C++ Workshop — Day 1 out of 5

Object

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

2015–2018
December 14, 2018

Outline

Outline

Outline

Foreword

Though frustrating for people who already "know" some stuff we will adopt a step-by-step introduction to C++ and OO.

Today:

```
auto  using  enum  type&  operator<<  class
public  private  =default  =delete  :  constexpr
operator=  this  const  explicit  {a,b}
new  new[]  delete  delete[]  nullptr
std::cout  std::vector
```

their meaning, and how to use them...

And that's a lot!

C

The C language is:

- imperative

```
int i = 1;
i = 2; /* updated */
```

- procedural (*not* functional)
- not type-safe

```
cowboy* c;
soccerplayer* s = (soccerplayer*)c; /* oops */
```

- compiled (*not* interpreted)
- roughly modular

Compiling actually means:

- run the preprocessor (text management with # macros and directives)

  `gcc -E toto.c > toto_.c`

- run the compiler

  `gcc -c toto.c` gives `toto.o`

- run the linker

  `ld *.o` gives the executable `a.out`

```
% gcc -Wall -ansi -pedantic -ggdb -c foo.c
% file foo.o
foo.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
% nm -C foo.o
U __assert_fail
000000de T main
0000004e T print_circle
U printf
00000000 T translate_circle
```

Reminder

You shall know:

- how to deal with file inclusion (and guards)
- when to use a forward declaration of a type (instead of file inclusion)
  avoid "`#include "toto.h"`" when "`struct toto;`" is enough
- how to handle mutually dependent types
- what is in .o files

Reminder

Don't

```
#define PI 3.14 // never!
```

```
void foo(const int i);
```

```
// ...
// ...
int i;
// some code w/o i...
// some code using i...
```

Do

```
const float pi = 3.14f;
// or better:
constexpr float pi = 3.14f;
```

```
void foo(int i);
```

```
// nice: i is not visible here
// some code w/o i...
int i;
// some code using i...
```

Reminder

```
//
// some code...
//
int i;
// some code using i...
```

```
// ...
// an 'i' might exist here
{
  int i;
  // some code using the local 'i'...
}
// the local 'i' is not visible
// ...
```

Outline

C++

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

Outline

Foreword

- `std` refers to the "standard library of C++"
- `std::vector<int>` is the type of an **array** of `int`
- the prefix `std::` means giving the full name of this type
- actually `std` is a `namespace`

'auto'

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

but:

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

'auto'

- `auto` is essential when types are unknown (in `template` world)
- `auto` is handy when types are really long:

```
typename std::vector<int>::const_iterator i = begin(v);
// vs
auto i = begin(v); // shorter!
```

- `auto` is robust to minor changes:
  the code above still compiles when `v` is turned into a `std::list`
- `auto` avoids stuttering code:

```
std::vector<std::string>* v = new std::vector<std::string>();
// vs
auto v = new std::vector<std::string>();
```

A Better 'typedef': 'using'

- `typedef` does **not** define a (new) type
  but define a name alias!
- The syntax of `typedef` is really dubious...

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

- Its legibility too...

```
typedef int arr[];
typedef int (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;
```

Weak typing in C and C++

In C and C++, you can mix:

- `bool` and `int`
- `enum` values and `int` values
- pointers

```
auto p1 = new soccerplayer;
auto p2 = (cowboy*)p1; // explicit cast
```

but you can do some reasonable low-level stuff with that

superposition of objects in memory...

A Better 'enum': 'class enum'

Plain enum:

```
enum month {
  january, february /*...*/ };
enum day {
  monday, tuesday /*...*/ };
```

Class enum:

```
enum class month {
  january, february /*...*/ };
enum class day {
  monday, tuesday /*...*/ };
```

Unsafe use:

```
// you get only a warning with:
std::cout << (january == monday) << '\n';
// true

bool b = monday;  // true?

std::cout << (february == 1) << '\n';
// true (C starts from 0...)

int sex = monday; // why not?
```

safe use:

```
auto
  m1 = month::january,
  m2 = month::february;
auto
  d = day::monday;

auto
  b1 = m1 == d,   // does not compile
  b2 = m2 == m1 + 1,       // likewise
  b3 = bool{day::monday}; // likewise
int sex = day::monday;     // likewise
```

that's weird!

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

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

Hint: use wisely, do not abuse...

Outline

What a reference is

A **reference** is:

- a non null constant pointer with a non-pointer syntax
- a variable that represents *an* object (one existing 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
- it is (just) an **alias**

A couple of exercises

```
int i = 1;
int& j = i;
j = 2;
bool b = i == 2;
// b is true or false?
```

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

Soluce (for C++ coder)

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

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

Soluce (for C coder)

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

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

Hint: prefer the former soluce!

Another example (swapping)

```c
// 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
// 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; this is VERY bad
  // 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) {    // likewise, modifies its input
  // same code but with ``c.''
}
```

Hints for beginners

Avoid:

```
type& routine() {
  type* p = // dyn. alloc.
  // ...
  return *p;
}
```

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

Prefer:

```
type* routine() {
  type* p = // dyn. alloc.
  // ...
  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;// RO alias
```

```
// better:

auto j1 = jumbo(10);
auto j2 = j1;   // copy
auto& j3 = j1;  // RW alias
const auto& j4 = j1;// RO alias
```

Outline

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'` (or `std::endl`)
char string type	`char*`	`std::string`
string stream	`#include <stdio.h>`	`#include <sstream>`
	use of `sscanf` and `sprintf`	use of `>>` and `<<`

Outline

Outline

Remember C $\Leftrightarrow$ Procedural Paradigm

A translation of the common assertion:

$$program \;=\; data\ structures \;+\; algorithms$$

```
// data structure:
typedef struct circle circle; // a type named 'circle'
struct circle
{
  float x, y, r;
};

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

The procedures' argument c is the **target** of the algorithms.
dx and dy are **auxiliary data**

(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
```

Hint: most compilers support *#pragma once* (put it on the 1st line) so you do not have to write these 3-line error-prone guards...

Terminology

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

Some terminology:

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

[1] a "regular" field like `r` for `circle`

[2] a routine with a clearly identified target.

Changes

In header file (`.h` / `.hh` or `.hpp`):

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); // mind the 'const'`
C++	`struct circle { ... void print() const; ... };`

Translation into C++ (2/3)

Sample use:

```
circle* c = // ...
c->translate(4, 5); // reading: circle at c,
                    //          do translate by (4,5)
c->print();

circle k;
// ...
k.translate(4, 5);
k.print(); // you talk to k, saying: do print (please!)
```

- Calling a method is just like accessing a structure field.
- `*c` and `k` are the **targets** (subjects) of method calls.
- The address of the target is given by the keyword `this`.

'this'

code	in "circle::print"
`circle* c1 = // ...`	
`circle* c2 = // ...`	
`c1->print();`	`this == c1`
`c2->print();`	`this == c2`
`circle k;`	
`k.print();`	`this == &k`
`const circle& k2 = k;`	
`k2.print();`	`this == &k`

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

Translation into C++ (3/3)

```cpp
// file circle.cc
#include "circle.hh"
#include <cassert>

void circle::translate(float dx, float dy)
{
  assert(this->r > 0.f); // 'this' is the address of the target
  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` / `.cc` or `.cpp`):

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 {...} // mind the 'const'`
C	`c->`
C++	`this->  // only valid in method code`

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; // does not compile
this->translate(0.f, 0.f); // does 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.

On a non-const instance, you can call both const and non-const methods.

Outline

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.

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

  having uninitialized variables is often evil!

- how to access information

  hiding information is great... to a certain extent!

- how to modify a particular attribute

  object state can change (we have an *imperative* language)

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)

```
// in circle.hh

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

```
// 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 RO access to attributes.

A **mutator** is a non-constant method that allows for modifying attributes.

```
class circle
{
public:
  //...
  float get_r() const;  // accessor (r_ is in Read-Only access)
                        // => const method

  void set_r(float r);  // mutator (r_ *may* change)
                        // => non-const method
  //...
};
```

Accessors and mutators (2/2)

In source file:

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

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

Ensures you that the radius remains positive.

Outline

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*.

Constructor

In header file:

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

In source file:

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

```
// Use it.
int main ()
{
  // Historical way:
  circle c1(0, 0, 1);

  // New ways:
  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(r > 0.f);
  // 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}
    // or use parenthesis
{
  // Possibly invalid object, but
  // well defined state.

  assert(r > 0.f);

  // Valid object.
}
```

Warning: The initializer list shall strictly follow the ordering of attributes.

Constructor vs Destructor

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

Destruction in Action

Objects live and die!

Here we have *static* memory allocations and dealloc.

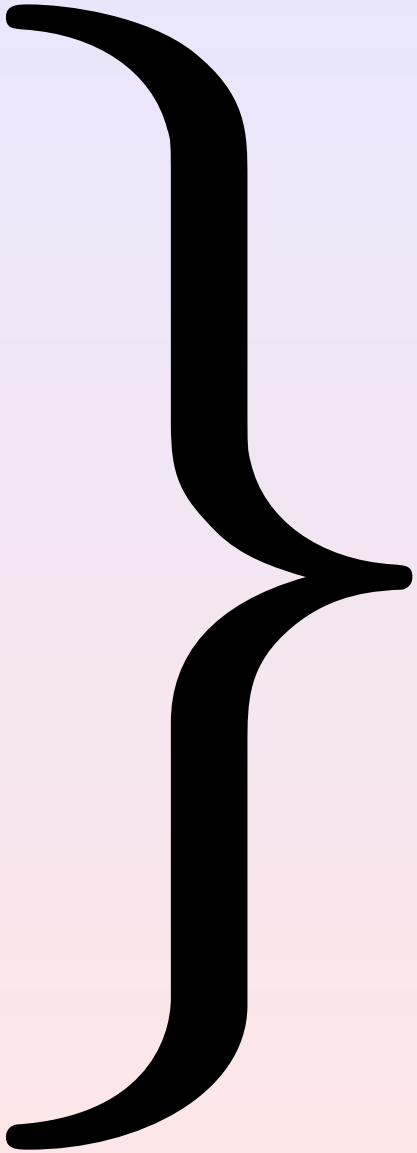
```
#include <iostream>

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}; }
}
```

static means "related to the compiler" or "at compile-time"

The compiler kills objects (since they were not dynamically allocated by a `new`).

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

Embrace the Closing Brace!



- A 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};
}
```

Everyone dies.

Relying on the compiler is great!

```
 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)
```

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

Raï Is Not Dead

A Misnomer

Resource Release Is Destruction

Resource Acquisition Is Initialization

RAII Applied to File Descriptors

```cpp
#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_;
};
```

```cpp
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 (day 2 and day 5)

Outline

Outputting (1/3)

In header file:

```
#include <iosfwd>

class circle {
  // ...
};

std::ostream& operator<<(std::ostream& ostr, const circle& c);
```

- C++ operators allow for some *syntactic* sugar
- a non-const stream `ostr` is an input, then is modified, last is returned
- here we have a **binary operator** that is a procedure
  - left operand = 1st argument
  - right operand = 2nd argument
  - so "`ostr << c`" means "`operator<<(ostr, c)`"

Outputting (2/3)

In source file:

```
#include <iostream>

// ...

std::ostream& operator<<(std::ostream& ostr, const circle& c)
{
  return ostr << '(' << c.get_x() << ", "
              << c.get_y() << ", "
              << c.get_r() << ')';
}
```

instead of:

```
...operator<<(operator<<(operator<<(ostr, '('), c.get_x()), ", ")...
```

Outputting (3/3)

Sample use:

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

gives:

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

Note: "`std::endl`" is "`'\n'` and *flush* the stream".

Outline

Outline

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 (FR: *le tas*)
  - Memory allocation (à-la `malloc`)
  - Object construction
- Use `delete` to deallocate
  - Object destruction
    (so call the proper destructor(s) to run some code)
  - Memory deallocation (à la `free`)
- This is **hand-made** *dynamic* (at run-time) memory management!
  vs *static* alloc./dealloc. in the stack (FR: *la pile*), by the compiler

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}; // or: auto* f = new foo{51};
                        // or: auto  f = new foo{51};
  auto g = foo{42};
  foo{96};
  delete f;
  foo{666};
}
```

Left as exercise: justify the following sequence...

```
 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

  including `std::vector<T>` for resizable arrays
- 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`

  actually, very few, if not none!
- Each `new` must have its `delete`

  and reciprocally!

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;
}
```

To allocate an array, use `new[]`

To deallocate it, use `delete[]`!

```
 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;  // oops!
}
```

```
 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

- An uninitialized pointer shall be set to `nullptr`
  (forget `NULL` and `0`)
- In modern C++, `new`/`delete` are little used
  yet you should know about them
- They are mostly useful for low-level code (e.g., libraries)
- Shared pointers are much better; they are *smart*!
  we will see them later...

Outline

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; // safety
}
```

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

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

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

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; // safety
}
```

the call `naive(run)` makes bug being a copy of run, so we have "`bug.ptr_ == run.ptr_`"; then `delete` is called **twice** on this addr with `bug.~easy()` (end of `naive`) and `run.~easy()` (end of `main`)!

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

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

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

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

  // an oddity:
  foo c(); // does not compile in C++...
           // use: foo c; or foo c{};

} // 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:

expression	*value*
`C->i` and `C->ptr`	undefined
`a.i` and `a.ptr`	undefined
`b.i` and `b.ptr`	resp. equal to `a.i` and `a.ptr` (copy of)
`aa.i` and `aa.ptr`	likewise (assignment of)

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
```

C++ special methods

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

	special methods:
`type::type()`	default constructor
`type::type(const type&)`	copy constructor
`type& type::operator=(const type&)`	assignment operator
`type::~type()`	destructor

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

About Idioms

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

Constructor: Delegation (C++ 11)

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

// 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}
{}
```

Constructor: Delegation (C++ 11)

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

// 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:

```
// code factorization (is great!)
circle::circle(float x, float y,
               float r)
  : x_{x}, y_{y}, r_{r}
{
  assert(r > 0.f); // always tested!
}

circle::circle(float r)
  : circle{0, 0, r} // calls the above
                    // version
{}

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

Hint: avoid using default arg values!

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; // use the
                      // defaults
private:
  // default member values:
  float x_ = 0, y_ = 0, r_ = 1;
};
```

Yes, you're not dreaming: we're coding methods directly in the header file (within the class declaration scope). We can actually do it, but just forget it.

Cool C++ 11 features

Explicitly forbid cpy ctor, op=

```
class limited
{
public:

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

  limited(const limited&) = delete;
  void operator=(const limited&)
                          = delete;
  // ...
};
```

Explicitly say: add a default impl

```
class lazy
{
public:

  lazy() = default;
  lazy(const lazy&) = default;

  // ...
private:
  float f;
};
```

Sample use:

```
auto l = lazy{};
// note that lazy.f is undefined...
```

Cool C++ 14 features

Default values and useless constructors:

```
// C header file in C++: 1st char is 'c', and no final ".hh"
#include <cassert>

struct point
{
  float x = 0, y = 0;  // public so...
};
```

Sample use:

```
auto p = point{},
     q = point{1, 1};  //...no constructor is required :-)

assert(p.x == 0 and p.y == 0);  // prefer "and" over "&&"
```