“Atelier C++” — Day 5 out of 5

Thierry Géraud

EPITA Research and Development Laboratory (LRDE)

2007

Outline

Outline

Outline

Outline

Outline

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...

and with inlining your program code gets shorter!

Class classics

In a class we have:

- attributes + methods (encapsulation)
- types aliases (thru the `typedef` keyword)

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

  typedef a_type an_alias_for_this_type;

  return_type a_function()
  {
    // function body
  }

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

- a namespace prevents 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

```
// in shape.hh

class shape
{
public:
  shape() {
    this->x_ = shape::default_x_; //...
  }
  float x_get() const { return this->x_; }

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

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

```
// in shape.cc

float shape::default_x_ = 5.1f;
// initialization:
// this variable does not exist thru an object

/*nothing!*/ float shape::default_x()
{
  return shape::default_x_;
}
```

Sample use

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

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

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
  circle* c = dynamic_cast<circle*>(s);

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

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_;
};
```

```
class circle : public shape
{
public:
  virtual bool operator==(const shape& rhs) const
  {
    const circle* rhs_ = dynamic_cast<const circle*>(&rhs);
    if (rhs_ == 0) // rhs is not a circle
      return false;
    return this->x_ == rhs_->x_ and this->y_ == rhs_->y_
           and this->r_ == rhs_->r_;
  }
private:
  float r_;
};
```

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

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

int main()
{
  negate_type negate;
  float x = -12.f;
  std::cout << negate(x) << std::endl;
}
```

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

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

int main()
{
  negate_type negate;
  float x = -12.f;
  std::cout << apply(negate, x) << std::endl;
  std::cout << apply(sqr, x) << std::endl;
}
```

- it looks like a function call
- but it is a method call:

  `negate.operator()(x)`

the function to apply can be:

- an object
- a regular procedure

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

```
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()
{
  sin_ax sin_2x(2);
  float x = -12;
  std::cout << apply(sin_2x, x) << std::endl;
}
```

Use in C++ std lib

```
struct date
{
  unsigned day, month, year;
  date(unsigned day, unsigned month, unsigned year) :
    day(day), month(month), year(year)
  {}
  bool operator<(const date& rhs) const {
    return lexi(this->year, this->month, this->day,
                rhs.year, rhs.month, rhs.day);
  }
};

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

```
int main()
{
  std::list<date> l;
  l.push_back(date(01,02,2004));
  l.push_back(date(24,12,2002));
  l.push_back(date(27,02,2003));
  l.push_back(date(28,02,2003));

  l.sort();
  l.sort(month_first());

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

?

does the live tour continue?

That's it folks!

- rich language
  - much much more then C
- as efficient as C at run-time
- C++ = C + OO
- ...