

Beginners' notions

typedef does not define a (new) type but define a name alias.

Possible to define default value for arguments in function

Reference

A reference is a non null constant pointer with a non pointer syntax, an alias and a variable representing an object.

```
1 int i = 1;
2 int& j = i;
3 j = 2;
4 bool b = i == 2; // true
```

C++ swap

```
1 void swap(int& i1, int& i2)
2 {
3     int tmp = i1;
4     i1 = i2;
5     i2 = tmp;
6 }
```

```
1 void foo(circle c) { }
2 void foo(const circle& c) { } // Better, no copy
```

Auto

auto is a placeholder for a “basic” type, hold a (deep) copy, can be qualified with const, *, &

```
1 jumbo j1 = jumbo(10);
2 jumbo j2 = j1;
3 jumbo& j3 = j1;
4 const jumbo& j4 = j1;
5 // Better
6 auto j1 = jumbo(10);
7 auto j2 = j1;
8 auto& j3 = j1;
9 const auto& j4 = j1;
```

Flux

```
1 #include <iostream>
2
3 std::cout << "FOO"
4           << true
5           << 23
6           << '\n'
```

std::cout => flux de sortie de la lib c standard, less flexivle than printf, IO manipulators to control formatting, type safe

My first C++ class

C is follows a procedural paradigm.

In C++, after defined struct type, not mandatory to put struct before name of type when calling it

```
1 #ifndef CIRCLE_HH
2 #define CIRCLE_HH
3 struct circle {
4     void translate(float dx, float dy);
5     void print();
6
7     float x, y, r;
8 }
9
10 void foo(const circle& c) { }
11
12 #endif
```

Method calling in structure == usual attribute calling

```
1 circle c* = //...
2 c->translate(x, y);
3 c->print();
```

“this->something” can be simplified in “something” when 0 ambiguity

```
1 //file circle.cc
2 #include "circle.hh"
3 #include <assert.h> // == #include <cassert>
```

```
1 void print () const { }
```

A method is tagged “const” if it doesn’t modify the target

Object paradigm

Two keywords : * *public* means “accessible from everybody” * *private* means “only accessible from methods of the same structure”

A class is a structure using both encapsulation and information hiding

```
1 class circle {
2 public:
3     void translate(float dx, float dy);
4     void print();
5 private:
6     float x_, y_, r_;
7 }
```

The *interface* is the public part of a class Hint : an interface contains only methods, attributes are private

An *object* is the instance of a class

A *constructor* has the name of the class

```
1 class circle
2 {
3 public:
4     circle(float x, float y, float r);
5     //..
6 }
7
8 circle::circle()
9 {
10     this->x_ = x;
11     this->y_ = y;
12     this->r_ = r;
13 }
```

An *accessor* is a constant method that gives RO acces to attributes A *mutator* is a non constant method that allows for modifying attributes

```
1 class circle {
2 public:
3     float get_r () const; // accessor, Read-Only
4     void set_r(float r); // mutator (r_ *may* change)
5 }
```

Lifetime management

How to handle birth and death of objects : *lifetime management*

```
1 //Use it
2 int main ()
3 {
4     // Historical way:
5     circle c1(0, 0, 1);
6
7     // New ways:
8     circle c2{0, 0, 1};
9     circle c3 = {0, 0, 1}
10
11     // Preferred:
12     auto c4 = circle{0, 0, 1}
13 }
14 circle::circle(float x, float y, float r)
15     :x_{x}, y_{y}, r_{r}
16 {
17     assert (r > 0.0f);
18 }
```

Here we have static memory allocations and dealloc, the compiler will kill those objects since they were not dynamically allocate with a new.

```
1 int main()
2 {
3     auto f = foo{1};
4     foo{2};
5     { foo{3}; }
6 }
```

The closing brace is a powerful feature of C++, deterministic destruction.

The *destructor* must be deterministic and is mandatory.

```
1 class filedes {
2 public:
3     ~filedes() {
4         close(val_);
5     }
6 }
```

Output streamable

C++ operators allow for some syntactic sugar, a non-const stream ostr is an input, then is modified, last is returned.

In header file:

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

In source file:

```
1 std::ostream& operator<<(std::ostream& ostr, const circle& c)
2 {
3     return ostr << '(' << c.x_get() << ","
4                 << c.y_get() << ","
5                 << c.get_r() << ')';
6 }
```

```
1 std::cout << "circle at " << &c << ": " << c << '\n';
```

gives “circle at 0x0006ffff: (0.4, 0.4, 1)”

Low-level memory management

New/delete

malloc and free are only about memory management

They are not related to object lifetime and not even typed.

In C++, use new to allocate an object on the heap, memory allocation and object construction.

Use delete to deallocate, object destruction and memory deallocation.

Hand-made dynamic memory management.

C++ library is rich, features many containers (including `std::vector<T>` for resizable arrays).

Should have few new and delete.

`new[2]` to allocate an array of 2 and `delete[]` to deallocate it.

Uninitialized pointer should be set to `nullptr` (not `NULL` or `0`)

In modern C++, new/delete little used, mostly for low-level codes.

Some C++ idioms

```
1  class circle
2  {
3  public:
4      circle(float x, float y, float r)
5          : x_{x}, y_{y}, r_{r}
6      {}
7      circle(float r)
8          : r_{r}
9      {}
10     circle() = default;
11 private:
12     float x_ = 0, y_ = 0, r_ = 1;
13 };_
```