

# C++ Workshop — Day 4 out of 5

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

(v2015-7-g3c99c09, 2015-12-02 11:02:33 +0100)

- 1 Exceptions
- 2 About constructors et al.
- 3 Live C++ tour

- 1 Exceptions
  - Introduction
  - Syntax
  - A “real” Class as an Exception
- 2 About constructors et al.
- 3 Live C++ tour

## 1 Exceptions

- Introduction
- Syntax
- A “real” Class as an Exception

## 2 About constructors et al.

## 3 Live C++ tour

- Use `assert` during the *development* process
  - to detect (and correct) bugs as early as possible
  - to ease and speed up the process
- In *release* process
  - a program should be robust
    - does not stop if a problem arises
  - so handling errors is not the `assert`-way
  - so you have to write specific code for that

Handling errors correctly means

- **recovering** a *coherent* and *stable* execution state
- having some transversal code in programs  
it is an “*aspect*” of your program

## About C-like error handling:

- the client has to test procedure return values and usually forgets to do so
- when an error is detected, you have to code the “unstacking” (procedure calls) process (“unwinding”) to get to where the error has to be processed...
- that is tedious...

# A simple illustration in C

without error management:

```
void baz() {
    // ...
    // an error happens here
    // ...
}

void bar() {
    // ...
    baz();
    // ...
}

void foo() {
    // ...
    bar(); // erroneous result...
    // ...
}
```

with error management:

```
int baz() {
    // ...
    if (test)
        return -1; // err detected!
    // ...
}

int bar() {
    // ...
    if (baz() == -1)
        return -1; // unstacking...
    // ...
}

void foo() {
    // ...
    if (bar() == -1) {
        // err handling...
    }
    // ...
}
```



- An **exception** is an object that represents the error.
- Such an object lives until the error has been properly processed.
- A routine that detects an error **throws** an exception  
in the previous example, it is the case for `baz`
- A routine in which an error might occur can **catch** this error to do something about it  
in the previous example, it is surely the case of `foo` but also the same for `bar`

## 1 Exceptions

- Introduction
- **Syntax**
- A “real” Class as an Exception

## 2 About constructors et al.

## 3 Live C++ tour

# Error hierarchies

An exception is an object so you (as a client) can define to describe errors:

```
#include <exception>

namespace error
{
    class any : public std::exception {};
    class math : public any {}; // abstract class

    // Concrete classes.
    class overflow : public math {};
    class zero_divide : public math {};
}
```

An `error::zero_divide` *is-an* `error::math`.

# Throwing an exception

```
float div(float x, float y)
{
    // code for handling err in dev mode:
    assert(y != 0);

    // code for handling err in release mode:
    if (y == 0)
        throw error::zero_divide(); // call to a ctor

    // code when everything is OK
    return x / y;
}
```

# Sample behavior

Imagine that program:

```
void baz() {  
    // code 3  
    div(a, b); // here!  
    // code 4  
}  
  
void bar() {  
    // code 2  
    baz();  
    // code 5  
}  
  
void foo() { // called somewhere  
    // code 1  
    bar(); // if not OK, continue  
    // code 6  
}
```

If  $b \neq 0$  in `baz`, execution performs:

- first code 1 to code 3,
- then `div(a, b)` that works fine,
- lastly code 4 to code 6.

If  $b == 0$ , execution should perform

- first code 1 to code 3,
- `div(a, b)` that does *not* work,
- then some specific code to handle this error!
- and finally code 6 (program resumes)

# Handling error

With error handling code in “foo”:

```
void baz() {  
    // code 3  
    div(a, b); // can fail!  
    // code 4  
}  
  
void bar() {  
    // code 2  
    baz();  
    // code 5  
}
```

```
void foo()  
{  
    try {  
        // code 1  
        bar();  
        // code 6  
    }  
    catch (...) {  
        // "... means any exception"  
        std::cerr << "bar aborted!\n";  
    }  
}
```

If no error: code 1 → code 2 → code 3 → div → code 4 → code 5 → code 6

If error: code 1 → code 2 → code 3 → div → err msg

# Recovery from error

```
void bar()
{
    data* ptr = nullptr;
    try {
        // ...
        baz();
        // ...
        ptr = new data; // dyn alloc
        // ...
        baz();
        // ...
    }
    catch (...) {
        delete ptr;
        throw;
    }
}
```

- the 2nd call to baz might fail
- in this example, some action is performed before this call (ptr allocation)
- bar *has to* perform some recovery code if an error occurs during that call (ptr deallocation)
- the `catch` code block is run when an exception has been thrown
- error handling is not completed so the caught exception is thrown again (`instruction throw;`); the error is still alive...

## Handling error (2/2)

With a more complete error handling code:

```
void baz() {
    try {
        // code 3
        div(a, b); // can fail!
        // code 4
    }
    // code Z: catch, fix, and throw
}

void bar() {
    try {
        // code 2
        baz();
        // code 5
    }
    // code R: catch, fix, and throw
}
```

```
void foo()
{
    try {
        // code 1
        bar();
        // code 6
    }
    catch (...) {
        // "..." means "any exception"
        std::cerr << "bar aborted!\n";
    }
}
```



# Selecting errors to handle

```
void foo() {
    try {
        // ...
    }
    catch (error::zero_divide) {
        // handles such error
    }
    catch (error::math) {
        // handles other math errors
    }
    catch (error::any) {
        // handles non-math client errors
    }
    catch (std::bad_alloc) {
        // handles an allocation ('new') that failed
    }
    catch (...) {
        // handles all remaining kinds of errors
    }
}
```

- `catch` clauses are inspected in the order they are listed
- the appropriate `catch` clause is selected from the error type
- the corresponding code is run

# A “real” Class as an Exception

## 1 Exceptions

- Introduction
- Syntax
- A “real” Class as an Exception

## 2 About constructors et al.

## 3 Live C++ tour

# The “real” Class

```
namespace error
{
    class problem : public any
    {
    public :
        problem(const std::string& fname,
                unsigned line,
                const std::string& msg);
        unsigned line() const;
        // ...
    private :
        std::string fname_;
        unsigned line_;
        std::string msg_;
    };
}
```

```
// in namespace error::
std::ostream&
operator<<(std::ostream& o,
           const problem& p)
{
    o << "err in " << p.fname()
      << "at line " << p.line()
      << ": " << p.msg();
    return o;
}
```

# Using the exception object

An exception is thrown  
an object is constructed

```
void parse(const std::string& s)
{
    // ...
    throw error::problem(__FILE__,
                          __LINE__,
                          "ICE!");
    // ...
}
```

The exception is caught  
the object is inspected

```
void compile()
{
    try {
        // parse something...
    }
    catch(error::problem& pb) {
        std::cerr << pb << '\n';
        // pb is a regular object!
    }
};
```

# About constructors et al.

## 1 Exceptions

## 2 About constructors et al.

- C++ is like C
- C++ idioms
- C++ is just like C: dangerous!
- Optimizations

## 3 Live C++ tour

# C++ is like C

## 1 Exceptions

## 2 About constructors et al.

- C++ is like C
- C++ idioms
- C++ is just like C: dangerous!
- Optimizations

## 3 Live C++ tour

## C behavior (1/3)

```
struct foo
{
    int i;
    float* ptr;
};

int main()
{
    foo* C = malloc(sizeof(foo));
    foo a, aa; // constructions
    foo b = a; // copy construction
    // but:
    aa = a;    // assignment
} // a, aa, and b die
// C also dies (niark!)
// so who does not?
```

```
void bar(foo d)
{
    // ...
} // d dies

foo baz()
{
    foo e;
    // ...
    return e; // e is copied
                // while baz returns
} // e dies

int main()
{
    foo f; // construction
    bar(f); // d is copied from f
                // when bar is called
} // f dies
```

## C behavior (2/3)

with:

```
struct foo { int i; float* ptr; };

int main() {
    foo* C = malloc(sizeof(foo));
    foo a, aa; // constructions
    foo b = a; // copy construction
    aa = a;   // assignment
}
```

we have:

<i>expression</i>	<i>value</i>
<code>C-&gt;i</code> and <code>C-&gt;ptr</code>	undefined
<code>a.i</code> and <code>a.ptr</code>	undefined
<code>b.i</code> and <code>b.ptr</code>	resp. equal to <code>a.i</code> and <code>a.ptr</code>
<code>aa.i</code> and <code>aa.ptr</code>	likewise



# C behavior (3/3)

this C code:

```
struct bar { /*...*/};

struct foo {
    bar b; int i; float* ptr;
};
```

is equivalent to the C++ code:

```
class foo {
public:
    foo();
    foo(const foo& rhs);
    foo& operator=(const foo& rhs);
    ~foo();
public: // no hiding!
    bar b; int i; float* ptr;
};
```

```
foo::foo()
: b{} // calls bar::bar()
{} // to construct this->b

foo::foo(const foo& rhs)
: b{rhs.b} // calls bar::bar(const bar&)
// to cpy construct this->b
, i{rhs.i} // integer cpy
, ptr{rhs.ptr} // pointer cpy
{}

foo& foo::operator=(const foo& rhs) {
    if (&rhs != this) {
        b = rhs.b;
        i = rhs.i;
        ptr = rhs.ptr;
    }
    return *this;
}

foo::~~foo()
{} // automatically calls bar::~~bar()
// on this->b so this->b dies
```

## 1 Exceptions

## 2 About constructors et al.

- C++ is like C
- **C++ idioms**
- C++ is just like C: dangerous!
- Optimizations

## 3 Live C++ tour

---

```
return_t type::method(/* args */)
```

---

**a regular method**

```
type::type()
```

```
type::type(const type&)
```

```
type& type::operator=(const type&)
```

```
type::~~type()
```

**special methods:**

default constructor

copy constructor

assignment operator

destructor

(and then you die)

---

when the programmer does not code one of these special methods, the compiler (in most cases...) adds this method following the C behavior.

# Special methods and inheritance

```
class base // are belong to us
{
public:
    base();
    base(int b);
    base(const base& rhs);
    base& operator=(const base& rhs);
    virtual ~base();
protected:
    int b_; /*...*/
};

class derived : public base
{
public:
    derived();
    derived(int b, float d);
    derived(const derived& rhs);
    derived& operator=(const derived& rhs);
    virtual ~derived();
private:
    float d_; /*...*/
};
```

```
derived::derived()
    : base(), d_(0) /*...
{ // allocate resource when needed
}

derived::derived(int b, float d)
    : base(b /*...*/), d_(d) /*...
{ // allocate resource when needed
}

derived::derived(const derived& rhs)
    : base(rhs), d_(rhs.d_) /*...
{ // allocate resource when needed
}

derived& derived::operator=(const derived& rhs) {
    if (&rhs != this) {
        this->base::operator=(rhs);
        this->d_ = rhs.d_; /*...
    }
    return *this;
}

derived::~~derived()
{ // resource deallocation when needed
  // warning: do NOT call base::~~base()
}
```

please do not think, just do like that (!)

- please *strictly* follow the idioms given in the previous slide
- `this->b_`, as an attribute of base, is not processed in the special methods of derived
- each constructor of derived first calls the appropriate constructor of base
- if a class has a `virtual` method, its destructor shall be tagged `virtual`
- in the destructor body (there is one per class), do *not* call the destructor of base classes
- in constructors and destructor bodies, do *not* call on `this` any `virtual` method from the same hierarchy

# C++ is just like C: dangerous!

## 1 Exceptions

## 2 About constructors et al.

- C++ is like C
- C++ idioms
- **C++ is just like C: dangerous!**
- Optimizations

## 3 Live C++ tour

# What's the problem?

```
class easy
{
public:
    easy();
    ~easy();
private:
    float* ptr_;
};

easy::easy()
{ // allocate a resource so...
    this->ptr_ = new float;
}

easy::~~easy()
{ // ...deallocate it!
    delete this->ptr_;
    this->ptr_ = nullptr; // real safety!
}
```

```
void naive(easy bug)
{
    // nothing done so ok!
}

int main()
{
    easy run;
    naive(run);
}

// compiles but fails at run-time!!!
```

either:

```
class easy
{
public:

    easy(); // defined in .cc
    ~easy(); // defined in .cc

private:
    float* ptr_;

    // declarations only:
    easy(const easy&);
    void operator=(const easy&);
    // not defined in .cc
};
```

or:

```
class easy
{
public:

    // defined in .cc
    easy();
    ~easy();
    easy(const easy& rhs);
    easy& operator=(const easy& rhs);
    // and with great care!

private:
    float* ptr_;
};
```



# Cool C++ 11 features

explicitly forbid cpy ctor, op=

```
class easy
{
public:

    easy(); // defined in .cc
    ~easy(); // defined in .cc

    easy(const easy&) = delete;
    void operator=(const easy&)
        = delete;

private:
    float* ptr_;
};
```

explicitly say:

“provide a default impl”

```
class easyII
{
public:

    easyII() = default;
    easyII(const easyII&) = default;
    // ...
};
```

## 1 Exceptions

## 2 About constructors et al.

- C++ is like C
- C++ idioms
- C++ is just like C: dangerous!
- **Optimizations**

## 3 Live C++ tour

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 = Named Return Value Optimization

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

auto ->

FIXME: add an example of: `auto foo(...) -> type`

- 1 Exceptions
- 2 About constructors et al.
- 3 Live C++ tour**

# Objectives

- classes
  - ⇒ encapsulation (attributes + methods) and information hiding
- a class hierarchy
  - ⇒ inheritance with an abstract class and concrete sub-classes
- special methods (ctors, cpy ctor, dtor, op=)
- design of class interfaces
- use of `std::` tools:
  - output stream
  - a container
  - iterations
- everything in a namespace

- several kinds of shapes
- a shape is in a page
- a page can be copied; a shape can be cloned
- every object is printable
- an exception arises when calling `circle::r_set(-1)`



# Now code

...