C++ Workshop — Day 3 out of 5

Polymorphisms

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)

Namespaces

Namespace

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

The class name is `my::vector`.
It cannot be confused with `std::vector` from the standard library.

- Avoid to "`using my;`", it is evil!
- It is absolutely forbiden at file level in headers.
- It's arguably ok inside a function, or inside a `*.cc` file.

Namespace

```
namespace my
{
  // here no need to use the prefix my::
}
```

A same namespace can be "split" into different files.

Namespaces can be nested:

```
namespace my
{
  namespace inner
  {
    // we are in my::inner:: here!
  }
}
```

The use of namespaces is also for modularity purpose.

Namespace std (1/2)

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>
  {
    // ...
  };
}
```

Namespace std (2/2)

```
namespace std
{
  // an object:
  _IO_ostream_withassign cout;

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

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

Range-based For-loops

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 << ' ';
std::cout << '\n';

// 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&)
```

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); // Parens, not braces!
```

or use some other types of std containers...

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

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

or

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

Polymorphisms

Four different kinds of polymorphism

Polymorphism can be:

	C	C++
coercion	yes	yes
inclusion	no	about 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 of different types.

Coercion

Sample code

```
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 values are involved:

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

Inclusion

Sample code (1/2)

```
// 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(); }
```

Sample code (2/2)

```
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

Sample code

```
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';

auto c = circle{1,2,3};
std::cout << c << '\n';
```

that means that several `operator<<` coexist:

```
// in C++ std lib:
std::ostream& operator<<(std::ostream&, const std::string&);

// in your program:
std::ostream& operator<<(std::ostream&, const circle&);
```

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_;
}
```

Parametric polymorphism

Sample code

```
template <typename T>
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 (namely `foo<int>`) is then compiled for this value / this specific case
- 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; }

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

With parameterization:

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

Parametric polymorphism

Definition

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>` (vector of $\mathbb{R}^3$)

Valuation

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

Mathematical example (1/2)

mathematical function:

$$a \in \mathbb{N},\ f_a : \left\{ \begin{array}{ccc} \mathbb{R} & \rightarrow & \mathbb{R} \\ x & \mapsto & \sin(ax) \end{array} \right.$$

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 $f_a(1)$ is an unknown real	`f<a>`	a description `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 returns a value

Templated classes

A Simple Example (1/4)

A Parameterized Class

original C++ code:

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

if we use `vec<3,float>` somewhere in a program, a first transformation by the compiler gives:

```
class vec<3,float>
{
public:
  using value_type = float;
  //...
private:
  float data_[3];
};
```

How to access a type alias in a class

from outside the "templated world":

```
int main()
{
  using my_vec = vec<2,double>;
  my_vec::value_type d;

  vec<2,bool> bb;
  foo(bb);
}
```

g++ -Wall says
that d is a double in main

from inside this world:

```
template <typename V>
void foo(const V& v)
{
 // 'typename' is mandatory below
  typename V::value_type b;
}
```

and that b is a bool in foo

A simple example (2/4)

```
template <unsigned n, typename T>
class vec
{
public:
  using value_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 access `v[0]`
- this method is overloaded (constness is part of methods' signature)
- short quiz: what does "`v[5] = 1`" do?

A simple example (3/4)

```cpp
// an algorithm

template <typename V>
void bar(V& v,
         typename V::value_type a)
{
  for (auto i = 0u; i < v.size(); ++i)
    v[i] = a;
}
```

```cpp
// in main.cc

int main()
{
  vec<3,float> vv;
  bar(vv, 21);

  std::vector<double> w(7);
  bar(w, 12);
}
```

Not so easy quiz:

- what can be a proper name to bar?
- what is amazing about this algorithm?
- are there limitations or weird things?

A simple example (4/4)

Quiz answers:

- bar can be renamed as `fill`
- this program works with both our class and `std::vector`
- there are problems:
  - `::size()` might not be efficient

    think about some hand-made lists...
  - the `[i]` notation is related to random access containers

    think again about lists...

Duality OO / genericity

Example

named typing and inheritance:

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

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


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

structural typing and genericity:

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

struct baz {
  void m() { /* code */ }
};

template <typename Bar>
void foo(const 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 = 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
{
  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 FrontInsertionSequence
{
  using value_type;
  using const_iterator : InputIterator;

  void push_front(const value_type& elt);
  const_iterator begin() const;
  const_iterator end() const;
  // ...
};
```

A tour of std containers

Introduction

History

The *Standard Template Library* (STL for short):

- is a code library of *containers*, *algorithms*, and related tools such as *iterators*,
- was first written by Alexander Stepanov,
- has been adopted as part of the ANSI/ISO C++ standard,
- is now widely available through several high-quality versions.

The *C++ Standard Library*:

- includes most of STL classes,
- features much more tools, e.g., `std::string`, `std::ostream`...
- is located in the `std` namespace.

Expressivity

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

int main()
{
  std::list<std::string> l;
  std::copy(std::istream_iterator<std::string>(std::cin),
            std::istream_iterator<std::string>(),
            std::back_inserter(l));
  l.sort();
  std::copy(std::begin(l),
            std::end(l),
            std::ostream_iterator<std::string>(std::cout,
                                               "\n"));
}
```

STL really lacks the concept of *range*. C++ 17?

Refinements

Only some containers propose a front insertion method:

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

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

and they are sequences!

Concepts

Concepts (1/3)

Container → Input Iterator

object that stores elements

- - - - - is refined into - - - - -

Forward Container → Forward Iterator

elements are arranged in a definite order

- - - - - is refined into - - - - -

Reversible Container → Bidirectional Iterator

elements are browsable in a reverse order

- - - - - is refined into - - - - -

Random Access Container → Random Access Iterator

elements are retrievable without browsing (amortized constant time access to arbitrary elements)

Concepts (2/3)

Forward Container

- - - - - is refined into - - - - -

Sequence

variable-sized container with elements in a strict linear order

- - - - - is refined into - - - - -

Front Insertion Sequence

first element insertion

in amortized constant time

Back Insertion Sequence

last element insertion

in amortized constant time

Concepts (3/3)

Forward Container

- - - - - is refined into - - - - -

Associative Container

element retrieving is based on key

- - - - - is refined into - - - - -

Simple Associative Container

elements are their own keys

Pair Associative Container

elements are (key,value) pairs

and/or

Unique Associative Container

each key is unique

Multiple Associative Container

several elements can have the same key

Containers

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)
`queue<T>`	first-in first-out structure (FIFO)
`map<K,V>`	sorted dictionary (or associative array)
`set<T>`	sorted mathematical set
`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

Extra

- `stack<T>` and `queue<T>` are adaptators (built from `deque`).
- `std::pair` 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>`

Common mistakes

```
#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());
}
```

Common strange behavior

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

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