C++ 04 - OOP
Encapsulation 2 and Access restriction

Feb’22

EPITA Research & Development Laboratory (LRDE)

Review and outlook

Accessibility

const-ness

Review and outlook

Review and outlook - Encapsulation

- How to bundle data and algorithms into a `class`: ✔
- How to restrict access to the data (the members): ✖
- How this is used to enforce coherence of an object: ✖

Review and outlook - Encapsulation

- How to bundle data and algorithms into a `class`: ✔
- How to restrict access to the data (the members): ✖
- How this is used to enforce coherence of an object: ✖

Additional keywords

Const-ness

- Keyword: const
- Specifies which member functions may change the data

Accessibility

- Keywords: public and private
- Specifies who can access the member (functions)
- public: everyone
  private: only member functions

Accessibility

Ensuring coherence - Access 1

Rule: Do not expose the data

All members of a class are private, as they are by default.
The interface of the class is provided by member functions.

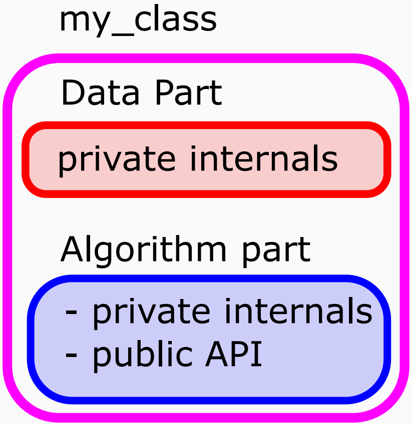
```cpp
class circle{
//``algorithm''-part
public:
  void translate(float dx, float dy);
  void print();
//``data''-part
private:
  float x_, y_, r_;
};
```

Commonly, a private member (function) gets “_” as a suffix or `m_` as a prefix.

Ensuring coherence - Access 2

If necessary, **accessors/mutators** (**getters/setters**) are provided.

```
// circle.hpp
class circle{
  // ...
  float get_r();
  void set_r(float new_r);
}
// circle.cpp
void circle::set_r(float new_r){
  // Ensure coherence
  assert(new_r > 0.f);
  r_ = new_r;
}
```



This allows to enforce **invariants** of your class during runtime.

A note on C++ struct's

C++ conserves the C keyword struct commonly used a simple aggregation of data members (POD = Plain Old Data)

It allows a nice initialization syntax ([1])

Only technical difference:

- everything is (by default) `private` in a `class`
- everything is (by default) `public` in a `struct` (as in 'C')

By convention

All member variables should be public, and we should add no member functions (and no constructors/destructors [2])

[1] Next course

[2] Next course

A proper use of aggregates

In a *2D point*, x and y are rather independent → no coherence to preserve

```
struct point2d
{
  int x;
  int y;
};
```

Equivalent to:

```
class point2d
{
  public:
    int x;
    int y;
};
```

Usage

- `point2d p = {2, 3};` (aggregate initialization)
- `point2d q = {.x = 2, .y = 0};` (C++ 20 designated initialization)

const-ness

Ensuring coherence - Const 1

Rule: Enforce const-ness

If it makes sense for a class to have *const instances*, then
→ all member functions that do not modify member variables **have to be** marked `const`

```
class circle{
//``algorithm''-part
public:
  void translate(float dx, float dy); // Can't be const
  void print() const; // Has to be const
//``data''-part
private:
  float x_, y_, r_;
};
```

This prevents unexpected changes of the object when passed by *const references/pointers* to functions.

Ensuring coherence - Const 2

Rules

- A **const** member function can not modify the member variables
- A **const** member function can not call *non-const* member functions
- A **non-const** member function can not be called on a *const* instance

All of this is enforced during compilation! → Improve code safety
⇒ Important first debugging step of your code!

```
void circle::print() const {
  // The two following lines do not actually change the
  // target (this), yet they do not compile!
  r_ += 0.f;
  translate(0.f, 0.f);
}
circle c1; // Create an instance
const circle& c1cref = c1; // Create a CONST reference of it
c1cref.translate(0.f, 0.f); // Does not compile
```

const is a promise !

✍ **Const** is an implicit API contract

A **const** member function can not modify its member variables…

→ It is assumed that it does not change the **object state**
→ It is assumed that it can be called safely by multiple threads !

const is a promise !

✍ **Const** is an implicit API contract

A **const** member function can not modify its member variables…

→ It is assumed that it does not change the **object state**
→ It is assumed that it can be called safely by multiple threads !

```cpp
class Cache {
public:
  int get_value() const {
    if (!computed_) {
      value_ = some_computation();
      computed_ = true;
    }
    return value_;
  }
private:
  mutable int  value_;  // <- *mutable* allows to modify value in const member functions
  mutable bool computed_;
};
```

What if `get_value()` is legitimately called concurrently ? → bad things happen !
So don’t use obscure C++ features that break the rules