly
```

Pointers and Containers

Replacing "`const phoenix*`" by "`std::shared_ptr<const phoenix>`":

```
int main()
{
  using phoenix_ptr
    = const phoenix*;
  auto v
    = std::vector<phoenix_ptr>{};
  v.emplace_back(new phoenix{});
  v.emplace_back(new phoenix{});
  for (auto s : v)
    s->fly();
}
```

gives:

```
fly
fly
```

```
int main()
{
  using phoenix_ptr
    = std::shared_ptr <const phoenix>;
  auto v
    = std::vector<phoenix_ptr>{};
  v.emplace_back(new phoenix{});
  v.emplace_back(new phoenix{});
  for (auto s : v)
    s->fly();
}
```

gives:

```
fly
fly
die!
die!
```

Avoid new, prefer make_shared

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

Some Sugar

Introducing decltype:

with :

```
struct test { void noop() { /*...*/ } };
```

```
auto p = std::make_shared<test>();
p->noop();  // p is used just like a pointer :-)

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

auto is often for

you_dont_want_to_write_a_type_because_it_is_too_long_and_or_obvious

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

What's the problem?

Reminder:

```
class easy
{
public:
  easy();
  ~easy();
private:
  float* ptr_;
};

easy::easy()
{ // allocate a resource so...
  this->ptr_ = new float;
}

easy::~easy()
{ // ...deallocate it!
  delete this->ptr_;
  this->ptr_ = nullptr; // safety
}
```

The call `naive(run)` makes bug being a copy of `run`, so we have "`bug.ptr_ == run.ptr_`"; then `delete` is called **twice** on this addr with `bug.~easy()` (end of `naive`) and `run.~easy()` (end of `main`)!

```
void naive(easy bug)
{
  // nothing done so ok!
}

int main()
{
  easy run;
  naive(run);
}

// compiles but fails at run-time!!!
```

What's the problem?

Solution 1: with & and delete

```
class easy
{
public:
  easy();
  easy(const easy&) = delete;
  void operator=(const easy&) = delete;
  ~easy();
private:
  float* ptr_;
};

easy::easy()
{ // allocate a resource so...
  this->ptr_ = new float;
}

easy::~easy()
{ // ...deallocate it!
  delete this->ptr_;
  this->ptr_ = nullptr; // safety
}
```

```
void naive(const easy& bug) // \o/
{
  // great, 'bug' is not a copy!
}

int main()
{
  easy run;
  naive(run);
}

// compiles and runs
```

What's the problem?

Solution 2: *shallow copy* with `std::shared_ptr`

```
class easy
{
public:
  easy() {
    ptr_ = std::make_shared<float>();
  }
  easy(const easy&) = default;
  easy& operator=(const easy&) = default;
  ~easy() = default;
private:
  std::shared_ptr<float> ptr_;
};
```

```
void naive(easy bug) // copy
{
  // ptr_ is shared between
  // 'run' and 'bug'
}

int main()
{
  easy run;
  naive(run);
}

// compiles and runs
```

The smart pointers do the work :-)

What's the problem?

Solution 3: *deep copy*

```
class easy
{
public:
  easy() = default;
  easy(const easy& that)
    // deep copy => get() is mandatory
    : ptr_{std::make_shared<float>(
                    *that.ptr_.get())}
  {}
  void operator=(const easy&) = delete;
  ~easy() = default;
private:
  std::shared_ptr<float> ptr_;
};
```

```
void naive(easy bug) // copy
{
  // So ptr_ is *not* shared between
  // 'run' and 'bug'!!!
}

int main()
{
  easy run;
  naive(run);
}

// compiles and runs
```

An unsatisfactory solution: we should have use `std::unique_ptr`

Outline

'std::optional' (C++17)

```
// in <cstdlib>
namespace std {
  int atoi(const char* str);  // converts a string to an int
}
```

```
auto i = std::atoi("0");  //
auto j = std::atoi("Pastis 51");
```

What's the problem?

```
namespace my {
  int atoi(const std::string& s, bool& ok)  // this is my::atoi
  {
    int i;
    std::istringstream{s} >> i;
    ok = std::to_string(i) == s;
    return i;
  }
}
```

What's the problem?

'std::optional'

```
namespace my
{
  std::optional<int> atoi(const std::string& s)
  {
    int i;
    std::istringstream{s} >> i;
    if (std::to_string(i) == s)
      return i;
    return {}; // default is ``no object''
               // or use std::nullopt
  }
}
```

Usage:

```
auto i = my::atoi("51");
if (i) // or i.has_value()
  std::cout << i.value() << std::endl;  // what's printed?

auto s = "51";
auto j = my::atoi(s).value_or(0);    // ;-)
```

'std::optional'

Forget:

```
class car { // ...
private:
  wheel* spare_;  // nullptr or addr of 1 object
  // ...
};
```

this version is better:

```
class car { // ...
private:
  std::shared_ptr<wheel> spare_;  // nullptr or 1 shared object
  // ...
};
```

or this one, with a different semantics:

```
class car { // ...
private:
  std::optional<wheel> spare_;  // 0 or 1 (copyable) object, not a ptr
  // ...
};
```

Outline

Outline

After day 1

We have

- a toy `circle` class
- nice features (encapsulation / information hiding)

We want rectangles!
→ 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 (`rectangle`) 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

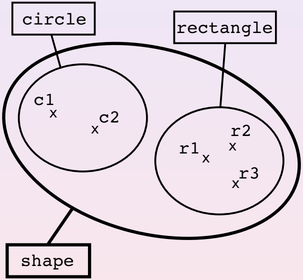
So we want to handle *shapes*:

- circles and rectangles are shapes
- a circle is a shape / a rectangle is a shape
- shapes can be translated and printed
- a shape is either a circle or a rectangle

Think this way, please

Consider that:

- a type (class) is like a mathematical set
- an instance (object) is like an element



`r2` is a `rectangle` $\equiv$ `r2` is an element belonging to the set `rectangle`

$\equiv$ `r2` is an instance of the class `rectangle`

$\Rightarrow$ `r2` is also a `shape`

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?)
- such an extension should be non-intrusive

The 3 notions "sub-module / subset / sub-class" are strongly related.

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!

Outline

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 (in an abstract class)
- a method that will be defined in some other classes (all the concrete sub-classes of the abstract class)

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
- (*piece of advice: a class which is not a "superset", which has no "subclass"*)

Abstractions

`shape` is an **abstraction** for both `circle` and `rectangle`;
`shape` is 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 (abstract)
   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'
```

We are just *unable* to code shape::print (so it is abstract).

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;
}
```

An abstract class looks like a concrete one.

Outline

"is-a"

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

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

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

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

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:

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

```cpp
#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 base class of `circle` is required

9 the sub-class 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);
  void print() const override;
  void translate(float dx, float dy); // inherited!
private:
  float r_;
protected:
  float x_, y_;                       // inherited!
};
```

so you do not write such code...

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}, r_{r}
{
  assert(r > 0.f);    // precondition
}

void circle::print() const  // kwd 'override' in .hh only
{
  assert(r > 0.f);  // invariant
  std::cout << '(' << x_ << ", " << y_ << ", " << r_ << ')';
}
```

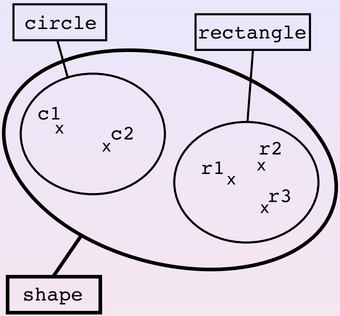
Circle as a C++ subclass (4/4)

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!*

Outline

Reminder



A `circle` is-a `shape`:

⇒ an element of the set `circle` belongs to its super-set `shape`

= an instance of the class `circle` is an instance of the super-class `shape`

Outline

An object and two types

Let us take a variable that designates an object.

The static type of the object is the type of the variable.
Always known at *compile-time*.

The dynamic type of the object is its type at instantiation.
We say also "exact type".
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();  // OK: print is declared in shape:: and is const
}
```

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

and what is its dynamic type?

Important notice:
a variable with an abstract type (such as `s`) is always a pointer or a reference.

Take a guess... (2/2)

and with:

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

int main()
{
  foo(circle{1,51,5});
}
```

can you answer?

Remark that we can “const reference” a temporary object!

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 give a pointer to a circle (`c`); this 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 the 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:

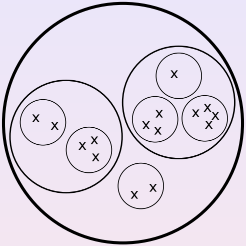
```
void foo(const shape& s) { s.print(); }

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...)

Quiz

In C++, how many different types for an object?



The case of C:

```
struct triangle* p;
void* q = p;
```

```
struct shape {
  float x, y;
  union {
    struct circle* c;
    struct rectangle* r;
  } sub;
};
```

Outline

Three Kinds of Accessibility

- `public`
  accessible from everybody everywhere
  example: `circle::get_r() 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 a derived class to redefine a method
- `final` allows to flag such cases

- Sometimes you do not want to be derived from
- `final` allows to flag such cases

Actually:

- 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 from
};
```

Thus all the methods of `A` are `final`.

Outline

An exercise from the real world

Printing a page means printing every shapes of this page:

```
void print(const page& p)
{
  // for each shape s in the container returned by p.shapes()
  for (const shape& s : p.shapes())
    print(s);
}
```

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

Yes, we want a procedure / function; that's a bit dummy but it's an exercise...

Soluce

```
void print(const page& p)
{
  for (const shape& s : p.shapes())
    print(s);
}

void print(const shape& s)  // no conflict with the 1st print
                            // this is overloading (see tomorrow)
{
  s.print();  // dispatches = calls either circle::print,
              //                     or rectangle::print,
              //                     or...
}
```

Dispatch is only for a method call w.r.t. the dynamic type of the target

A procedure does *not* dispatch!

→ “`s.print()`” dispatches; “`print(s)`” does *not*.

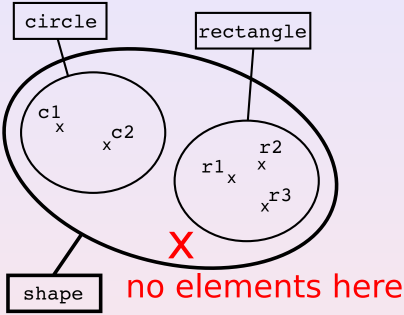
Hint for beginners

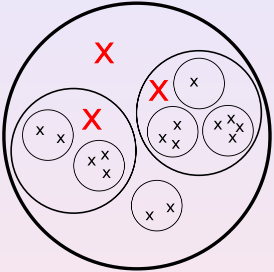
You can avoid many problems by following this advice:

- 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...

Back with Sets





You can only **create** instances (elements)
of leaf classes (deepest sub-sets) of the hierarchy

Hohoho!

$$\text{Object-Orientation (OO)} = \text{Object (O)} + \text{Class hierarchies}$$

Inheritance is just an artifact of class / set inclusion!

- Rationale: if a `shape` can give its color, then a `circle` can!
- So prefer the term **class hierarchies** over **inheritance**.

Idioms of Special Methods with Hierarchies

```
class base // are belong to us
{
public:
  base();
  base(int b /*...*/);
  base(const base& rhs);
  base& operator=(const base& rhs);
  virtual ~base();
protected:
  int b_;
  //...
};
```

```
class derived : public base
{
public:
  derived();
  derived(int b, float d);
  derived(const derived& rhs);
  derived& operator=(const derived& rhs);
  virtual ~derived();
private:
  float d_;
  //...
};
```

Idioms of Special Methods with Hierarchies

```
derived::derived()
  : base(),
    d_(0) //...
{
  // allocate resource when needed
}

derived::derived(int b, float d)
  : base(b /*...*/),
    d_(d) //...
{
  // allocate resource when needed
}

derived::derived(const derived& rhs)
  : base(rhs),
    d_(rhs.d_) //...
{
  // allocate resource when needed
}
```

```
derived&
derived::operator=(const derived& rhs)
{
  if (&rhs != this)
  {
    this->base::operator=(rhs);
    this->d_ = rhs.d_; //...
  }
  return *this;
}

derived::~derived()
{
  // resource deallocation when needed
  // warning: do NOT call base::~base()
}
```

Again: please do not think,
just do like that (!)

Hints

- please *strictly* follow the idioms given in the previous slide
- `this->b_`, as an attribute of `base`, is not processed in the special methods of `derived`
- each constructor of `derived` first calls the appropriate constructor of `base`
- if a class has a `virtual` method, its destructor shall be tagged `virtual`
- in the destructor body (there is one per class), do *not* call the destructor of base classes
- in constructors and destructor bodies, do *not* call on `this` any `virtual` method from the same hierarchy

Outline

Pointers and Containers

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

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

class shape {
public:
  virtual ~shape() { PING(); }
  virtual void print() const = 0;
};

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

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

Pointers and Containers

Replacing "`const shape*`" by "`std::shared_ptr<const shape>`":

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

gives:

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

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

gives:

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

Outline

Outline

Development v. release

- Use `assert` during the *development* process
  - to detect (and correct) bugs as early as possible
  - to ease and speed up the process
- In *release* process
  - a program should be robust
    does not stop if a problem arises
  - so handling errors is not the assert-way
  - so you have to write specific code for that

Development v. release

Handling errors correctly means

- **recovering** a *coherent* and *stable* execution state
- having some transversal code in programs

  it is an "*aspect*" of your program

Development v. release

About C-like error handling:

- the client has to test procedure return values

  and usually forgets to do so

- when an error is detected, you have to code the “unstacking” (procedure calls, and also methods in C++) process (“unwinding”) to get to where the error has to be processed...

- that is tedious...

A simple illustration in C

without error management:

```c
void baz() {
  // ...
  // an error happens here
  // ...
}


void bar() {
  // ...
  baz();
  // ...
}


void foo() {
  // ...
  bar();  // erroneous result...
  // ...
}
```

with error management:

```c
int baz() {
  // ...
  if (test)
    return -1; // err detected!
  // ...
}

int bar() {
  // ...
  if (baz() == -1)
    return -1; // unstacking...
  // ...
}

void foo() {
  // ...
  if (bar() == -1) {
    // err handling...
  }
  // ...
}
```

Definitions

- An **exception** is an object that represents the error.
- Such an object lives until the error has been properly processed.
- A routine that detects an error `throw`s an exception

  in the previous example, it is the case for `baz`
- A routine in which an error might occur can `catch` this error to do something about it

  in the previous example, it is surely the case of `foo` but also the same for `bar`

Outline

Error hierarchies

An exception is an object so you (as a client) can define to describe errors:

```
#include <exception>

namespace error
{
  class any : public std::exception {};
  class math : public any {}; // abstract class

  // Concrete classes.
  class overflow : public math {};
  class zero_divide : public math {};
}
```

An `error::zero_divide` *is-an* `error::math`.

Throwing an exception

```
float div(float x, float y)
{
  // code for handling err in dev mode:
  assert(y != 0);

  // code for handling err in release mode:
  if (y == 0)
    throw error::zero_divide();  // call to a ctor

  // code when everything is OK
  return x / y;
}
```

Sample behavior

Consider that program:

```
void baz() {
  // code 3
  div(a, b); // here!
  // code 4
}

void bar() {
  // code 2
  baz();
  // code 5
}

void foo() { // called somewhere
  // code 1
  bar();  // if not OK, continue
  // code 6
}
```

If `b != 0` in baz, execution performs:

- first code 1 to code 3,
- then div(a, b) that works fine,
- lastly code 4 to code 6.

If `b == 0`, it should perform:

- first code 1 to code 3,
- div(a, b) that does *not* work,
- then some specific code to handle this error!
- and finally code 6 (program resumes)

Handling error

With error handling code in "foo":

```
void baz() {
  // code 3
  div(a, b); // can fail!
  // code 4
}

void bar() {
  // code 2
  baz();
  // code 5
}
```

```
void foo()
{
  try {
    // code 1
    bar();
    // code 6
  }
  catch (...) {
    // "..." means "any exception"
    std::cerr << "bar aborted!\n";
  }
}
```

If no error: code 1 → code 2 → code 3 → div → code 4 → code 5 → code 6

If error: code 1 → code 2 → code 3 → div → err msg

Recovery from error

```
void bar()
{
  data* ptr = nullptr;
  try {
    // ...
    baz();
    // ...
    ptr = new data; // dyn alloc
    // ...
    baz();
    // ...
  }
  catch (...) {
    delete ptr;
    throw;
  }
}
```

- the 2nd call to baz might fail
- in this example, some action is performed before this call (`ptr` allocation)
- bar *has to* perform some recovery code if an error occurs during that call (`ptr` deallocation)
- the `catch` code block is run when an exception has been thrown
- error handling is not completed so the caught exception is thrown again (instruction `throw;`); the error is still alive...

Handling error (2/2)

With a more complete error handling code:

```
void baz() {
  try {
    // code 3
    div(a, b); // can fail!
    // code 4
  }
  // code Z: catch, fix and throw
}

void bar() {
  try {
    // code 2
    baz();
    // code 5
  }
  // code R: catch, fix and throw
}
```

```
void foo()
{
  try {
    // code 1
    bar();
    // code 6
  }
  catch (...) {
    // "..." means
    //         "any exception"
    std::cerr
        << "bar aborted!\n";
  }
}
```

Selecting errors to handle

```
void foo() {
  try {
    // ...
  }
  catch (error::zero_divide) {
    // handles such error
  }
  catch (error::math) {
    // handles other math errors
  }
  catch (error::any) {
    // handles non-math client errors
  }
  catch (std::bad_alloc) {
    // handles an allocation ('new') that failed
  }
  catch (...) {
    // handles all remaining kinds of errors
  }
}
```

- `catch` clauses are inspected in the order they are listed
- the appropriate `catch` clause is selected from the error type
- the corresponding code is run

Outline

The “real” Class

```cpp
namespace error
{
  class problem : public any
  {
  public :
    problem(const std::string& fname,
            unsigned line,
            const std::string& msg);
    unsigned line() const;
    // ...
  private :
    std::string fname_;
    unsigned line_;
    std::string msg_;
  };
}
```

```cpp
// in namespace error::.
std::ostream&
operator<<(std::ostream& o,
           const problem& p)
{
  o << "err in " << p.fname()
    << "at line " << p.line()
    << ": " << p.msg();
  return o;
}
```

Using the exception object

An exception is thrown
an object is constructed

```
void parse(const std::string& s)
{
  // ...
  throw error::problem(__FILE__,
                       __LINE__,
                       "ICE!");
  // ...
}
```

The exception is caught
the object is inspected

```
void compile()
{
  try {
    // parse something...
  }
  catch(error::problem& pb) {
    std::cerr << pb << '\n';
    // pb is a regular object!
  }
};
```