C++ Workshop — Day 5 out of 5

Pot Pourri

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)

Without

```
// in circle.hh

class circle : public shape
{
public:
  float r_get() const;
  // ...
private:
  float r_;
};
```

```
// in circle.cc

float circle::r_get() const
{
  return this->r_;
}
```

- `circle::r_get()` has an address at run-time
- the binary code of this method lies in `circle.o`
- calling such a method has a cost at run-time

With inlining

```
// in circle.hh

class circle : public shape
{
public:
  float r_get() const { return r_; }
  // ...
private:
  float r_;
};

// no circle::r_get in circle.cc
```

- including `circle.hh` allows to know the C++ code of `circle::r_get()`
- *a method call can be replaced by its source code*
- thus resulting code can be much more optimized by the compiler...

Using inlining

- There's no excuse for not using accessors
  There is no performance loss
- `inline` considerably improves the opportunities for optimization
- However excessive inlining causes code bloat
- Therefore the compiler is allowed *not* to inline
- Hence `inline` is not (really) about inlining
- Rather it means "multiple copies are ok, just keep one"

Callable Entities

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".

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
```

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
```

- You can set local variables
- Did you notice we did not have to declare its type?

Lambdas are More than Functions

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

```
54
```

- You can set local variables
- 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!!!)

Function object

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';
}
```

```
1.25567e-06 == 1.25567e-06
```

Use in C++ std lib

```
struct date
{
  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
{
  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>{};
  // ...
}
```

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 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{};
```

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 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{};
```

Smart Pointers

There are three important types:

- shared ownership
- no ownership at all
- unique ownership

- `shared_ptr` shares ownership.
  A reference counter is used so that the object managed is deallocated automatically.
- `weak_ptr` is a non-owning pointer.
  It is used to reference an object managed by a `shared_ptr` without adding a reference count. But it knows if the object is still alive.
- `unique_ptr` is a transfer of ownership pointer.

Unique Pointers

Unique Pointers

- Unique pointers accept only *one* owner
- It is impossible to have two unique pointers point to the same object
- This relies on a C++ feature we won't explain now:
  The *move semantics*
- `a = b` copies `b` into `a`
  Both `a` and `b` are alive and "full"
- `a = std::move(b)` moves destructively the content of `b` into `a`
  Both `a` and `b` are alive, but `b` is emptied

Unique Pointers

- Building a unique pointer:

```
auto u = std::make_unique<int>(12);
auto s = std::make_unique<std::string>("hello");
```

- Cannot copy a unique pointer:

```
auto u = std::make_unique<int>(12);
auto u2 = u;  // error: call to implicitly-deleted copy constructor
```

- Can *move* a unique pointer:

```
auto u = std::make_unique<int>(12);
auto u2 = std::move(u);
assert(u == nullptr);
```

Unique Pointers and Shared Pointers

- This is ok, there is a unique owner: `p`.

```
auto p = std::shared_ptr<int>{};
p = std::make_unique<int>(12);
```

The temporary unique pointer gave (moved) its content before dying

- This is not ok, there are two owners: `p` and `u`.

```
auto p = std::shared_ptr<int>{};
auto u = std::make_unique<int>(12);
p = u;  // KO cause *not* unique
```

- This is ok, `u` released its contents, only `p` owns

```
auto p = std::shared_ptr<int>{};
auto u = std::make_unique<int>(12);
p = std::move(u);  // transfer
assert(!u);
```

Weak Pointers

Shared v. Weak (0/3)

`shared_ptr` are nice but:

- we just need some local temporary pointers
- or we can have circular references
  (`a` $\rightarrow$ `b` and `b` $\rightarrow$ `a`)
- or we can also have asymmetrical relationships
- or we want to avoid dangling pointer problems

a `weak_ptr` is great because it does not count...

Weak

Consider

```
auto p = new int(10);
auto p2 = p;
delete p;
// p2 is around...
```

versus:

```
auto p = std::make_shared<int>(10);
auto p2 = std::weak_ptr<int>{p};
p.reset();
// p2 is around but:
assert(p2.expired() == true);
```

Shared v. Weak (1/3)

```
struct page;

struct shape
{
  shape(const std::weak_ptr<page>& p) : p_{p} {}
  virtual ~shape() { std::cout << "a shape dies\n"; }
  std::weak_ptr<page> p_;
};

struct page
{
  ~page() { std::cout << "a page dies\n"; }
  std::vector<std::shared_ptr<shape>> s_;
};

struct circle : public shape
{
  circle(const std::weak_ptr<page>& p) : shape{p} {}
};
```

Shared v. Weak (2/3)

```
int main()
{
  {
    auto p = std::make_shared<page>();

    {
      auto c = std::make_shared<circle>(p);
      p->s_.push_back(c);

    } // c is not deleted here

  } // p is deleted so c is

  std::cout << "the end\n";
}
```

```
a page dies
a shape dies
the end
```

Shared v. Weak (3/3)

Replacing all the `weak_ptr` occurrences by `shared_ptr` gives:

```
the end
```

Within class space

Class classics

In a class we have:

- attributes + methods (encapsulation)
- types aliases (using `using`)

with three different kinds of accessibility (**public**, **protected**, and **private**)

the most important feature is:

a class is a type

About namespaces (1/2)

```
namespace a_namespace_name
{
  a_type a_variable; // an object

  using an_alias_for_this_type
    = a_type;

  return_type a_function()
  {
    // function body
  }

  class a_class
  {
    // class definition
  };
}
```

- namespaces prevent naming conflicts
  `std::vector` cannot be confused by `my::vector`
- a namespace provides a way to categorize entities
  `std::cout` is standardized
- a namespace expresses a module
  precisely a coherent collection of piece of software
- a namespace can be defined in another namespace
  so it is a sub-namespace

About namespaces (2/2)

What we do not have:

- they do not inherit from each other
- they are not types

so what have we left?

- *a tool to handle modularity with names*
  ($\Rightarrow$ artifact: a name disambiguation tool)

languages often provide tools for expressing modularity and...
these tools are not equivalent!

a **package** in Java $\neq$ a **package** in Ada $\neq$ a **namespace** in C++ $\neq$ a **class** in C++ $\neq$ a **module** in Haskell $\neq$...

Adding variables and procedures to class

```
// shape.hh
class shape
{
public:
  shape() : x_{default_x_} {}

  float x_get() const {
    return x_;
  }

  static float default_x();
  // remember:
  // no target => no virtual,
  //              and no const

  // ...
protected:
  float x_, y_;
  static float default_x_;
};
```

```
// in shape.cc

// This variable belongs to
// the class, not an object.
float shape::default_x_ = 5.1f;

// Don't declare static!
float shape::default_x()
{
  return shape::default_x_;
}
```

Sample use

```
int main()
{
  auto* c1 = new circle{1,66,4};
  auto* c2 = new circle{16,6,4};
  std::cout << shape::default_x() << '\n';
  // memory use:
  //    on heap          : 2 * sizeof(circle)
  //    on stack         : 2 * sizeof(void*)
  //    in static memory: 1 * sizeof(float)
}
```

heap (*le tas*, FR) $\neq$ stack (*la pile*, FR)

RTTI

At run-time

```
int main()
{
  shape* s = new circle{1,66,4};
  // ...
}
```

then, at run-time, at the address given by `s`, the object is known as a `circle` since a call `s->print()` is bound to `circle::print`

so dynamic types can be *identified* at run-time!

This is called Run-Time Type Identification (RTTI)

Transtyping upwards and downwards

```
void foo(shape* s) // s is a shape so it can be a rectangle...
{
  // if it is a circle do something specific:

  // trying to downcast
  auto c = dynamic_cast<circle*>(s);

  // if the result is not null then it is a circle
  if (c)
    std::cout << "radius = " << c->r_get() << '\n';
}

int main()
{
  shape* s = new circle{1,66,4}; // upcast = always valid
  foo(s);
}
```

A real application

```
class shape
{
public:
  virtual bool operator==(const shape& rhs) const = 0;
  // ...
protected:
  float x_, y_;
};
```

A real application

```
class circle
  : public shape
{
public:
  bool operator==(const shape& rhs) const override
  {
    if (auto* that = dynamic_cast<const circle*>(&rhs))
      return (   this->x_ == that->x_
              && this->y_ == that->y_
              && this->r_ == that->r_);
    else // rhs is not a circle
      return false;
  }
private:
  float r_;
};
```

So Many Other Things

C++ is a very rich language

- move semantics
- variadic templates
- perfect forwarding
- overloading resolution
- SFINAE
- `std::enable_if_t`
- etc.

For more, see the CXXA course.

C++ is cluttered with historical artifacts

- avoid old idioms
- stay away from dark corners
- learn to love Stack Overflow

> *Within C++, there is a much smaller and cleaner language struggling to get out.*
>
> — [Stroustrup, 1994]

So Many Other Things

C++ is a very rich language

- move semantics
- variadic templates
- perfect forwarding
- overloading resolution
- SFINAE
- `std::enable_if_t`
- etc.

For more, see the CXXA course.

C++ is cluttered with historical artifacts

- avoid old idioms
- stay away from dark corners
- learn to love Stack Overflow

> *Within C++, there is a much smaller and cleaner language struggling to get out.*
>
> — [Stroustrup, 1994]

So Many Other Things

C++ is a very rich language

- move semantics
- variadic templates
- perfect forwarding
- overloading resolution
- SFINAE
- `std::enable_if_t`
- etc.

For more, see the CXXA course.

C++ is cluttered with historical artifacts

- avoid old idioms
- stay away from dark corners
- learn to love Stack Overflow

> “*Within C++, there is a much smaller and cleaner language struggling to get out.*
>
> — [Stroustrup, 1994]

So Many Other Things

C++ is a very rich language

- move semantics
- variadic templates
- perfect forwarding
- overloading resolution
- SFINAE
- `std::enable_if_t`
- etc.

For more, see the CXXA course.

C++ is cluttered with historical artifacts

- avoid old idioms
- stay away from dark corners
- learn to love Stack Overflow

> “*Within C++, there is a much smaller and cleaner language struggling to get out.*
> *And no, that smaller and cleaner language is not Java or C#.*
>
> — [Stroustrup, 1994]

That's All Folks!

- rich language
  - much much more than C
- as efficient as C at run-time
- C++ $\gg$ C + OO
- ...

Dictionary

OO	C++
attribute	data member
method	(virtual) member function
superclass	base class
subclass	derived class
generics	templates

More at `http://www.stroustrup.com/glossary.html`

Part I

Appendix

Bibliography I

Stroustrup, B. (1994).
The Design and Evolution of C++.
ACM Press/Addison-Wesley Publishing Co.

Some Misc Remarks

- `boost::scoped_ptr` is not in **std**...
- `auto_ptr` is a deprecated tiece of ship, so forget it.
- `std::make_unique` arrived in C++ 14.