“Atelier C++” — Day 3 out of 5

Thierry Géraud

EPITA Research and Development Laboratory (LRDE)

2007

Outline

Outline

1. Polymorphisms
   - Coercion
   - Inclusion
   - Overloading
   - Parametric polymorphism
2. Parametric polymorphism
   - Definition
   - Templated classes
   - Duality OO / genericity
3. A tour of std containers
   - Introduction
   - Concepts
   - Containers

Outline

4 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 polymorph if it accepts input of different types.

Outline

Sample code

```
double sqr(double d) { return d * d; }

void bar()
{
  int i = 3;
  double r = sqr(i);

  float f = 4;
  r = sqr(f);
}
```

Two objects are involved:

- the client one (`i` or `f`), to be converted
- the argument of `sqr` (`d`)

Outline

Sample code (1/2)

```
class scalar { /* abstract class */ }

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

// routine:
scalar& sqr(const scalar& s) { return s * s; }
```

Sample code (2/2)

```
void bar()
{
  my_int i = 1;
  scalar& ii = sqr(i);

  my_float f = 2;
  scalar& ff = sqr(f);
}
```

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

Transtyping is in use here cause `i` and `ii` (resp. `f` and `ff`) represent the *same* object.

Outline

Sample code

```
int    sqr(int i)    { return i * i; }
float  sqr(float f)  { return f * f; }
double sqr(double d) { return d * d; }

void bar()
{
  int i = 1;
  i = sqr(i);  // calls sqr(int)

  float f = 2;
  f = sqr(f);  // calls sqr(float)
}
```

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

Operator overloading

To be able to write:

```
std::string s = "hello world";
std::cout << s << std::endl;

circle c(1,2,3);
std::cout << c << std::endl;
```

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);
  const 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() :
  shape(0.f, 0.f)
{
  this->r_ = 1.f;
}

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

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

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

Outline

Sample code

```
template <typename T>
T sqr(T t)
{
  return t * t;
}

void bar()
{
  int i = 1;
  i = sqr(i);  // calls sqr<int>

  float f = 2;
  f = sqr(f);  // calls sqr<float>
}
```

How it works (1/2)

In `template <typename T> T sqr(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` (precisely `int`)
- a *specific* procedure (namely `foo<int>`) is then compiled for this value
- at last, two different routines are compiled: `foo<int>` and `foo<float>`, and they do not share the same binary code

How it works (2/2)

We end up with overloading; the program actually compiled is:

```
int sqr<int>(int t) { return t * t; }
int sqr<float>(float t) { return t * t; }

void bar() {
  int i = 1;  i = sqr<int>(i);
  float f = 2;  f = sqr<float>(f);
}
```

With parameterization:

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

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., `sqr<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; conversely it is not mandatory for calling routines:
we can write `foo(i)` instead of `foo<int>(i)`

Mathematical example (1/2)

mathematical function:

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

equivalent C++ piece of code:

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

- `x` is an argument ⇔ valued at run-time
- `a` is a parameter ⇔ valued at compile-time-time

Mathematical example (2/2)

f_a	a parametric function	`f<a>`	a description
	$f_a(1)$ cannot be computed		`f(1)` do 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

Outline

A simple example (1/4)

We said a class can be parameterized:

original C++ code:

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

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

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

How to access a typedef in a class

from outside the “templated world”:

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

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

from inside this world:

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

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

and that `b` is a `bool` in `foo`

A simple example (2/4)

```
template <unsigned n, typename T>
class vec
{
public:
  typedef T value_type;
  const 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`"?

A simple example (3/4)

```
// an algorithm

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

```
// in main.cc

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

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

No 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 both with our class than with the `vector` class from the C++ standard library
- there are problems:
  - `::size()` is not always an efficient method
    think about lists...
  - the `[i]` notation is related to random access containers
    think again about lists...

Outline

Example

named typing and inheritance:

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

struct baz : public bar {
  virtual void m() { /* 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 should fullfil a class 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() {
  typedef std::list<std::string> my_list;
  my_list l;
  std::string s;
  while (std::getline(std::cin, s))
    l.push_front(s);
  l.sort();
  for (my_list::const_iterator i = l.begin(); i != l.end(); ++i)
    std::cout << *i << std::endl;
}
```

Some concepts (2/2)

warning: this is pseudo-C++!

```
concept InputIterator
{
  typedef ... value_type;

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

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

```
concept FrontInsertionSequence
{
  typedef ... value_type;
  typedef InputIterator const_iterator;

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

Outline

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(l.begin(),
            l.end(),
            std::ostream_iterator<std::string>(std::cout,
                                               "\n"));
}
```

Namespace

```
namespace std
{
  // a class:

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

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

  // a type alias:
  typedef basic_string<char> string;

  // a procedure:
  istream& operator>>(bool&);
}
```

Refinements

Only some containers propose a front insertion method:

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

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

and they are sequences!

Outline

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 access

in amortized constant time

Back Insertion Sequence

last element access

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

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)
`queue<T>`	first-in first-out structure (FIFO)
`map<Key,Value>`	dictionary (or associative array)
`set<T>`	mathematical 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 <vector>
#include <list>
#include <algorithm>

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

Common strange behavior

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

int main() {
  std::map<std::string,float> var;
  var["pi"] = 3.14159;
  std::cout << var["e"] << std::endl;
  std::cout << var.size() << std::endl;
}
```