C++ 03 - Object Oriented Programming - Encapsulation 1

Feb'22

EPITA Research & Development Laboratory (LRDE)

Encapsulation

Scope and instances

Mutual dependence

Encapsulation

What is *Object-Oriented Programming*, and why use it?

- In procedural programming (like in C), the *data*, often in the form of structures, is passed to *procedures* that use/modify them
- The general idea behind object-oriented programming (OO) is to bundle data and functionalities

Decl. in C

circle.h

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

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

What is *Object-Oriented Programming*, and why use it?

- In procedural programming (like in C), the *data*, often in the form of structures, is passed to *procedures* that use/modify them
- The general idea behind object-oriented programming (OO) is to bundle data and functionalities

Decl. in C

circle.h

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

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

Decl. in C++

circle.hpp

```c
struct circle{
  //``algorithm''-part
  void translate(float dx, float dy);
  void print();

  //``data''-part
  float x, y, r;
};
```

Terminology

Encapsulation

Action of *grouping* data and algorithms into a structure.

Some terminology:

C Coder	C++	OO	meaning
structure field[1]	member	attribute	state (`data`)
function[2]	member function	method	behaviour (`algo`)

[1] a “regular” field like `cpp r` for `cpp circle`

[2] a routine with a clearly identified target

Encapsulation cont'd

Suppose a sorted vector used as set

"C-style" procedural

```
IntVec my_set;
list_insert(my_set, 0, 1); // [0]
list_insert(my_set, 1, 3); // [0,3]
list_insert(my_set, 1, 2); // [0,2,3]
list_bsearch(my_set, 3);   // OK
list_insert(my_set, 1, 5); // [0,5,2,3]
list_bsearch(my_set, 2);   // (Broken)
```

OO in C++

```
auto my_set = Set<int>();
my_set.add(1);  // Insert sorted
my_set.add(2);  // Insert sorted
my_set.find(3); // BSearch
```

Scope and instances

Class scope and this pointer

Class Scope

The scope of a member (function) of a class begins at its declaration and includes the rest of the class body.
This includes **all member** function bodies.

Class scope and this pointer

circle.hpp

```
struct circle{
  //``algorithm''-part
  void translate(float dx,
                 float dy);
  void print();
  void trans_print(float dx,
                   float dy);

  //``data''-part
  float x, y, r;
}
```

circle.cpp

```
#include "circle.hpp"

void circle::translate(float dx, float dy){
  x += dx;
  y += dy;
} // Only dx/dy go out of scope
void circle::print() {
  std::cout << "(x="  << x << ", y=" << y
            << ", r=" << r << ')';
}
void circle::trans_print(float dx, float dy){
  translate(dx, dy);
  print();
}
```

Classes and instances

- We call the definition of such a bundle of data and algorithms a **class**.
- A **class** is a blueprint of how to create and use objects.
- An actual object obtained from the class is called an **instance**.
- We also say the instance is described by the class.

```
#include "circle.hpp" // Contains the class circle

int main(){
  circle c; // c is an instance of the class circle
}
```

Accessing members and member functions (from the outside)

	Reference	Pointer
Member function	`obj.foo()`	`obj->foo()`
Member variable	`obj.var`	`obj->var`

```
int main(){
  circle c1; // c1 is an instance of the class circle

  // We can access member (functions) of objects via "."
  c1.translate(1.1, 2.2);
  c1.print();

  // To access them via pointers we need to use "->"
  circle* c1ptr = &c1;
  c1ptr->translate(-1.1, -2.2);
  c1ptr->print();
}
```

Class scope and this pointer - cont'd

Reconsider

```
// Contains the class circle
#include "circle.hpp"

int main(){
  circle c1; // c1 instance of circle
  circle c2;

  // We can access member (functions)
  // of objects via ``.''
  c1.translate(1.1, 2.2);
}
```

```
// file circle.cpp
void circle::translate(float dx,
                       float dy){
  x += dx;
  y += dy;
}
```

How does `circle::translate` know whether we want to modify the member `x` of `c1` or `c2`?

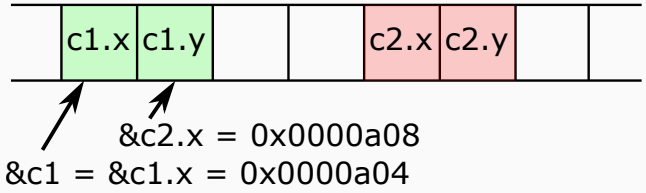
Class scope and this pointer - cont'd

When a member function is called on a **target**, implicitly the address of **target** is passed along as well.
It is accessible under the name this inside the function bodies.

```
// circle.cpp
circle::translate(float dx,
                  float dy){
  this->x += dx;
  this->y += dy;
}
```

```
// main.cpp
circle c1;
circle c2;
c1.print(); // "this" is &c1
c2.print(); // "this" is &c2
```

In `c1.print()`, this = 0x0000a04



Mutual dependence

Mutually dependent types

Sometimes, a type depends on another, possibly in different files.
A `page` contains a set of `circle`, but the `circle` needs to know its page.

circle.hpp

```
// Declaration of the type "page"
struct page;

// Decl + Def. of the type circle
struct circle{
  float x, y, r;
  // A forward declaration is enough
  page* p;
};
```

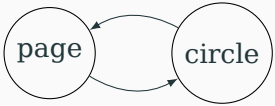
page.hpp

```
#include "circle.hpp"
struct page{
  // ...
  std::vector<circle> circles;
}
```

`circle.hpp` and `page.hpp` are included into `circle.cpp` and `page.cpp`.
The definition of `circle` needs to know how the `page` works and vice-versa.

Mutually dependent types

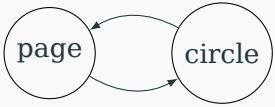
- circle : Definiton
- circle : Declaration
- → : SRC depends on DST



Cycle ☞

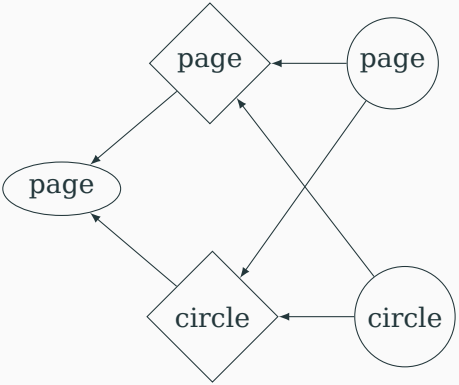
Mutually dependent types

- circle : Definiton
- circle : Declaration
- → : SRC depends on DST



Cycle 👉

- circle : Forward Declaration



No cycle 👍

Guidelines

- Use opaque (incomplete) types in headers only (mostly to break circular dependencies)
- Opaque types can only be used as pointer/reference type (or return type)
- You need full-definition when you *use* them (in `.cpp`)

Incomplete types (ab)uses

```
struct Page;
struct Circle;
```

Expression	Validity
`struct Book { Page* index; };`	✔
`struct Book { Page index[256]; };`	✖
`void addCircle(Page* p, Circle* c);`	✔
`void addCircle(Page& p, Circle& c);`	✔
`void addCircle(Page& p, Circle c);`	✖
`Circle* createCircle(Page& p);`	✔
`Circle  createCircle(Page& p);`[1]	✔

[1]: But the user of this function will need the definition of `Circle`.

Guidelines

- Use opaque (incomplete) types in headers only (mostly to break circular dependencies)
- Opaque types can only be used as pointer/reference type (or return type)
- You need full-definition when you *use* them (in `.cpp`)

Incomplete types (ab)uses

```
struct Page;
struct Circle;
```

Expression	Validity
`struct Book { Page* index; };`	✔
`struct Book { Page index[256]; };`	✖
`void addCircle(Page* p, Circle* c);`	✔
`void addCircle(Page& p, Circle& c);`	✔
`void addCircle(Page& p, Circle c);`	✖
`Circle* createCircle(Page& p);`	✔
`Circle  createCircle(Page& p);`[1]	✔

[1]: But the user of this function will need the definition of `Circle`.

☞ The PIMPL offers better alternative to hide implementation details.