C++ Workshop — Day 4 out of 5

Pot Pourri

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 23, 2021

Outline

Outline

Outline

Outline

Outline

Outline

Outline

Outline

RVO

RVO = Return Value Optimization

```
struct test
{
  test() {
    std::cout << "ctor\n";
  }
  test(const test&)
  {
    std::cout << "cpy ctor\n";
  }
  ~test() {
    std::cout << "dtor\n";
  }
  void operator=(const test&) = delete;
};
```

```
test foo()
{
  // ...
  return test();
}

int main()
{
  test t = foo();
  // t *looks like* to be
  // constructed by copy...
}
```

gives: ctor dtor

Copying returned objects is avoided!

NRVO

NRVO = Named Return Value Optimization

```
test foo()
{
  test res;    // note that the returned object has a name (!)
  // ...
  return res; // RVO can also work!
}
```

RVO and NRVO are guaranteed :)

and there is no magic (the compiler just transforms your code):

```
// foo compiled with RVO:
void foo(test* ptr_)
{
  test& res = *ptr_;
  // ...
  // so nothing returned
}
```

```
// main compiled with RVO:
int main()
{
  // test t = foo(); is transformed into:
  test t;
  foo(&t); // so no cpy ctor
}
```

Outline

Without inlining

```
// in circle.hh

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

```
// in circle.cc

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

- `circle::get_r()` 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 get_r() const {
    return r_;
  }
  // ...
private:
  float r_;
};

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

- including `circle.hh` allows to know the C++ code of `circle::get_r()`
- *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"

Outline

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

but it is also a name space...

Adding variables and procedures to class

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

  float x() const {
    return x_;
  }

  static float default_x();
  // no target => no virtual,
  //                 and no const
  // ...
protected:
  float x_, y_;
  static float default_x_;
};
```

```cc
// in shape.cc

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

// Don't repeat the 'static' kwd:
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)

About namespaces

- 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

- Avoid to "`using namespace my;`", it is great evil!
- Also avoid to "`using my::vector;`", it is evil!
- It is absolutely forbiden at file level in headers.
- It's arguably OK inside a function, or inside a `*.cc` file.

About namespaces

What we do not have:

- they do not inherit from each other
- they are not types
- they are just about naming

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 $\neq$...

About sub-namespaces

```
namespace my
{
  void foo() {}  // my::foo

  void baz() {}  // my::baz

  namespace sub  // start of my::sub::
  {

    void baz()   // my::sub::baz
    {
    }

    void bar()   // my::sub::bar
    {
      foo(); // --> calls my::foo
      baz(); // --> calls my::sub::baz
    }

  } // end of my::sub::

} // end of my::
```

- Within a namespace, no need to use the suffix.
- The look-up is 1st local then upwards.

Argument-Dependent Lookup (ADL)

It is about **procedure lookup**.
`my::sample` is a *module* with methods and procedures:

```
namespace my {

  class sample
  {
    public:
      void method();
    // ...
  };

  std::ostream&
    operator<<(std::ostream& ostr,
               const sample& s);

  void foo(sample& s);

} // end of my::

void foo(int i);
```

Sample use, outside of `my::`

```
my::sample s;

// these 3 lines are equivalent:

std::cout << s << '\n';
operator<<(std::cout, s) << '\n';
my::operator<<(std::cout, s) << '\n';

// these 2 lines are equivalent:

my::foo(s);
foo(s);
```

Outline

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->get_r() << '\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_;
};
```

Outline

Smart Pointers

There are three important types:

- shared ownership
- no ownership at all
- unique ownership

We have:

- `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.

Outline

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-del'd cpy ctor
```

- 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 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);
```

- 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

Outline

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 vs Weak (1/3)

```
struct page; // page and shape are mutually dependent types

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 vs 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 vs Weak (3/3)

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

```
the end
```

Outline

That's a fact! (1/3)

```
unsigned fact(unsigned n) { // fact as a ``regular'' procedure
  return n == 0 ? 1 : n * fact(n-1);
};

template <unsigned U> struct test {};

int main() {
  const auto five = 5u;
  test<five>{};

  const auto f5 = fact(five); // is f5 known at compile-time?
  test<f5>{};

  unsigned u;
  std::cin >> u;
  const auto fu = fact(u); // can fu be known at compile-time?
}
```

That's a fact! (1/3)

```
unsigned fact(unsigned n) { // fact as a ``regular'' procedure
  return n == 0 ? 1 : n * fact(n-1);
};

template <unsigned U> struct test {};

int main() {
  const auto five = 5u;
  test<five>{};  // OK, compile

  const auto f5 = fact(five); // is f5 known at compile-time?  No (yet maybe!)
  test<f5>{};  // KO, do *not* compile

  unsigned u;
  std::cin >> u;
  const auto fu = fact(u); // can fu be known at compile-time?  NO WAY...
}
```

Ugly things...

Spot the `const`:

```
  const unsigned zero = 0;
  *(unsigned*)(void*)(&zero) = 1;  // OK, compile (!)
  std::cout << zero << '\n';
```

Does it compile? What's the output?
You just do **not** want to know, **never** write this!
Neither that:

```
struct circle {
  void change() { radius = 0.f; }
  float radius = 1.f;
};

int main() {
  const auto& c = circle{};  // radius == 1.f *and* const...
  ((circle*)(&c))->change();
  std::cout << c.radius << '\n';  // now 0.f!
}
```

That's a fact! (2/3)

```
// this is called ``meta-programming''

template <unsigned n>
struct fact    {  enum { value = n * fact<n-1>::value };  };

template <>
struct fact<0> {  enum { value = 1 };  };

template <unsigned U> struct test {};

int main() {
  const unsigned f5 = fact<5>::value;
  // f5 is known at compile-time  ->  you've forced the compiler to compute f5

  test<f5>{};  // now OK, compile
}
```

Seriously, do you really want to write such a code? (no)

It is deprecated since C++11... so forget it!

That's a fact! (3/3)

Now using constexpr:

```
// this routine is versatile: usable both at run-time *and* compile-time
constexpr auto fact(unsigned n) {
  return n == 0 ? 1u : n * fact(n-1);
}

template <unsigned U> struct test {};

int main() {
  constexpr auto f5 = fact(5);  // known at compile-time

  test<f5>{};  // OK again, compile

  unsigned u;
  std::cin >> u;
  auto fu = fact(u);  // calculated at run-time
}
```

constexpr rocks!

Outline

Design Patterns

Design pattern:
an element of architecture, reusable solution to a common problem

Two examples:

- Adapter:
  allows the interface of an existing class to be used as another interface
  e.g., `my::queue<T>` is an adaptation of `my::array<T>`

- Chain of responsibility:
  allows one object to be processed by the appropriate handler
  e.g., `hdlr.read(file)` only works with `pdf_hdlr.read(file)`

Outline

Adapter

Adaptee:

```
namespace my
{
  template <typename T>
  class array
  {
  public:
    unsigned size() const;
    void append(const T& t);
    T operator[](unsigned i) const;
    // ...
  };
}
```

- queue shall not feature `operator[]`, neither append
- queue shall feature push and pop
- yet queue can rely on `array`

Adapter:

```
namespace my
{
  template <typename T>
  class queue {
  public:
    bool is_empty() const {
      return head_ == a_.size();
    }
    void push(const T& t) {
      a_.append(t);
    }
    T pop() {
      return a_[head_++];
    }
  private:
    array<T> a_;
    unsigned head_ = 0;
  };
}
```

Outline

Chain of Responsibility

```
class handler {
public:
  virtual info read(std::ifstream& file) = 0;
  // ...
protected:
  std::unique_ptr<handler> next_; // not shared!
};

class pdf_handler : public handler {
public:
  info read(std::ifstream& file) override {
    info i;
    bool ok = false;
    // try to read...
    if (ok)
      return i;
    else
      return next_->read(file);
  }
  // ...
};
```

the type of `next_` in `pdf_handler` is from another `handler` concrete sub-class (maybe `ps_handler`...)

Disclaimer:

- this is some dummy code
- simplified on purpose
- some more effective designs exist!

Outline

Outline

Reminder: OO vs GP

named typing and inheritance (Object-Orientation):

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

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


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

structural typing and genericity (Generic Programming):

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

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

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

Making “constraints” explicit

named concept:

```
#include <concepts>

template<typename T>
concept Barable = requires(T t)
{
  { t.m() } -> std::same_as<void>;
};

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

explicit constraint:

```
template <Barable B>
void foo(B& arg)
{
  arg.m();
}

void foo2(Barable auto& arg)
{
  arg.m();
}
```

Outline

Modules (1/2)

```
// bar.hh -> bar.o
#include "foo.hh"
#define PI 3.14
void doit();
// ...
```

```
// main.cc -> main.o
#include "baz.hh"
#include "bar.hh"
//...
```

```
// baz.hh -> baz.o
#include "foo.hh"
#include "bar.hh"
#define PI 3.14159265359
void doit();
//...
```

- is `doit()` multiply defined?
  it depends on guards...
- is `foo.hh` included several times?
  yes, in several units (once per each)
- does the inclusion sequence matters?
  WTF, yes!

then "modules" came with C++20 (but are not yet fully supported by every compiler...)

Modules (2/2)

They are for modularity:

- they enable you to express the logical structure of your code
- you can explicitly specify what is exported (`export` kwd)
- you can bundle a few modules into a larger one

with many advantages:

- there is no need to separate interface and implementation
- they are only imported once and literally for free
- it makes no difference in which order you import a module (`import` kwd)

They thus change the way "translation units" are compiled and "objects" linked!

Outline

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.

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

So Many Other Things

"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#."

— Bjarne Stroustrup, 1994.

Boost

Boost is a meta-library, a repo of libraries:

- `http://www.boost.org`
- many libraries are header-only
- many parts have been integrated into C++11

  `https://stackoverflow.com/questions/8851670/which-boost-features-overlap-with-c11`

A Comparison

C

C++

C++ with Boost

That's All Folks!

- rich language
  - much much more than most languages
  - so requires a lot of expertise...
- 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`

Further Readings

By Herb Sutter and Andrei Alexandrescu:

- "C++ Coding Standards: 101 Rules, Guidelines, and Best Practices", Addison-Wesley, 2004.

By Scott Meyers:

- "Effective C++ (3rd ed.): 55 Specific Ways to Improve Your Programs and Designs", Addison-Wesley, 2005.

- "**Effective Modern C++**: 42 Specific Ways to Improve Your Use of C++11 and C++14", Addison-Wesley, **2014**.