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.

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

C++ swap

```
void swap(int& i1, int& i2)
{
  int tmp = i1;
  i1 = i2;
  i2 = tmp;
}
```

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

Auto

auto is a placehoder for a "basic" type, hold a (deep) copy, can be qualified with const, *, &

```
jumbo j1 = jumbo(10);
jumbo j2 = j1;
jumbo& j3 = j1;
const jumbo& j4 = j1;
// Better
auto j1 = jumbo(10);
auto j2 = j1;
auto& j3 = j1;
const auto& j4 = j1;
```

Flux

```
#include <iostream>

std::cout << "FOO"
          << true
          << 23
          << '\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

```
#ifndef CIRCLE_HH
  #define CIRCLE_HH
struct circle {
  void translate(float dx, float dy);
  void print();

  float x, y, r;
}

void foo(const circle& c) { }

#endif
```

Method calling in structure == usual attribute calling

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

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

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

```
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

```
class circle {
public:
  void translate(float dx, float dy);
  void print();
private:
  float x_, y_, r_;
}
```

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

An *object* is the istance of a class

A *constructor* has the name of the class

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

circle::circle()
{
  this->x_ = x;
  this->y_ = y;
  this->r_ = r;
}
```

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

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

Lifetime management

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

```
//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}
}
circle::circle(float x, float y, float r)
  :x_{x}, y_{y}, r_{r}
{
  assert (r > 0.0f);
}
```

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

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

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

The *destructor* must be deterministic and is mandatory.

```
class filedes {
public:
  ~filedes() {
    close(val_);
  }
}
```

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:

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

In source file:

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

```
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

```
class circle
{
public:
  circle(float x, float y, float r)
    : x_{x}, y_{y}, r_{r}
  {}
  circle(float r)
    : r_{r}
  {}
  circle() = default;
private:
  float x_ = 0, y_ = 0, r_ = 1;
};_
```