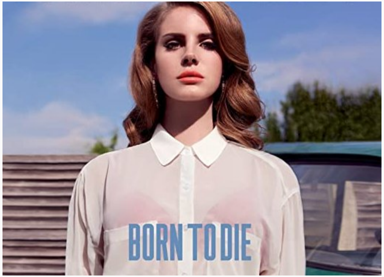
C++ 05 - Objects are 

Feb'22

EPITA Research & Development Laboratory (LRDE)

Review and outlook

The object’s birth

Destructor

Extra

Review and outlook

Review and outlook

OO what we have seen so far

- A class bundles **data** and **algorithms**
- A class ensures **coherence** (always in a *valid state*)
- Coherence is ensured with **const-ness** and **visibility** restriction

[1] A relation notion is “move”, introduced in Unit 10

Review and outlook

OO what we have seen so far

- A class bundles **data** and **algorithms**
- A class ensures **coherence** (always in a *valid state*)
- Coherence is ensured with **const-ness** and **visibility** restriction

What have not seen yet

- How an object comes to life
- How an object dies
- That objects can be copied and assigned during its life[1]
- How to enforce coherence for the entire object's lifetime

[1] A relation notion is "move", introduced in Unit 10

The object’s birth

Constructor

Constructor

Constructors are special member functions, used to **initialize** objects

When

As soon as you **declare** an instance of the class (its birth), **a**[1] *constructor* is called.

```
MyClass c;              MyClass x = {2,3};        auto x = MyClass{}
MyClass c(2,3);         MyClass x = {};
MyClass c{2,3};         MyClass y = x;²
```

[1]: **a** $\neq$ **the** (it would be too simple 😈)
[2]: do not confuse **initialization** `auto x = y;` with **assignment** `x = y;`

Usage

You may want to customize object initialization from the *default* because:

- you have some sensible default-values for the members of your object
  → Use **default member initializers**
- you want to initialize the object state and ensure coherence at construction !
  → Use a **custom constructor**

Customizing the object initial state

```cpp
struct Point{
  int x;
  int y;
};
```

```cpp
Point p;
// p.x → undefined
```

```cpp
struct Point{
  int x = 0;
  int y = 0;
};
```

```cpp
Point p;
assert(p.x == 0);
```

```cpp
class Circle {
  int x_, y_, r_;
public:
  Circle(int x, int y, int radius);
};

// In Circle.cpp
Circle::Circle(int x, int y, int radius)
  : x_{x}, y_{y}, r_{radius}{
  assert(radius > 0);
}
```

```cpp
Circle c(0,0,1);
assert(c.get_radius() == 1);
```

Construction rules (1/2)

```
struct MyClass { int a;  int b = 42; int c = 51; };
```

Legacy C++	Modern C++	**POD** (a,b,c)
`MyClass obj1;`	No uninitialized variables	(?, 42, 51)
`MyClass obj2 = {};`	`auto obj2 = MyClass{};`	(0, 42, 51)
`MyClass obj3 = {1,2};`	`auto obj3 = MyClass{1,2};`	(1, 2, 51)
No named initialization	`auto obj4 = MyClass{.a=1, .c=3};`	(1, 42, 3)
`MyClass cpy = obj3;`	`auto cpy = obj3;`	(1, 2, 51)

Fields get initialized by priority:

1. From the initialization list
2. From the member initializers
3. Otherwise, they have default values (zero or *undefined*)

The copy construction copies all fields.

Construction rules (2/2)

For classes, you can customize constructors (*if needed*):

```
class Circle {
  int x_ = 0, y_ = 0, r_ = 1;
public:
  Circle() = default;                         // #1
  Circle(int x, int y, int radius = 1); // #2
  Circle(const Circle&) = default;          // #3 Copy-constructor auto-generated by default
};
```

```
Circle::Circle(int x, int y, int radius)
  : x_{x} , y_{y}, r_{radius}{
  assert(radius > 0);
}
```

Legacy C++	Modern C++	Constructor calls	(x,y,r)
`Circle c;`	`auto c = Circle{};`	`Circle()` #1	(0,0,1)
~~`Circle c();`~~			Error
~~`Circle c(1);`~~	~~`auto c = Circle{1};`~~		Error
`Circle c(1,2);`	`auto c = Circle{1,2};`	`Circle(int,int)`	(1,2,1)
`Circle c(1,2,3);`	`auto c = Circle{1,2,3};`	`Circle(int,int,int)`	(1,2,3)
`Circle cpy = c;`	`auto cpy = c;`	`Circle(const Circle&)` #3	(1,2,3)

Legacy C++	Modern C++	Constructor calls	(x,y,r)
`Circle c;`	`auto c = Circle{};`	`Circle()` #1	(0,0,1)
~~`Circle c();`~~			Error
~~`Circle c(1);`~~	~~`auto c = Circle{1};`~~		Error
`Circle c(1,2);`	`auto c = Circle{1,2};`	`Circle(int,int)`	(1,2,1)
`Circle c(1,2,3);`	`auto c = Circle{1,2,3};`	`Circle(int,int,int)`	(1,2,3)
`Circle cpy = c;`	`auto cpy = c;`	`Circle(const Circle&)` #3	(1,2,3)

C++ has **sensible** auto-generated constructors with `= default`.

- The generated **default constructor** (#1) uses the same rules as for the POD
- The generated **copy constructor** (#3) copies all fields

Code less !

You should customize these constructors only if you really need it (in few cases)

Constructor - In detail

```cpp
Circle::Circle(int x, int y, int radius)
  : x_{x}
  , y_{y}
  , r_{radius}
{
  assert(radius > 0);
  // NO return!
}
```

Signature
Initializer list
Ctor function body

Member initializer list in constructors

- Use the member initializer list as much as possible
- Check invariants in the constructor body.

Constructor delegation

✖

```cpp
class Circle { ...
  // General case.
  Circle(int x, int y, int radius = 1);
  // Centered circle.
  Circle(int radius);
  // Unit circle
  Circle() = default;
};
// In cpp
Circle::Circle(int x, int y, int r)
  : x_{x}, y_{y}, r_{r}{
  assert(r > 0);
}
Circle::Circle(int r)
  : x_{0}, y_{0}, r_{r}{
  // There's a bug here!
}
```

✔

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

```cpp
class Circle { ...
  // General case.
  Circle(int x, int y, int radius = 1);
  // Centered circle.
  Circle(int radius);
  // Unit circle
  Circle() = default;
};
```

Beware the default value in function

`Circle(int x = 0, int y = 0, int radius = 1)` would be a bad API
→ you do not want to set the `x` value of the center alone

however:

```cpp
Circle(Point center = {0,0},
       int radius = 1)
```

is ok and equivalent to the rightside

```cpp
class Circle {
  int x_ = 0, y_ = 0, r_ = 1;
public:
  Circle() = default;
  Circle(Point center, int radius = 1);
}
```

Conclusion about constructor rules

- Use auto-generated constructors
  and constructor delegation when possible
- When writing custom constructors, use the *member initializer list*
- Abuse of contract checking, at least in debug-mode
- Prefer modern style `auto ... = X{...};` over old style

Destructor

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.

Destructor

Usage

Define a destructor if an object needs **explicit** actions when it dies.

- Destruction is deterministic,
  - it happens in the reverse order of constructions
- Destruction happens immediately when the object goes out-of-scope
  - (no delays/overhead induced by garbage collectors)
- Destruction *always* happens
  - (Well, obviously not in case of abortion such as SEGV)
- Therefore, we can use the destructor to ensure code execution (Nice!)

Destructor

Usage

Define a destructor if an object needs **explicit** actions when it dies.

- Destruction is deterministic,
  - it happens in the reverse order of constructions
- Destruction happens immediately when the object goes out-of-scope
  - (no delays/overhead induced by garbage collectors)
- Destruction *always* happens
  - (Well, obviously not in case of abortion such as SEGV)
- Therefore, we can use the destructor to ensure code execution (Nice!)

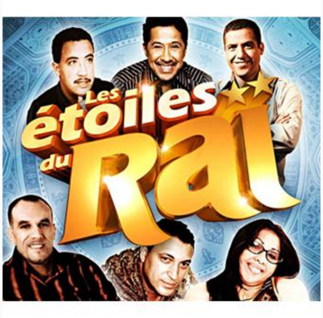
Be lazy !

There is an *auto-generated* destructor by default.
Define the destructor if you need *non-default* cleanup actions.

The RAII in the place

Two (confusing) names

- **RAII**: Resource Acquisition is Initialization
- **SBRM**: Scope Bound Resource Management



for a *simple* idea

- Resources are handled by an object and
- Objects clean up after themselves

Putting the concept to use - File descriptor

```
#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};
} // all fdX go out of scope
  // -> destruction
```

Putting the concept to use - File descriptor

```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};
} // all fdX go out of scope
  // -> destruction
```

What about:

```cpp
...
  auto fd4 = fd3;
} // What happens for fd3 and fd4 ?
```

Putting the concept to use - File descriptor

```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};
} // all fdX go out of scope
  // -> destruction
```

What about:

```cpp
...
  auto fd4 = fd3;
} // What happens for fd3 and fd4 ?
```

Rule of 0/3

A constructor is never be defined alone.
See next course !

Known Uses of RAII

Advantages

- Clear ownership of the resource
- Never forget to close the file / release the resource again

- Files (`std::stream`)
- **Locks**
- Threads
- etc.

- And of course…

Known Uses of RAII

Advantages

- Clear ownership of the resource
- Never forget to close the file / release the resource again

- Files (`std::stream`)
- **Locks**
- Threads
- etc.

- And of course...

Memory!

Smart pointers (Unit 8, 9)

Extra

Constructor troubles - implicit conversion

What is the difference between using `x_(x)` and `x_{x}` ?

→ `x_{x}` restricts which implicit conversions can be used.
→ Disallows narrowing and widening conversion.

```
// Compiles
circle::circle(int x, int y, int r)
  : x_(x), y_(y), r_(r){...}

// Does not compile
circle::circle(float x, float y, float r)
  : x_{x}, y_{y}, r_{r}{...}
```

Constructor troubles - implicit conversion

"The same" goes for the constructor call, however it only affects narrowing conversions.

```
auto d = 2.2L; // d is a double
// Compiles
auto c1 = circle(d); // Conversion double -> float
// Does not compile
auto c1 = circle{d};
```

Avoid implicit conversion

Hint: Always use the `{}` version and make the conversion explicit.
If not possible, at least comment your code.