C++ Workshop — Day 3 out of 5

Polymorphisms

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 20, 2021

Outline

Outline

Outline

Outline

Outline

Comparison

C:

> You shoot yourself in the foot and then nobody else can figure out what you did.

vs

C++:

> You accidentally create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical assistance is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying: "That's me, over there."

Outline

Outline

Namespace

```
namespace my
{
  class vector
  {
    // ...
  };

  // here no need to use the prefix my::

} // end of namespace my::  (no ';' here)
```

The class *full* name is `my::vector`.
It cannot be confused with `std::vector<T>` from the standard library.
A same namespace can be "split" into different files.
The use of namespaces is first for *modularity* purpose.

Namespace std

The C++ standard library is in std.

```
namespace std
{
  // a container class:
  template <class _Tp,
            class _Alloc = __STL_DEFAULT_ALLOCATOR(_Tp) >
  class list : protected _List_base<_Tp, _Alloc>
  {
    // ...
  };

  // an object:
  _IO_ostream_withassign cout;   // you, do not _athing

  // a type alias:
  using string = basic_string<char>;

  // a procedure:
  istream& operator>>(istream&, unsigned char&);
}
```

Outline

Loops

Consider this code:

```
auto v = std::vector<int>{1, 2, 4, 8};

for (FIXME i : v) // <- FIXME: there's a FIXME...
  std::cout << i << ' ';
```

it displays "1 2 4 8 "

we can have these loops:

```
for (auto i : v)        // access by value (type of i = int)
for (const int& i : v) // access by const reference
for (auto&& i : v)      // access by reference (type of i = int&)

for (auto&  i : v)      // likewise
                        // but only for modifiable lvalues
```

Outline

Buffer

In C++, instead of creating buffers:

```
int* buf = new int[n];
```

you usually prefer to rely on *dynamic arrays* (more about that later):

```
auto arr = std::vector<int>(n); // Parentheses, not braces!
```

or use some other types of std containers...

E.g., the class "page" contains:

```
std::vector<shape*> s_; // attribute
```

or better:

```
std::vector<std::shared_ptr<shape>> s_;
```

Outline

Four different kinds of polymorphism

Polymorphism can be:

	C	C++
coercion	yes	yes
inclusion	no	"yes"
overloading	no	yes
parametric	no	yes

In many OO books, "polymorphism" means "method polymorphism thanks to subclassing" (it is related to *inclusion* polymorphism)...

A routine is *polymorphic* if it accepts input with different types.

Coercion

```
bool is_positive(double d) { return d > 0.; }

void bar()
{
  int i = 3;
  std::cout << is_positive(i) << '\n';

  float f = 4;
  std::cout << is_positive(f) << '\n';
}
```

At each call, **two** instances are involved:

- the client one (resp. `i` and `f`), to be converted
- the argument of `is_positive` (`d`), result of the **conversion** (cast)

Inclusion

```
// abstract class:
class scalar
{
  // ...
  virtual bool is_positive() const = 0;
  // ...
};

// concrete classes:
class my_int    : public scalar { /*...*/ };
class my_float  : public scalar { /*...*/ };
class my_double : public scalar { /*...*/ };

// routine:
bool is_positive(const scalar& s) { return s.is_positive(); }
```

Inclusion

```
void bar()
{
  my_int i = 1;
  std::cout << is_positive(i) << '\n';

  my_float f = 2;
  std::cout << is_positive(f) << '\n';
}
```

Thanks to inheritance, `is_positive` works for any subclass of `scalar`.

Transtyping is in use here: `i` (resp. `f`), which is a `my_int` (resp. a `my_float`) is cast to `scalar` when passed as argument to `is_positive`.

Overloading

```
bool is_positive(int i)    { return i > 0;   }
bool is_positive(float f)  { return f > 0.f; }
bool is_positive(double d) { return d > 0.;  }
bool is_positive(unsigned) { return true;    }

void bar()
{
  int i = 1;
  std::cout << is_positive(i) << '\n'; // calls is_positive(int)

  float f = 2;
  std::cout << is_positive(f) << '\n'; // calls is_positive(float)
}
```

Several versions of an operation (`is_positive`);
signatures are different and not ambiguous for the client.

Operator overloading

To be able to write:

```
auto s = std::string{"hello world"};
std::cout << s << '\n';  // calls an std::operator<<(..)

std::cout << circle{1,2,3} << '\n';
std::cout << rectangle{1,2,3,4} << '\n';
```

that means that several operator<< coexist:

```
// in C++ std lib:
namespace std {
  ostream& operator<<(ostream&, const string&);
}
// in your program:
std::ostream& operator<<(std::ostream&, const circle&);
std::ostream& operator<<(std::ostream&, const rectangle&);
```

Method overloading (1/2)

```
class circle : public shape
{
public:
  circle();
  circle(float x, float y, float r);
  float  x() const;
  float& x();
  //...
};
```

- a couple of constructors `circle::circle`

  but "`circle::circle()`" $\neq$ "`circle::circle(float, float, float)`"

- a couple of methods `x`

  but "`circle::x() const`" $\neq$ "`circle::x()`"

Method overloading (2/2)

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

circle::circle()
  : circle{0.f, 0.f, 1.f}
{}
```

```
float circle::x() const
{
  return x_;
}

float& circle::x()
{
  return x_;
}
```

nice, we can write:

```
// with 'c' a non-const circle
c.x() = 3.14f;
```

Parametric polymorphism

```
template <typename T>  // reading: for all type T, we have
bool is_positive(T t)
{
  return t > 0;
}

void bar()
{
  int i = 1;
  std::cout << is_positive(i) << '\n'; // calls is_positive<int>

  float f = 2;
  std::cout << is_positive(f) << '\n'; // calls is_positive<float>
}
```

How it works (1/2)

In `template <typename T> bool foo(T t);`

- the formal parameter `T` represents a type (keyword `typename`)
- this kind of procedure is a **description** of a **family of procedures**
- values of `T` are not known yet
- the call "`foo(i)`" forces the compiler to set a value for `T` (with "`int i`" the call "`foo(i)`" means that `T` is `int`)
- a *specific* procedure is then compiled for this value / this specific case; namely it is `foo<int>`
- at last, two different routines are compiled: `foo<int>` and `foo<float>`, and their binary codes differ!

How it works (2/2)

We end up with overloading because...
...the program is **transformed by the compiler** into:

```
bool is_positive<int>  (int   t) { return t > 0;   }
bool is_positive<float>(float t) { return t > 0.f; }

void bar() {
  int i = 1;    std::cout << is_positive<int>(i) << '\n';
  float f = 2;  std::cout << is_positive<float>(f) << '\n';
}

// do not write that, the compiler does it!
```

With parameterization:

- there is no coercion in passing arguments
- `is_positive` is written once

auto ->

When the classical writing:

```
return_type routine(list_of_args l)
```

is better written:

```
auto routine(list_of_args l) -> return_type
```

you can write:

```
template <typename T1, typename T2>
auto plus(const T1& t1, const T2& t2) -> decltype(t1 + t2)
{
  return t1 + t2;
}
```

Outline

Outline

Through the `template` keyword

Formal parameter

variable attached to an entity and valued *at compile-time*

C++ entities that can be parameterized are:

- procedures, e.g., `is_positive<int>`
- methods, e.g., a ctor of `std::pair<T1,T2>` (see later)
- classes, e.g., `vec<3,float>` (algebraic vector of $\mathbb{R}^3$)

Valuation

- should be explicit for expressing classes;
- it is not mandatory for calling routines:
  we can write "`is_positive(i)`" instead of
  "`is_positive<int>(i)`"

Mathematical example (1/2)

mathematical function:

$$a \in \mathbb{N},\ f_a : \begin{cases} \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & \sin(ax) \end{cases}$$

equivalent C++ piece of code:

```
template <unsigned a>
float f(float x)
{
  return sin(a * x);
}
```

- x is an **argument** $\Leftrightarrow$ valued at **run-time**
- a is a **parameter** $\Leftrightarrow$ valued at **compile-time**

Mathematical example (2/2)

f_a	a parametric function	`f<a>`	a description of code `f(1)` does not compile
f_2	a function	`f<2>`	a procedure so is compilable
$f_2(1)$	a value	`f<2>(1)`	a procedure call so a value (returned)

Outline

A Simple Example (1/4)

A Non-Parameterized Class

Two algebraic vector classes:

```
class vec_2D
{
public:
  //...
private:
  float coord_[2]; // coordinates
};

class vec_3D
{
public:
  // oops, the code here is very
  // redundant with the one above..
private:
  float coord_[3];
};
```

You do *not* write such code:

- Code redundancy is awkward...
- Copy-paste-modify is evil!
- Factor code is great.

We want just one piece of code so *one* class.

They have exactly the same interface...

A Simple Example (2/4)

A Parameterized Class

One algebraic vector class:

```
template <unsigned n>
class vec
{
public:
  //...
private:
  float coord_[n];
};
```

"`template <unsigned n>`"
means:
"whatever an `unsigned n`, the
`class vec` is..."

With this sample use:

```
vec<2> v2; // a 2D vector
vec<3> v3; // a 3D one
```

the compiler **automatically generates**:

```
class vec<2>
{
public:
  //...
private:
  float coord_[2];
};
```

and likewise for `vec<3>`.

A Simple Example (3/4)

A Parameterized Class

For the coordinates, what about using `double` instead of `float`?

This version is not sufficient; it lacks a parameter for the coordinate type:

```
template <unsigned n>

class vec
{
public:
  //...
private:
  float coord_[n];
};
```

"Whatever an `unsigned n` and a `type T` (by default `float`), the `class vec` is..." maps to:

```
template <unsigned n,
          typename T = float>
class vec
{
public:
  //...
private:
  T coord_[n];
};
```

A Simple Example (4/4)

A Parameterized Class

With:

```
template <unsigned n,
          typename T = float>
class vec
{
  //...
  T coord_[n];
};
```

and:

```
vec<2> v;
vec<3,double> w;
```

the compiler *generates*:

```
class vec<2,float>
//aka vec<2>
{
  //...
  float coord_[2];
};

class vec<3, double>
{
  //...
  double coord_[3];
};

// do *not* write such a code!
```

A more complete example

```
template <unsigned n,
          typename T>
class vec
{
public:
  using coord_type = T;
  T  operator[](unsigned i) const;
  T& operator[](unsigned i);
  unsigned size() const { return n; }
  //...
private:
  T data_[n];
};
```

- a method is named "`operator[]`"
  so with an object `v`
  we can write `v[i]`
  equiv. to
  `v.operator[](i)`
- this method is overloaded
  (constness is part of
  methods' signature)
- short quiz: what does
  "`v[5] = 1`"
  do?

Outline

Example

named typing and inheritance:

```
struct bar {
  virtual void m() = 0;
};

struct baz : public bar {
  void m() override {
    // ...
  }
};


void foo(bar& arg)
{
  arg.m();
}
```

structural typing and genericity:

```
// concept Bar {
//   void m();
// };

struct baz {
  void m() {
    // ...
  }
};

template <typename Bar>
void foo(Bar& arg)
{
  arg.m();
}
```

Concept

In C++, a `concept` is a **list of requirements**
that a class should fullfil to be a valid input of an algorithm.

Some concepts (1/2)

Find (partly) some concepts behind this program:

```
#include <iostream>
#include <string>
#include <list>

int main()
{
  using list = std::list<std::string>;
  auto l = list{};
  auto s = std::string{};
  while (std::getline(std::cin, s))
    l.push_front(s);
  l.sort();
  for (list::const_iterator i = l.begin(); i != l.end(); ++i)
    std::cout << *i << '\n';
}
```

Some concepts (2/2) – 1st part

Warning: this is pseudo-C++!

```
concept InputIterator // for const_iterator(s)
        // i.e., read-only browsers of container elements
{
  using value_type;

  InputIterator(const InputIterator& rhs);
  InputIterator& operator=(const InputIterator& rhs);

  bool operator!=(const InputIterator& rhs) const;
  const Any& operator*() const;
  InputIterator& operator++();
  // ...
};
```

Some concepts (2/2) – 2nd part

Warning: this is pseudo-C++!

```
concept Container
{
  using value_type;
  using const_iterator; // model of InputIterator
  const_iterator begin() const;
  const_iterator end() const;
  // ...
};

concept FrontInsertionSequence : /*...*/ Container
{
  void push_front(const value_type& elt);
  //...
}
```

Outline

C++ std lib

The *C++ Standard Library*:

- includes most of the historical *Standard Template Library* (STL)
  - authored by Alexander Stepanov
  - including *containers*, *algorithms*, and related tools such as *iterators*
- features much more tools, e.g., `std::string`, `std::ostream`...
- is located in the `std` namespace.

Expressivity

```cpp
#include <iostream>
#include <iterator>
#include <string>
#include <list>

int main()
{
  using std::string; // ok, in this case (very limited scope)
  std::list<string> l;

  using istrit = std::istream_iterator<string>;
  std::copy(istrit(std::cin), istrit(),
            std::back_inserter(l));  // std:: can be removed here

  l.sort();

  std::copy(begin(l), end(l),  // std:: is removed here
            std::ostream_iterator<string>(std::cout, "\n"));
}
```

Hum... Expressivity works when you know the language! Yet, try to figure out what's done...

Outline

Concepts

Key idea: learn concepts, their interface (easy), then you know a lot.

The concepts are *refined* (augmented / extended) from top to bottom when there is a double line.

Container	object that stores elements
Forward Container	elements are arranged in a definite order
Reversible Container	elements are browsable in a reverse order
Random Access Container	elements are retrievable without browsing (amortized constant time access to arbitrary elements)

Container	object that stores elements
Sequence	variable-sized container with elts in a strict linear order
Front Insertion Sequence	first element insertion in amortized constant time
Back Insertion Sequence	last element insertion in amortized constant time

...

Outline

Names without default parameters

`vector<T>`	dynamic array
`list<T>`	doubly-linked list
`deque<T>`	double-ended queue
`stack<T>`	last-in first-out structure (LIFO, *pile* in FR)
`queue<T>`	first-in first-out structure (FIFO, *file* in FR)
`map<K,V>`	sorted dictionary (also AKA associative array)
`set<T>`	(sorted) mathematical set (not a multiset)
`unordered_map<K,V>`	hash-based dictionary
`unordered_set<T>`	hash-based set

Taxonomy

forward containers	all
reversible containers	`vector`, `list`, `deque`
random access containers	`vector`, `deque`
front insertion sequences	`list`, `deque`
back insertion sequences	`vector`, `list`, `deque`
associative containers	`set`-based, `map`-based
unique associative containers	`set`, `map`
multiple associative containers	`multiset`, `multimap`
simple associative containers	`set`-based
pair associative containers	`map`-based

Extras

- `std::stack<T>` and `std::queue<T>` are adaptators (default = built from `std::deque<T>`).
- `std::pair<T1,T2>` is a utility class used to store data in `std::map` it looks like:

```
template <typename T1, typename T2>
struct pair
{
  T1 first;
  T2 second; // ...
};
```

`std::map<std::string,float>::value_type` actually is `std::pair<std::string,float>` for the couple (key, associated value).

Extras

```
template <typename T1,
          typename T2>
struct pair
{
  pair(const T1& t1,
       const T2& t2)
  // ...
  T1 first;
  T2 second;
};

template <typename T1,
          typename T2>
pair<T1, T2>
make_pair(const T1& t1,
          const T2& t2)
{
  return pair<T1, T2>(t1, t2);
}
```

Two equivalent ways to create an obj:

```
auto
  p1 = std::pair<float,int>{16.f, 64},
  p2 = std::make_pair(16.f, 64);
```

New C++-17 way:

```
auto p = std::pair{16.f, 64};
```

No need to explicitly give the parameters.

Common mistakes

What happens?

```
#include <algorithm>
#include <list>
#include <vector>

int main()
{
  auto v = std::vector<int>{};
  for (int i = 0; i < 10; ++i)
    v[i] = i;
  auto l = std::list<int>;
  std::copy(v.begin(), v.end(), l.begin());
}
```

Remember: C++ is just like C = designed to be very fast

(no extra operation performed, if not asked for...)

Oops

What is printed?

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

int main()
{
  auto v1 = std::vector<int>(1, 0);
  std::cout << v1.size() << std::endl;

  auto v2 = std::vector<int>{1, 0};
  std::cout << v2.size() << std::endl;
}
```

Hint: what actually does a constructor? what about for a container?

'explicit'

With:

```
class string {
public:
  string(unsigned len = 0);
  string(unsigned len, char c);
  // ...
};

void foo(const string& s);
```

a possible use is:

```
foo(51);         // works
foo({51, 'c'}); // also works
```

This is *coercion*: one object (or a list {..}) converts to another one having a different type.

Then you can write:

```
// concatenation:
string operator+(const string& lhs,
                 const string& rhs);
// ...

string s;
s = s + 1; // does work! oops...
           // clearly a bug
```

You can prevent coercion:

```
class string {
public:
  explicit string(unsigned len = 0);
  // ...
};
```

Common strange behavior

What is printed?

```
#include <iostream>
#include <map>
#include <string>

int main()
{
  // exercise: explain the {{..}} below
  auto var = std::map<std::string, float>{{"zero", 0.f}};
  var["pi"] = 3.14159;
  std::cout << var["e"] << '\n';
  std::cout << var.size() << '\n';
}
```

Hint: think about how "`operator[](key)`" can work?

Outline

An object that behaves like a function (1/2)

```
struct negate_type
{
  float operator()(float x) const
  {
    return -x;
  }
};

int main()
{
  auto negate = negate_type{};
  auto x = -12.f;
  std::cout << negate(x) << '\n';
}
```

- it looks like a function call
- but it is a method call: `negate.operator()(x)`

An object that behaves like a function (2/2)

```
template <typename F>
float invoke(F f, float x) {
  return f(x);
}

float sqr(float x) { return x * x; }

int main()
{
  float x = -12.f;
  std::cout << invoke(negate_type{}, x) << '\n'
            << invoke(sqr, x) << '\n';
}
```

the function to invoke can be:

- an object
- a regular procedure

An object that behaves like a function (3/2)

```
class sin_ax
{
public:
  sin_ax(unsigned a) : a_(a) {}
  float operator()(float x) const
  {
    return sin(a_ * x);
  }
private:
  const unsigned a_;
};

int main()
{
  auto sin_2x = sin_ax{2};
  auto x = -3.141592f;
  std::cout << sin_2x(x) << " == " << invoke(sin_2x, x) << '\n';
}
```

Use in C++ std lib

```
struct date
{
  // useless due to date{d,m,y}
  /*
  date(unsigned d, unsigned m, unsigned y)
    : day{d}, month{m}, year{y}
  {}
  */

  bool operator<(const date& rhs) const
  {
    return (std::tie(year, month, day)
            < std::tie(rhs.year, rhs.month, rhs.day));
  }

  unsigned day, month, year;
};
```

Use in C++ std lib

```
struct month_first
{
  // the method constness is mandatory
  //    to make temporary functors be referencable...

  bool operator()(const date& lhs, const date& rhs) const
  {
    return (std::tie(lhs.month, lhs.day, lhs.year)
            < std::tie(rhs.month, rhs.day, rhs.year));
  }
};
```

Use in C++ std lib

```
int main()
{
  auto l = std::list<date>
    {
      date{01, 02, 2004},
      date{24, 12, 2002},
      // You don't even need date!
      {27, 02, 2003},
      {28, 02, 2003},
    };

  l.sort();
  l.sort(month_first{});

  auto s = std::set<date, month_first>{};
  // ...
}
```

Outline

Callable Entities

In C++ many things can be "called":

- functions
- references to functions
- function objects
- lambdas
- member function pointers (off topic today)
- etc.

Being "callable" means "behaving like a function".

Outline

Code as Value has Value

- The implementation of `sort` does not depend on the comparison
  It only invokes it
- The implementation of `find_if` does not depend on the predicate
  It only invokes it
- etc.
- It's handy to be able to pass a piece of code as an argument

Lambdas: Predicates for Standard Algorithms

```
template <typename T>
std::ostream& operator<<(std::ostream& o,
                         const std::vector<T>& v)
{
  auto sep = "{";
  for (const auto& e: v) {
    o << sep << e;
    sep = ", ";
  }
  return o << "}";
}

template <typename T>
void print(const T& v)
{
  std::cout << v << '\n';
}
```

Lambdas: Predicates for Standard Algorithms

```
auto is = std::vector<int>{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
random_shuffle(begin(is), end(is));
print(is);

sort(begin(is), end(is));
print(is);

sort(begin(is), end(is), [](int a, int b) { return a > b; });
print(is);

sort(begin(is), end(is),
     [](int a, int b) {
       return std::make_tuple(a % 2, a) < std::make_tuple(b % 2, b);
     });
print(is);
```

```
{6, 0, 3, 5, 7, 8, 4, 1, 2, 9}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
{9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
{0, 2, 4, 6, 8, 1, 3, 5, 7, 9}
```

Lambdas are Functions

```
auto incr = [](int x) { return x + 1; };
std::cout << incr(incr(40)) << '\n';
```

```
42
```

Did you notice we left the return-type to the compiler?

Lambdas are More than Functions

```
auto delta = 2;
auto incr = [delta](int x) { return x + delta; };
std::cout << incr(incr(40)) << '\n';

delta = 3;
std::cout << incr(incr(40)) << '\n';
```

```
44
44
```

- In the square brackets, you list the *captures*
- `[foo]` means keeping a copy

Lambdas are More than Functions

```
auto delta = 2;
auto incr = [&delta](int x) { return x + delta; };
std::cout << incr(incr(40)) << '\n';

delta = 3;
std::cout << incr(incr(40)) << '\n';
```

```
44
46
```

- [&foo] means keeping a reference

Lambdas are More than Functions

```
auto incr = [delta = 1+2*3](int x) { return x + delta; };
std::cout << incr(incr(40)) << '\n';
```

```
54
```

Did you notice we did not have to declare its type?

Lambdas are More than Functions

```
auto incr = [](auto x) { return ++x; };
std::cout << incr(-40) << '\n';
std::cout << incr(2.1415) << '\n';
std::cout << int(incr(uint8_t{255})) << '\n';
std::cout << incr("ZWTF???") << '\n';
```

```
-39
3.1415
0
WTF???
```

- The arguments can be `auto` (magic!!!)

Outline

Implementation of Lambdas

- Can you guess how the compiler translates this?

```
auto month_first
  = [](const date& lhs, const date& rhs)
    {
      return std::tie(lhs.month, lhs.day, lhs.year)
             < std::tie(rhs.month, rhs.day, rhs.year);
    };
```

Implementation of Lambdas

- Can you guess how the compiler translates this?

```
auto month_first
  = [](const date& lhs, const date& rhs)
    {
      return std::tie(lhs.month, lhs.day, lhs.year)
             < std::tie(rhs.month, rhs.day, rhs.year);
    };
```

- A function object!

```
struct month_first_type {
  // actually the struct name is unpredictable
  bool operator()(const date& lhs, const date& rhs) const
  {
    return std::tie(lhs.month, lhs.day, lhs.year)
           < std::tie(rhs.month, rhs.day, rhs.year);
  };
}
auto month_first = month_first_type{};
```

'mutable'

Back with:

```
struct month_first
{
  bool operator()(const date& lhs, const date& rhs) const
  {
    return std::tie(lhs.month, lhs.day, lhs.year)
           < std::tie(rhs.month, rhs.day, rhs.year);
  }
};
```

You want every functor to count the number of calls to `operator()`.
You think to add an attribute...
Yet, the operator is `const` so it cannot modifies the counter!

'mutable'

Back with:

```
struct month_first
{
  bool operator()(const date& lhs, const date& rhs) const
  {
    counter += 1;
    return (std::tie(lhs.month, lhs.day, lhs.year)
            < std::tie(rhs.month, rhs.day, rhs.year));
  }

  mutable unsigned counter = 0;
};
```

`mutable` means modifiable in const methods (that's helpful!)

you do *not* want to use/know the ugly trick: `const_cast<month_first*>(this)->counter += 1;`