C++ Workshop — Day 1 out of 5

Object

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)

1. A Better C
2. My First C++ class
3. Low-Level Memory Management

A Better C

1. A Better C
   - Handy Tools
   - References
   - C++ I/O Streams
2. My First C++ class
3. Low-Level Memory Management

C++

- C++ inherits from C
- A blessing
- And a curse
- Learning "C++ as a better C" might not be the best path
- Yet…

Handy Tools

1. A Better C
   - Handy Tools
   - References
   - C++ I/O Streams
2. My First C++ class
3. Low-Level Memory Management

auto

```cpp
auto i = 0; // i is an int.
auto u = 0u; // u is an unsigned int.
auto s = make_string("foo") // s is a string
auto it = begin(s); // it has a really ugly type.

auto x;  // this is invalid, forces you to provide
         // the initial value.
```

auto

- `auto` is essential when types are unknown
  We'll see with `template`s that it's possible not to know,
  and with lambdas that it's *impossible* to know.
- `auto` is handy when types are really long
  '`typename std::vector<int>::const_iterator i = begin(v);`' vs.
  '`auto i = begin(v);`'.
- `auto` is robust to minor changes
  '`typename std::vector<int>::const_iterator i = begin(v);`', fails
  to compile when `v` is turned into a `std::list`.
  '`auto i = begin(v);`' applies to both.
- `auto` avoids stuttering code
  `std::vector<std::string>* v = new std::vector<std::string>()`
  vs. `auto v = new std::vector<std::string>()`.
- `auto` is nice looking when used consistently
  Nice left-alignment.

auto

- `auto` is essential when types are unknown
  We'll see with `template`s that it's possible not to know,
  and with lambdas that it's *impossible* to know.
- `auto` is handy when types are really long
  '`typename std::vector<int>::const_iterator i = begin(v);`' vs.
  '`auto i = begin(v);`'.
- `auto` is robust to minor changes
  '`typename std::vector<int>::const_iterator i = begin(v);`', fails to compile when `v` is turned into a `std::list`.
  '`auto i = begin(v);`' applies to both.
- `auto` avoids stuttering code
  `std::vector<std::string>* v = new std::vector<std::string>()`
  vs. `auto v = new std::vector<std::string>()`.
- `auto` is nice looking when used consistently
  Nice left-alignment.

auto

- `auto` is essential when types are unknown
  We'll see with `template`s that it's possible not to know,
  and with lambdas that it's *impossible* to know.
- `auto` is handy when types are really long
  '`typename std::vector<int>::const_iterator i = begin(v);`' vs.
  '`auto i = begin(v);`'.
- `auto` is robust to minor changes
  '`typename std::vector<int>::const_iterator i = begin(v);`', fails to compile when `v` is turned into a `std::list`.
  '`auto i = begin(v);`' applies to both.
- `auto` avoids stuttering code
  `std::vector<std::string>* v = new std::vector<std::string>()`
  vs. `auto v = new std::vector<std::string>()`.
- `auto` is nice looking when used consistently
  Nice left-alignment.

auto

- `auto` is essential when types are unknown
  We'll see with `template`s that it's possible not to know,
  and with lambdas that it's *impossible* to know.
- `auto` is handy when types are really long
  '`typename std::vector<int>::const_iterator i = begin(v);`' vs.
  '`auto i = begin(v);`'.
- `auto` is robust to minor changes
  '`typename std::vector<int>::const_iterator i = begin(v);`', fails to compile when `v` is turned into a `std::list`.
  '`auto i = begin(v);`' applies to both.
- `auto` avoids stuttering code
  `std::vector<std::string>* v = new std::vector<std::string>()`
  vs. `auto v = new std::vector<std::string>()`.
- `auto` is nice looking when used consistently
  Nice left-alignment.

auto

- `auto` is essential when types are unknown
  We'll see with `templates` that it's possible not to know,
  and with lambdas that it's *impossible* to know.
- `auto` is handy when types are really long
  '`typename std::vector<int>::const_iterator i = begin(v);`' vs.
  '`auto i = begin(v);`'.
- `auto` is robust to minor changes
  '`typename std::vector<int>::const_iterator i = begin(v);`', fails
  to compile when `v` is turned into a `std::list`.
  '`auto i = begin(v);`' applies to both.
- `auto` avoids stuttering code
  `std::vector<std::string>* v = new std::vector<std::string>()`
  vs. `auto v = new std::vector<std::string>()`.
- `auto` is nice looking when used consistently
  Nice left-alignment.

A Better typedef: using

- The syntax of typedef is really dubious...

```
typedef unsigned int uint;
unsigned typedef int uint;
unsigned int typedef uint;
int typedef unsigned uint;
```

- Its legibility too...

```
typedef int arr[];
int typedef arr[];
typedef int (main)(int argc, const char* argv[]);
int typedef (main)(int argc, const char* argv[]);
```

- using is much saner:

```
using uint = unsigned int;
using arr  = int[];
using main = auto (int argc, const char* argv[]) -> int;
```

A Better typedef: using

- The syntax of typedef is really dubious. . .

```
typedef unsigned int uint;
unsigned typedef int uint;
unsigned int typedef uint;
int typedef unsigned uint;
```

- Its legibility too. . .

```
typedef int arr[];
int typedef arr[];
typedef int (main)(int argc, const char* argv[]);
int typedef (main)(int argc, const char* argv[]);
```

- using is much saner:

```
using uint = unsigned int;
using arr  = int[];
using main = auto (int argc, const char* argv[]) -> int;
```

A Better typedef: using

- The syntax of typedef is really dubious...

```
typedef unsigned int uint;
unsigned typedef int uint;
unsigned int typedef uint;
int typedef unsigned uint;
```

- Its legibility too...

```
typedef int arr[];
int typedef arr[];
typedef int (main)(int argc, const char* argv[]);
int typedef (main)(int argc, const char* argv[]);
```

- using is much saner:

```
using uint = unsigned int;
using arr  = int[];
using main = auto (int argc, const char* argv[]) -> int;
```

cdecl

Have fun with `man 1 cdecl` or `http://www.cdecl.org`:

```
char (*(*(* const x[3])())[5])(int)
=> "declare x as array 3 of const pointer to function returning
    pointer to array 5 of pointer to function (int) returning char"

"declare bar as volatile pointer to array 64 of const int"
=> const int (* volatile bar)[64]
```

Argument Default Values

It is possible to define default values for arguments:

```
int succ(int i, int delta = 1)
{
  return i + delta;
}

int one = 1;
int two = succ(one);
int ten = succ(two, 8);
```

Applies everywhere, except, weirdly, in lambdas (day 5).

1 A Better C
- Handy Tools
- References
- C++ I/O Streams

2 My First C++ class

3 Low-Level Memory Management

What a reference is

A **reference** is:

- a non null constant pointer with a non-pointer syntax
- a variable that represents *an* object
  - this variable has to be initialized with an object
    since every constant should be initialized
  - this variable will always represent this object
    do *not* imagine that the reference will point to another object

A couple of exercises

```cpp
int i = 1;
int& j = i;
j = 2;
i == 2; // true or false?
```

```cpp
int i = 3, j = 4;
int& k = i;
k = j;
j = 5;
// i == ?  k == ?
```

Soluce (for C coder)

```c
int i = 1;
int *const p_j = &i;
*p_j = 2;
bool b = i == 2; /* true */
```

```c
int i = 3, j = 4;
int *const p_k = &i;
*p_k = j;
j = 5;
/*  i == 4  *p_k == 4  */
```

Soluce (for C++ coder)

```cpp
int i = 1;
// 'j' is 'i'
i = 2;
bool b = i == 2; // true
```

```cpp
int i = 3, j = 4;
// 'k' is 'i'
i = j;
j = 5;
// i == 4  k == 4
```

Another example (swapping)

```
// C swap
void int_swap(int* pi1, int* pi2)
{
  int tmp = *pi1;
  *pi1 = *pi2;
  *pi2 = tmp;
}

void foo()
{
  int i = 5, j = 1;
  swap(&i, &j); // pointers
}
```

```
// C++ swap
void swap(int& i1, int& i2)
{
  int tmp = i1;
  i1 = i2;
  i2 = tmp;
}

void foo()
{
  int i = 5, j = 1;
  swap(i, j); // references
}
```

Reference best use

Pick one of these:

```
void foo(circle c) {        // copy the whole object
  // code
}
void foo(const circle& c) { // avoid copy (faster)
  // same code
}
```

Pick one of these:

```
void foo(circle* p_c) {  // so modifies its input
  // code with ``p_c->'', beware of nullptr
}

void foo(circle& c) {    // the same
  // code with ``c.''
}
```

Hints for beginners

Avoid:

```
type& routine() {
  type* p = // dynamic allocation
  // ...
  return *p;
}
```

```
class a_class {
  // ...
  type& ref_;
};
```

Prefer:

```
type* routine() {
  type* p = // dynamic allocation
  // ...
  return p;
}
```

```
class a_class {
  // ...
  type* ptr_;
};
```

With C++ 11, you'd prefer '`shared_ptr<type>`' (or equiv) over '`type*`'.

Back to auto

- `auto` is a placeholder for a “basic” type
- It will hold a (deep) copy
- But you may qualify it with `const`, $*$, and &

```
// jumbo instances are large.
jumbo j1 = jumbo{10};
jumbo j2 = j1;         // copy
jumbo& j3 = j1;        // RW alias
const jumbo& j4 = j1;// R alias
```

```
// jumbo instances are large.
auto j1 = jumbo{10};
auto j2 = j1;          // copy
auto& j3 = j1;         // RW alias
const auto& j4 = j1;// R alias
```

1. A Better C
   - Handy Tools
   - References
   - C++ I/O Streams
2. My First C++ class
3. Low-Level Memory Management

C++ Streams are Typed

```
#include <iostream>
// ...
std::cout << "Foo" // const char[4]
          << true  // bool
          << 23    // int
          << '\n'; // char
```

- Less flexible than `printf`
- But there are "IO manipulators" to control formatting
- Type safe, contrary to `printf`
  no possible mismatch between format and argument
- Extensible to user types

C to C++ translator

	C	C++
inclusion	#include <stdio.h>	#include <iostream>
input type	FILE*	std::istream&
input file type	FILE*	std::ifstream
inputting	(many ways)	use of >>
output type	FILE*	std::ostream&
output file type	FILE*	std::ofstream
outputting	(many ways)	use of <<
standard output	stdout	std::cout
standard error	stderr	std::cerr
end of line	'\n'	'\n' (not std::endl!)
char string type	char*	std::string
string stream	#include <stdio.h>	#include <sstream>
	use of sscanf and sprintf	use of >> and <<

1 A Better C

2 My First C++ class
- Introducing attributes and methods
- Heart of the “O” Paradigm
- Lifetime Management
  - Constructors
  - Destructor
  - RAII
- Output Streamable

3 Low-Level Memory Management

Foreword

Though frustrating for people who already “know” OO,
we will adopt a step-by-step introduction of object-oriented features to a definition of `circle`.

Introducing attributes and methods

1. A Better C
2. My First C++ class
   - Introducing attributes and methods
   - Heart of the "O" Paradigm
   - Lifetime Management
   - Output Streamable
3. Low-Level Memory Management

Remember C

```
typedef struct circle circle;
struct circle
{
  float x, y, r;
};

void circle_translate(circle* c, float dx, float dy);
void circle_print(const circle* c);
```

Procedural paradigm

A translation of the common assertion:

$$\textit{program} = \textit{data structures} + \textit{algorithms}$$

Here, we have a data structure + a couple of procedures.

Towards another paradigm

Note that the procedures' argument **c** is in both cases the **target** of the algorithms:

- what is translated?
  answer: the circle **c**
- what is printed?
  answer: the circle **c**
- what really are **dx** and **dy**?
  answer: auxiliary data to perform translation

(Raw) Translation into C++ (1/3)

```
// circle.hh
#ifndef CIRCLE_HH
# define CIRCLE_HH

struct circle
{
  void translate(float dx, float dy);
  void print() const;

  float x, y, r;
};

#endif
```

Most compilers support: `#pragma once`
so you do not have to write these guards (error prone).

Terminology

Encapsulation: action of *grouping* data and algorithms into a structure.

Some terminology:

C coder	C++ coder	OO	meaning
structure field[1]	member	**attribute**	"data"
function[2]	member function	**method**	"algorithm"

[1] a "regular" field like **r** for **circle**
[2] a routine with a clearly identified target.

Changes

In header file:

C	`typedef struct ... circle;`
C++	`struct circle ... ;`
C	`void circle_translate(circle* c, float dx, float dy);`
C++	`struct circle ... void translate(float dx, float dy);`
C	`void circle_print(const circle* c);`
C++	`struct circle ... void print() const; ... ;`

Translation into C++ (2/3)

Sample use:

```
int main()
{
  circle* c = // ...
  c->translate(4, 5);
  c->print();

  circle k;
  // ...
  k.translate(4, 5);
  k.print();
}
```

Calling a method is just like accessing a structure field.

`*c` and `k` are the **targets** of method calls.

Translation into C++ (3/3)

```
// file circle.cc

#include "circle.hh"
#include <cassert>

void circle::translate(float dx, float dy)
{
  assert(0.f < this->r);
  this->x += dx;
  this->y += dy;
}

void circle::print() const
{
  assert(0.f < this->r);
  std::cout << "(x="  << this->x
            << ", y=" << this->y
            << ", r=" << this->r)
            << ")";
}
```

Changes

In source file:

C	`void circle_translate(circle* c, float dx, float dy) {.`
C++	`void circle::translate(float dx, float dy) {...}`
C	`void circle_print(const circle* c) {...}`
C++	`void circle::print() const {...}`
C	`c->`
C++	`this->`

“this”

The keyword `this` is a pointer to the target.

code	in “circle::print”
`circle* c1 = // ...`	
`circle* c2 = // ...`	
`c1->print();`	`this == c1`
`c2->print();`	`this == c2`
`circle k; // ...`	
`k.print();`	`this == &k`

About "this"

`this` is only valid in method code.

`this` is a constant pointer:
You cannot assign to it: "`this = //...`"

type of "`this`"	in
`const type *const`	`type::method(args) const`
`type *const`	`type::method(args)`

The writing "`this->something`" can be simplified into "`something`" when there is no ambiguity.

About method constness

A method is tagged "`const`" if it does not modify the target.

Corollaries:

- you cannot modify `this` in "`circle::print() const`"
  "`this->r = 0.f;`" would not compile
- you cannot call a non-const method on a const instance:

```
const circle* c = //...
c->translate(1, 2);
```

  does not compile
- you cannot call a non-const method on `this` in a const method.

Nota bene: on a non-const instance, you can call both const and non-const methods.

Heart of the “O” Paradigm

1 A Better C

2 My First C++ class

- Introducing attributes and methods
- Heart of the “O” Paradigm
- Lifetime Management
- Output Streamable

3 Low-Level Memory Management

Hiding information

We said that we cannot prevent the programmer from breaking invariants

- because data are not protected
- because writing "`c->r = -1;`" is valid C++

⇒ A client should be restricted to access only to some part of a structure.

Two keywords:

- `public` means "accessible from everybody"
- `private` means "only accessible from methods of the same structure"

Class

A **class** is a structure using both encapsulation and information hiding.

Re-writing:

```
class circle
{
public:
  //...
  void translate(float dx, float dy);
  void print() const;
private:
  float x_, y_, r_;
};
```

Definition and hints

The **interface** of a class is its public part.

Some hints:

- the interface contains only methods
- attributes are private
- the suffix "_" qualifies non-public names.

Object

An **object** is an instance of a class.

So:

- we can call methods on it
- it hides some information

Further

At this point, we do not know:

- how to initialize an object
- how to access information
- how to modify a particular attribute

Constructor (1/2)

A **constructor** is a particular kind of methods that allows for instantiating objects with proper *initialization* for their attributes.

Syntax:

- a constructor is named after its class
- it is not constant
- it has no return

Constructor (2/2)

```cpp
// in circle.hh

class circle
{
public:
  circle(float x, float y,
         float r);
  //...
};
```

```cpp
// in circle.cc

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

Accessors and mutators (1/2)

An **accessor** is a constant method that gives a read-only access to a class attribute.
A **mutator** is a (non-constant) method that allows for modifying a class attribute value.

```
class circle
{
public:
  //...
  float r_get() const;
  void  r_set(float r);
  //...
};
```

Accessors and mutators (2/2)

In source file:

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

```
void circle::r_set(float r)
{
  assert(r > 0.f);
  this->r_ = r;
}
```

Ensures you that the radius remains positive.

1. A Better C
2. My First C++ class
   - Introducing attributes and methods
   - Heart of the “O” Paradigm
   - Lifetime Management
     - Constructors
     - Destructor
     - RAII
   - Output Streamable
3. Low-Level Memory Management

Lifetime Management

- Objects must remain in a consistent state
  Their *invariants* must be established and preserved
- More often than not, objects hold resources
  Allocated memory, file descriptors, system locks, etc.
- So we need a means to initialize an object
  Set up in the invariants, possibly acquire resources
- And a means to return these resources
  Release memory, close file descriptors, etc.

This is *lifetime management*: birth and death of objects.
Or rather, *construction* and *destruction*.

Constructors

1 A Better C

2 My First C++ class
- Introducing attributes and methods
- Heart of the “O” Paradigm
- Lifetime Management
  - Constructors
  - Destructor
  - RAII
- Output Streamable

3 Low-Level Memory Management

Constructor

```
class circle
{
public:
  // Declare the constructor.
  circle(float x, float y, float r);
private:
  float x_, y_, r_;
};

// Implement it.
circle::circle(float x, float y,
               float r)
{
  // Ensure invariants.
  assert(0 < r);
  x_ = x;
  y_ = y;
  r_ = r;
}
```

```
int main ()
{
  // Use it.
  circle c1(0, 0, 1);
  circle c2{0, 0, 1};
  circle c3 = {0, 0, 1};
  // Preferred.
  auto c4 = circle{0, 0, 1};
}
```

Constructor: Initializers

```
circle::circle(float x, float y,
               float r)
{
  // Invalid state,
  // random values...
  assert(0 < r);
  // Invalid state...
  x_ = x;
  // Invalid state...
  y_ = y;
  // Invalid state...
  r_ = r;
  // Valid state!
}
```

```
circle::circle(float x, float y,
               float r)
  : x_{x}, y_{y}, r_{r}
//: x_(x), y_(y), r_(r)
{
  // Possibly invalid object, but
  // well defined state.
  assert(0 < r);
  // Valid object.
}
```

Constructor: Initializers

```cpp
circle::circle(float x, float y,
               float r)
{
  // Invalid state,
  // random values...
  assert(0 < r);
  // Invalid state...
  x_ = x;
  // Invalid state...
  y_ = y;
  // Invalid state...
  r_ = r;
  // Valid state!
}
```

```cpp
circle::circle(float x, float y,
               float r)
  : x_{x}, y_{y}, r_{r}
//: x_(x), y_(y), r_(r)
{
  // Possibly invalid object, but
  // well defined state.
  assert(0 < r);
  // Valid object.
}
```

Constructor: Delegation (C++ 11)

```
// General case.
circle::circle(float x, float y,
               float r)
  : x_{x}, y_{y}, r_{r}
{
  assert(0 < r);
}

// Centered cycle.
circle::circle(float r)
  : x_{0}, y_{0}, r_{r}
{
  // There's a bug here!
}

// Unit circle.
circle::circle()
  : x_{0}, y_{0}, r_{1}
{}
```

Prefer this version:

```
circle::circle(float x, float y,
               float r)
  : x_{x}, y_{y}, r_{r}
{
  assert(0 < r); // always tested!
}

circle::circle(float r)
  : circle{0, 0, r}
{}

circle::circle()
  : circle{1}
{}
```

Constructor: Default Member Values (C++ 11)

```
class circle
{
public:
  circle(float x, float y, float r)
    : x_{x}, y_{y}, r_{r}
  {}

  circle(float r)
    : circle{0, 0, r}
  {}

  circle()
    : circle{1}
  {}

private:
  float x_, y_, r_;
};
```

```
class circle
{
public:
  // Actually useless if you use
  // the braces: circle{...}.
  circle(float x, float y, float r)
    : x_{x}, y_{y}, r_{r}
  {}

  circle(float r)
    : r_{r}
  {}

  circle() = default;

private:
  float x_ = 0, y_ = 0, r_ = 1;
};
```

Destructor

1. A Better C
2. My First C++ class
   - Introducing attributes and methods
   - Heart of the "O" Paradigm
   - Lifetime Management
     - Constructors
     - Destructor
     - RAII
   - Output Streamable
3. Low-Level Memory Management

Destructor

- Constructor are not/cannot be a "method", why?
- There can be many constructors, why?
- There can be only one destructor, why?
- Destructors can be "methods", why?

Destructor

- Constructor are not/cannot be a "method", why?
- There can be many constructors, why?
- There can be only one destructor, why?
- Destructors can be "methods", why?

Destructor

- Constructor are not/cannot be a “method”, why?
- There can be many constructors, why?
- There can be only one destructor, why?
- Destructors can be “methods”, why?

Destructor

- Constructor are not/cannot be a "method", why?
- There can be many constructors, why?
- There can be only one destructor, why?
- Destructors can be "methods", why?

Destruction in Action

```
#include <iostream>

class foo {
public:
  foo(int v) : val_(v) {
    std::cerr << " foo::foo(" << val_ << ")\n";
  }
  ~foo() {
    std::cerr << "foo::~foo(" << val_ << ")\n";
  }
private:
  int val_;
};

int main() {
  auto f = foo{1};
  foo{2};
  { foo{3}; }
}
```

```
 foo::foo(1)
 foo::foo(2)
foo::~foo(2)
 foo::foo(3)
foo::~foo(3)
foo::~foo(1)
```

Embrace the Closing Brace!

}

- According to some, the most powerful feature of C++
- Deterministic destruction
- *Whatever the way we quit the scope!*
  End of scope, `break`, `return`, `throw`, `goto`, ...
- Unparalleled in other programming languages
  Different from Java's `finalize`,
  approximated by Python's "context managers" (`with`),
  etc.

Embrace the Closing Brace!

- According to some, the most powerful feature of C++
- Deterministic destruction
- *Whatever the way we quit the scope!*
  End of scope, `break`, `return`, `throw`, `goto`, ...
- Unparalleled in other programming languages
  Different from Java's `finalize`,
  approximated by Python's "context managers" (`with`),
  etc.

}

- According to some, the most powerful feature of C++
- Deterministic destruction
- *Whatever the way we quit the scope!*
  End of scope, `break`, `return`, `throw`, `goto`, ...
- Unparalleled in other programming languages
  Different from Java's `finalize`,
  approximated by Python's "context managers" (`with`),
  etc.

Embrace the Closing Brace!

}

- According to some, the most powerful feature of C++
- Deterministic destruction
- *Whatever the way we quit the scope!*
  End of scope, `break`, `return`, `throw`, `goto`, ...
- Unparalleled in other programming languages
  Different from Java's `finalize`,
  approximated by Python's "context managers" (`with`),
  etc.

Embrace the Closing Brace!

}

- According to some, the most powerful feature of C++
- Deterministic destruction
- *Whatever the way we quit the scope!*
  End of scope, `break`, `return`, `throw`, `goto`, . . .
- Unparalleled in other programming languages
  Different from Java's `finalize`,
  approximated by Python's "context managers" (`with`),
  etc.

Destruction in Action

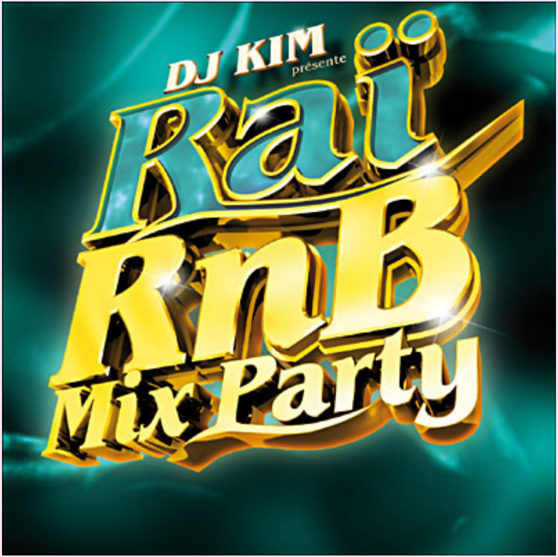
```
void bar(int i) {
  auto f1 = foo{i};
  if (i % 2 == 0)
    return;
  auto f2 = foo{1000 * i};
}

int main() {
  bar(1);
  bar(2);
  for (int i = 3;; ++i) {
    auto f = foo{i};
    if (i == 3)
      continue;
    else if (i == 4)
      break;
  }
  auto f = foo{51};
}
```

```
 foo::foo(1)
 foo::foo(1000)
foo::~foo(1000)
foo::~foo(1)
 foo::foo(2)
foo::~foo(2)
 foo::foo(3)
foo::~foo(3)
 foo::foo(4)
foo::~foo(4)
 foo::foo(51)
foo::~foo(51)
```

RAII

1. A Better C
2. My First C++ class
   - Introducing attributes and methods
   - Heart of the "O" Paradigm
   - Lifetime Management
     - Constructors
     - Destructor
     - RAII
   - Output Streamable
3. Low-Level Memory Management

Raï Is Not Dead



A Powerful Construct: The Destructor

- Destruction is deterministic
- Destruction happens immediately (no delays)
- Destruction *always* happens
  (Well, obviously not in case of abortion such as SEGV)
- Therefore, we can use the destructor to ensure code execution

A Powerful Construct: The Destructor

- Destruction is deterministic
- Destruction happens immediately (no delays)
- Destruction *always* happens
  (Well, obviously not in case of abortion such as SEGV)
- Therefore, we can use the destructor to ensure code execution

A Misnomer

Resource	Resource
Release	Acquisition
Is	Is
Destruction	Initialization

RAII Applied to File Descriptors

```
#include <sys/types.h>
#include <sys/stat.h> // open!!!
#include <fcntl.h>
#include <unistd.h> // close?!?  WTF???

class filedes {
public:
  filedes(int val)
    : val_{val} {}

  filedes(const char* path, int oflag)
    : filedes{open(path, oflag)} {}

  ~filedes() {
    close(val_);
  }
private:
  int val_;
};
```

```
int main()
{
  // Autoclose std::cout.
  auto fd1 = filedes{1};

  auto fd2
    = filedes{open("fd.cc",
                   O_RDONLY)};

  auto fd3
    = filedes{"fd.cc", O_RDONLY};
}
```

Known Uses of RAII

- Files (`std::stream`)
- Locks
- Threads
- etc.
- And of course...

Known Uses of RAII

- Files (`std::stream`)
- Locks
- Threads
- etc.
- And of course...

Memory!

Smart pointers (tomorrow)

Output Streamable

1. A Better C

2. My First C++ class
   - Introducing attributes and methods
   - Heart of the "O" Paradigm
   - Lifetime Management
   - Output Streamable

3. Low-Level Memory Management

Outputting (1/3)

In header file:

```
#include <iosfwd>

class circle
{
  // ...
};

// Operators will be better explained later.
std::ostream& operator<<(std::ostream& ostr, const circle& c);
```

Outputting (2/3)

In source file:

```
#include <iostream>

// ...

std::ostream& operator<<(std::ostream& ostr, const circle& c)
{
  return ostr << '(' << c.x_get() << ", "
              << c.y_get() << ", "
              << c.r_get() << ')';
}
```

Sample use:

```
auto c = circle{1, 6, 6.4};
std::cout << "circle at " << &c << ": " << c << '\n';
```

gives:

```
circle at 0xbffff7d0: (1, 6, 6.4)
```

Low-Level Memory Management

1 A Better C

2 My First C++ class

3 Low-Level Memory Management

malloc/free

- `malloc` and `free` are *only* about memory management
- They are not related to object lifetime
- They are not even typed!
- Hence *you* have to compute the size to allocate

new/delete

- Use `new` to allocate an object on the heap
  - Memory allocation (à la `malloc`)
  - Object construction
- Use `delete` to deallocate
  - Object destruction
  - Memory deallocation (à la `free`)

Stack vs Heap

```
class foo {
public:
  foo(int v) : val_(v) {
    std::cerr << " foo::foo(" << val_ << ")\n";
  }
  ~foo() {
    std::cerr << "foo::~foo(" << val_ << ")\n";
  }
private:
  int val_;
};

int main() {
  foo* f = new foo{51};
  foo g = foo{42};
  foo{96};
  delete f;
  foo{666};
}
```

```
 foo::foo(51)
 foo::foo(42)
 foo::foo(96)
foo::~foo(96)
foo::~foo(51)
 foo::foo(666)
foo::~foo(666)
foo::~foo(42)
```

Proper Use of `new`/`delete`

- The C++ library is rich
- It features many containers
- They shield us from having to allocate on the heap
- *Value semantics is much more common in C++ than in C*
- So you should have few `new`/`delete`
- Each `new` must have its `delete`
- And reciprocally!

A C++ Oddity

- To allocate an array, use `new[]`
- To deallocate it, use `delete[]`!

new[]/delete[]

```
static int counter = 0;
class foo {
public:
  foo() : foo{counter++} {}

  foo(int v) : val_{v} {
    std::cerr << " foo::foo(" << val_ << ")\n";
  }
  ~foo() {
    std::cerr << "~foo::foo(" << val_ << ")\n";
  }
private:
  int val_;
};

int main() {
  foo* fs = new foo[3];
  delete[] fs;
}
```

```
 foo::foo(0)
 foo::foo(1)
 foo::foo(2)
foo::~foo(2)
foo::~foo(1)
foo::~foo(0)
```

Never Mix new/delete and new[]/delete[]

```
int main() {
  foo* fs = new foo[3];
  delete fs;
}
```

```
 foo::foo(0)
 foo::foo(1)
 foo::foo(2)
foo::~foo(0)
new-delete-mix.exe(48517,0x7fff79da6000) malloc:
*** error for object 0x7f9ac8c033b8: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
```

Dynamic Memory Management

- In modern C++, `new`/`delete` are little used
- They are mostly useful for low-level code (e.g., libraries)
- Shared pointers are much better (see tomorrow)

1. A Better C
   - Handy Tools
   - References
   - C++ I/O Streams
2. My First C++ class
   - Introducing attributes and methods
   - Heart of the “O” Paradigm
   - Lifetime Management
   - Output Streamable
3. Low-Level Memory Management