

Loïc YON
Assistant Professor at ISIMA

Software development

Operations Research
Software engineering
Performance

loic.yon@isima.fr

https://perso.isima.fr/loic

https://isima.kiux.net

- Teams ?
- Not so easy to communicate

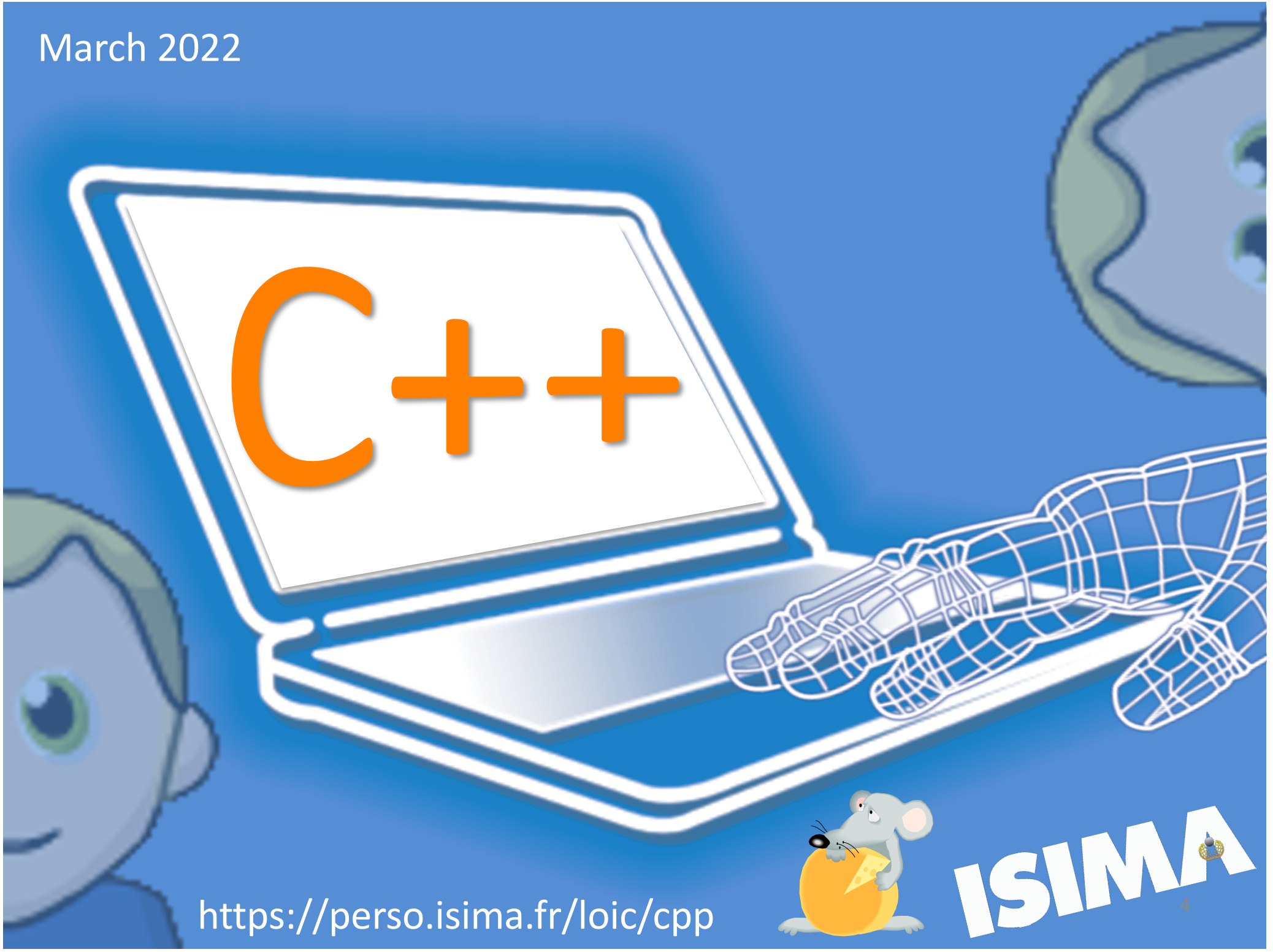
- Thanks Bi Dong !!!
- 谢谢 毕栋

Requirements

- Compilers
  - g++
  - clang++ `https://godbolt.org/`
- OS
  - Linux
  - Windows/WSL
- Other tools
  - Editor with syntax highlighting (VS Code)
  - A debugger (gdb …)
  - valgrind
  - make



March 2022



https://perso.isima.fr/loic/cpp

ISIMA

Overview

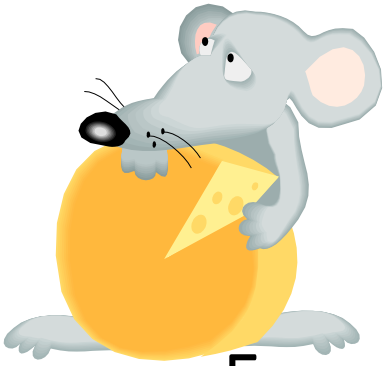
- Around 30h

- Practicing is more important than lectures !
  - Direct use of the lessons
  - Use of tools : *git*, *valgrind*, *makefile*
  - Unit tests





Plan



1. ++C 5
2. Fundamentals 21
3. Subtleties 72
4. How to code the string 98
5. Exceptions 123
6. Templates 148
7. Standard Library 171
8. It can be frightening… 218

Far away these slides were designed by Bruno-Laurent GARCIA, ISIMA
They were improved by David HILL and Christophe DUHAMEL, ISIMA
And reviewed by Violaine ANTOINE and Bruno BACHELET

Presentation

- Stroustrup, 1983
- Based on C
- Strongly typed
- Object oriented language
  - Classes from Simula
  - Non object variables
  - Functions
- Genericity support
  - Functions
  - Classes
- Exceptions handling
  - Advanced mecanism of handling errors



– Norms : 1999 (2003), 2011 (2014), 2017, 2020

NO OBject



C / C++ / ++C ?



ISIMA

More than C

- ✓ Functions overloading
- ✓ Parameters with values by default
- ✓ Mute parameters
- ✓ Constants
- ✓ Reference type
- ✓ Handling of memory
- ✘ Cast operators
- ✘ Runtime information
- ✘ Advanced genericity
- ✘ Advanced standard library
- ✘ 2011 norm
- ✘ HMI (Qt …)

Not included in this course



First program

```
#include <iostream>

// using namespace std;
// using std::cout;

int main(int argc, char ** argv)
{
   std::cout << "Hi guys " << 2022;
   std::cout << std::endl;

   return 0;
}
```



import

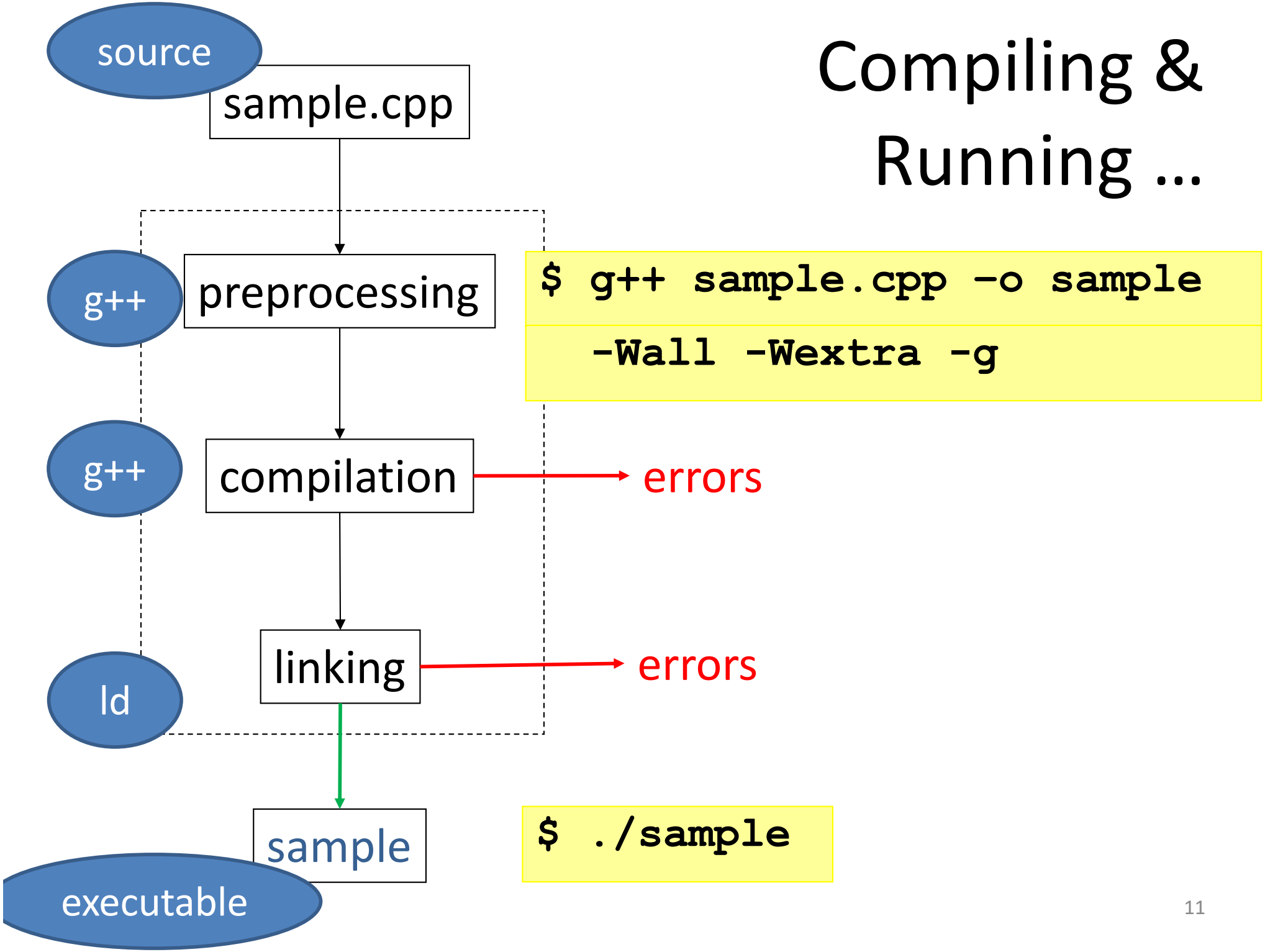


System.out.println

Insertion operator – « put to »

Other streams : std::cin,std:: cerr, std::clog …

Compiling & Running …




```
$ g++ sample.cpp -o sample
    -Wall -Wextra -g
```

```
$ ./sample
```

Comments

- One line

```
// end of the line
```

- Legacy C comment

```
/* commented block */
```

  - Not nestable

```
/*
   code … // first
   code … // second
*/
```

  - Tool ?



Overloading (1)

- To realize an action with different parameters

```
void display (something); // ?
```

```
void display_int     (int     i);
void display_double  (double d);
void display_pointer (void * p);
```

C

```
int dummy  = 2;
display_int     (dummy);
display_double  (dummy * 50.);
display_pointer (&dummy);
```

Overloading (2)

- Weak form of polymorphism
  - ⇨Functions with identical names but different signatures

```
void display (int    i);
void display (double d);
void display (void * p);
```

C++

```
int dummy = 2;
display (dummy);
display (dummy * 50.);
display (&dummy);
```

Parameters with default value

Omitting a regular value

- Declaration

```
void display(string txt, int col = BLUE);
```

```
void display(string txt = "", int coul);
```

- Definition / implementation

```
void display(string txt, int coul)  {
    // code
}
```

- Appel

```
display("loic");
display("loic", RED);
```



Mute parameter

- A parameter with no name
  - Compatibility with another version
  - Avoiding warning compilation
  - Convention respect (eg main)
- Syntax

```
type function(type1 p1, type, type2 p2);
```

- Example

```
int main(int argc, char ** argv)
```

```
int main(int, char **)
```

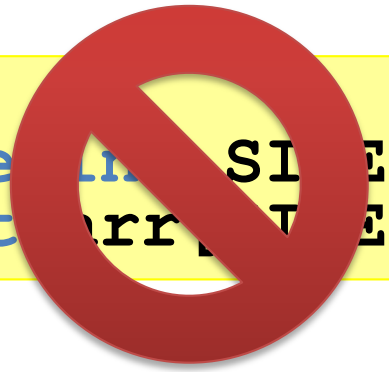


main() is a function !

Litteral constant

C

```
#de   SI  E 10
int  rr[  E];
```



```
enum {SIZE=10};

int arr[SIZE];
```

- Fast
- No type checking
- Int only

C++

```
const int SIZE=10;

int arr[SIZE];
```

- Use of a more powerful const
- No name conflict
- Easy to add/remove
- Any type
- Type checking



Const looks like final

Reminder : *const* à la C



« I promise not to modify »

```
const int * t1;
int const * t2;
int * const t3;
const int * const t4;
int const * const t5;
```

Mandatory initialization

const is applied on what stands before it

Reference (1)

- Parameter by value in C

out or *in / out* *parameter*
Use of a POINTER

Not readable code

Error prone

- Reference
  - Hidden pointer
  - Simulates parameter by reference
  - Used as a variable



Reference (2)

```
#include<iostream>

void f1(int b) {
  b = 5;
}

void f2(int *p) {
  *p = 7;
}

void f1(int& c) {
  c = 9;
}
```

Regular parameter

A copy

Pointer as parameter

A reference !!!

An alias

```
int main(int, char **)
{
  int a = 3;
  f1(a);
  std::cout << a;

  f2(&a);
  std::cout << a;

  f3(a);
  std::cout << a;

  return 0;
}
```

No change

Updated - complicated ?

Updated

Reference (3)

```
void swap(int *a, int *b)
{
  int c=*b;
  *b=*a;
  *a=c;
}

int main(int, char **)
{
  int i=5;
  int j=6;

  swap (&i, &j);

  return 0;
}
```

```
void swap(int &a, int &b)
{
  int c=b;
  b=a;
  a=c;
}

int main(int, char **)
{
  int i=5;
  int j=6;

  swap (i, j);

  return 0;
}
```

Reference (4)

Efficiency

Clearer code

Less error

Easier call

Easy to understand ?

Ambiguous syntax &

Pointer ?

A reference is ALWAYS linked to a variable

The reference NULL does not exist



Dynamic memory allocation

```
int * p = (int *) malloc(10 * sizeof(int));

free(p);
```

C

```
double * d = new double;
double * n = nullptr;  //

delete   d;
```



C++

```
int     *  p = new int[10];

delete [] p;
```

What to do if impossible ?
Stay tune

Type inference




```
int    i1 = 10;
auto   i2 = 5;
```

```
auto   y = f(x);
```

Easy

2 meanings for *auto*, before and after 2011

Do not overuse

Weird syntax

Newbie



FUNDAMENTALS



ISIMA

Object Oriented Paradigm in C++



- How to write a class
- Relationship between classes
  - Aggregation
  - Inheritance
  - Association
- Exceptions
- Genericity (not truly OOP but …)
  - Functions
  - Classes

How to write a class



- Basics syntax
- Declaration
  - Attributes and methods prototypes
- Methods definition/implementation
- Life cycle of objects
  - Construction
  - Method calling
  - Destruction

Basics (0)

Sample
+ display()

```
#include <iostream>

class Sample {
  public:
  void display() {
     std::cout << "hello" << std::endl;
  }
};

int main(int, char **)
{
  Sample s;

  s.display();

  return 0;
}
```

MANDATORY

(File Sample.cpp)

Instanciation of s

Call of display()

Class in C++

```
class Point
{
 public:
    double getX();
    double getY();
    void setX(double);
    void setY(double);
    void moveFrom (double, double);
    void moveTo   (double, double);

 private:
    double x;
    double y;
};
```

MANDATORY

Point
-x : real -y : real - counter : integer
+ getX() : real + setX(real) + moveFrom(dx : real, dy : real) + moveTo(px : real, px : real) + getCounter() : integer

STYLE GUIDE ?

Declaration
(File Point.hpp)

```
// Getter
double Point::getX()
{
   return x;
}
```

```
// Setter
void Point::setX(double px)
{
   x = px;
}
```

```
// method
void Point::moveFrom(double dx, double dy)
{
   x+=dx;
   y+=dy;
}
```

Definition
(File Point.cpp)

```
// method
void Point::moveTo(double px, double py)
{
   setX(px);
   setY(py);
}
```

Encapsulation

- Attribute / data member / property
- is PRIVATE => you don’t want the outside knowing about it
- But you agree on access :
- Methods to get access to it => set / get
- You can control the access

- Member funtions / methods
- => the outside knows that the method exist but you don’t give a complete access, you don’t give access to the code

- Array with values => 1, 8, 13, 4, …
- Sort(array)



Basics (1)

```
#include <iostream>

int main(int, char **)
{

   Point p;

   std::cout << p.getX() << " ";
   std::cout << p.getY() << std::endl;

   p.moveTo(2., 4. );

   return 0;
}
```

Instanciation

2 ways of compiling…

- Compile all cpp files together

```
$ g++ Point.cpp main.cpp -o main
```

NEVER compile a header file (hpp)

- Compile each cpp file

```
$ g++ -c Point.cpp
$ g++ -c main.cpp
$ g++ Point.o main.o -o main
```

Use a makefile





Constructor

- To initialize attributes
  - Name of the class
  - No return type
  - **Default version if not written**

```
class Point
{
 public:
    Point();
    Point(double, double);
};
```

```
Point()
{
   x = y = .0;
}
```

Can be overloaded

```
Point(double = .0, double = .0);
```

Basics (2)

```
#include <iostream>

using namespace std::cout;
using std::cout;

int main(int, char **)
{
   Point   p1;
   Point   p2(5., 5.);
   Point * pp = new Point(10.,10.);

   cout << (*pp).getX() << endl;
   cout << pp->getX()   << endl;

   delete pp;

   return 0;
}
```

Allocates memory

Frees memory

Cooking (1)

Recipe
-eggs : integer -flour: real -sugar:real -butter:real
+ constructor() ? + setEggs(int) + setFlour(real) + setSugar(real) + setButter(real) +bake(int, int) +display();



```
int main(int, char**)  {
  Recipe cake;

  cake.display();
  // nothing to prepare
  cake.setEggs(4);       // number of eaggs
  cake.setButter(240);  // weight
  cake.setFlour(240);
  cake.setSugar(240);
  cake.display();
  // Eggs : 4, Flour : 240, …
  cake.bake(180, 40);  // temperature, time
  cake.display();
  // Recipe is ready to eat
}
```

Class members

Point
-x : real -y : real - counter : integer
+ getX() : real + setX(real) + moveFrom(dx : real, dy : real) + moveTo(px : real, px : real) + getCounter() : integer

```
class Point
{
 public:
    Point(int, int);
    // …
    static int getCounter();

 private:
    double x;
    double y;
    static int counter;

};
```

keyword **static**
Class member

```
// Constructor
Point::Point(double px, double py)
{
   x=px;
   y=py;
   ++counter;
}

// RO attribute
// (from outside)
int Point::getCounter()
{
   // cout << getX();
   return counter;
}
```

Definition
(File cpp)

REMINDER : instance attributes are not available

static is not specified here

```
// initialization
int Point::counter = 0;
```



Basics (3)

```
#include <iostream>
using namespace std;

int main(int, char **)
{
  // to call a class method…
  cout << Point::getCounter() << endl;

  Point p(10., 10.);
  // ambiguous but usable syntax
  cout << p.getCounter() <<  end;

  return 0;
}
```

Mores classes ?



Person.hpp

```
class Person {

};
```

main.cpp

```
int main(int, char**) {
}
```

Point.hpp

```
class Point {

};
```

#include

#include

#include

#include

Person.cpp

Point.cpp

Declaration

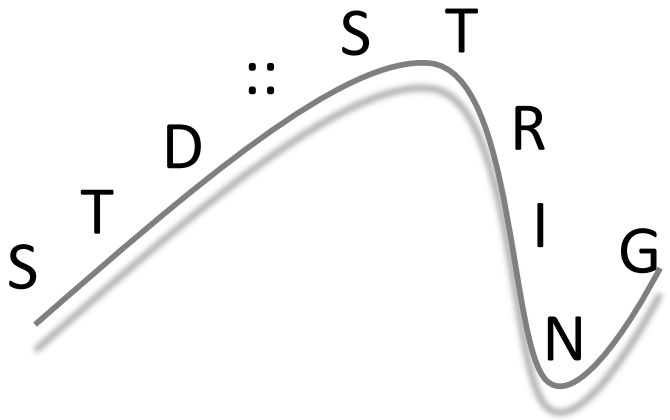
Definition
Implementation

Person

Point

Predefined class : std::string

- Header : string
- A string of characters
  - Encapsulation of C string
  - Others types (eg UTF-8)
- A lot of stuff …
- Intuitive handling



Usual operations

  - We will learn how to program them

Complete samples on https://perso.isima.fr/loic/cpp/string.php

Instanciation

```
std::string   s1;
std::string   s2("string 2");
std::string   s3 = "string 3";
```

Standard streams

```
std::cout << s2;
std::cin  >> s3;
getline(std::cin, s1);
```

Redirection operators

Array of char

```
std::cout << s2[3];
std::cout << s2.at(3);
s2[2]     = 'A';
s2.at(3) = 'B';
```

[] operator

Tests

```
if (s2==s3) cout << "equal" ;
if (s2!=s3) cout << "different" ;
if (s2< s3) cout << "smaller";
if (s2.compare(s3)==0) cout << "equal" ;
```

Concatenation

```
s2 += s3;
s2.append(s3);
```

```
s1.empty() ?   s1.length()
s1.clear()s1.size()
```

C string

```
s2.c_str();
```

```
insert/erase, replace,
substr, find, …
```

Implicit cast ?

```
const char *
```

```
"exemple"
```

```
char *
```

```
char [256]
```

```
std::string
```



Streams

- In / Out streams handling
- Source or destination of characters

- Standard stream `cin cout cerr clog`
- File streams `fstream`
- String streams `stringstream`

Handling with practical work



Cast

```
#include <sstream>
```

C functions can be used

- String stream

```
std::ostringstream oss;
oss << value;

Result : oss.str();
```

10.5 -> "10.5"

```
std::istringstream iss(string_to_decypher);
iss >> integer ;
```

"8" -> 8

Encapsulation



Point
~~-x : real~~ ~~-y : real~~ - r : real - t : real
+ Point() + getX() : real + setX(real) + getY() : real + setY(real)

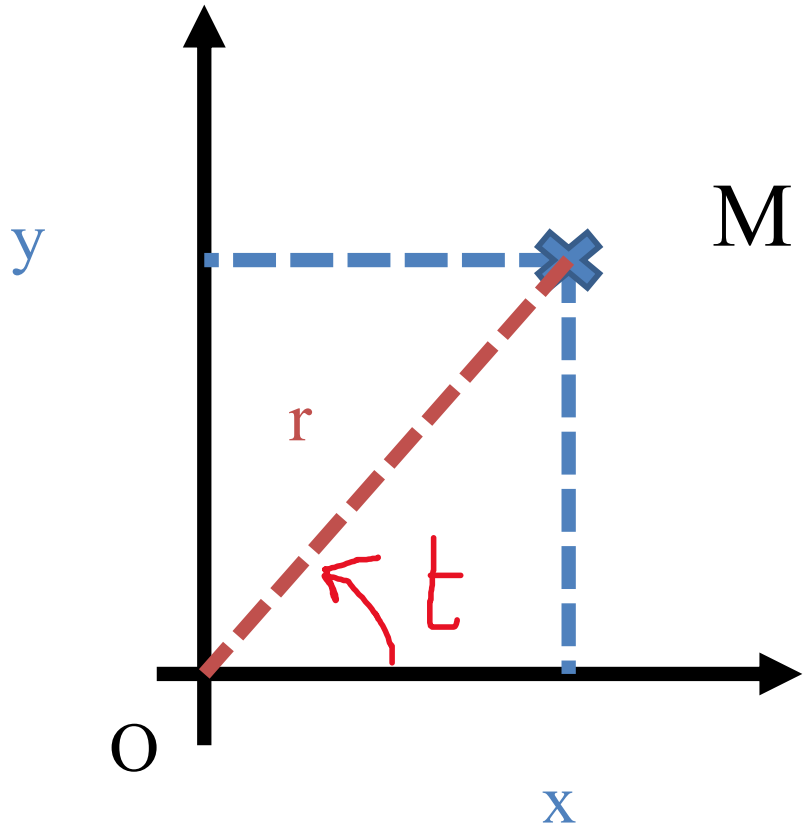
No change interface

Implementation change

Hidden outside the class

```
// Getter, get method
double Point::getX()
{
  return x;
}
```

```
// Getter, get method
double Point::getX()
{
  return r*cos(t);
}
```



M(x, y) - cartesian

M(r, t) - polar

Objets Lifecycle

①Construction

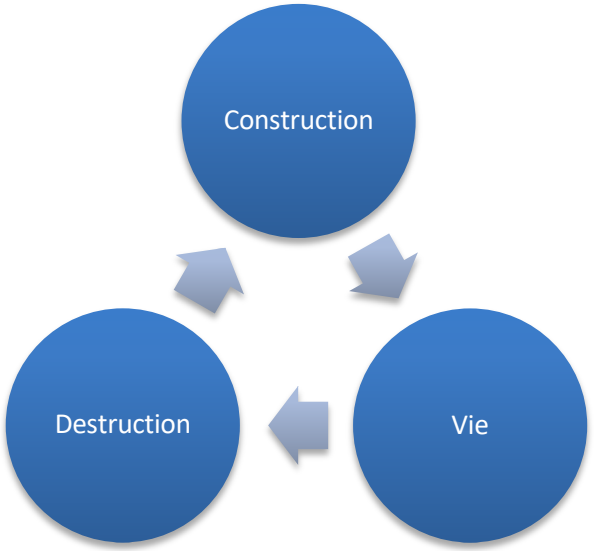
- Memory allocation
- Constructor call

②Life

- Messages passing

③Destruction

- Destructor call
- Memory freed



Constructor : initialization list

```
Point::Point(double px, double py) :
    x(px), y(py)
{
    // …
}
```

Initialization list

- Flexible
  - Constants or immediate values
  - Functions
  - Arithmetical expressions
- Respect the class declaration order
- Evaluation before the code

Initialization order

```
class Rational
{
  public:
    Rational(int n=0, int d=1):
          deno(d),      nume(n),
          nume(n)       deno(d)
    {}

  private:
    int nume;
    int deno;
};
```

Warning
-Wreorder



Complex expression

- Computed attribute : distance

```
Point::Point(double px, double py):
                x(px), y(py) {
  distance = sqrt(x*x+y*y);
}
```

```
Point::Point(double px, double py):
                x(px), y(py),
                distance(sqrt(px*px+py*py)) {
}
```

Construction ?

```
class Point
{
 public:
    // No other ctor
    Point(double, double);
};
```

What if the ctor is not public ?

```
int main(int, char**)
{
   Point p1(3., .4);
   Point p;
   Point * a = new Point[7];
   return 0;
}
```

ERROR : ctor not found

Instanciation of 7 objects

```
void Point::moveTo(double x, double)
{
      cout << x;
      cout << this->x;
}
```

> Compilation ✓
> Running ✓
> ?

```
Point::Point(double x, double y)
{
      this->x = x;
      this->y = y;
}
```



this

```
Point::Point(double x, double y):x(x), y(y)
{
   //x ?
}
```

Defined in the norm

List & constructor

- Call to a constructor from the same class

```
Point::Point(double px, double py):
                x(px), y(py) {
    // ++counter;
}

Point::Point():Point(0.0, 0.0) {
}
```

Avoid code duplication

Inspired from other langugages



Reminder : allocation types

"automatic" destruction

1. Static
   - Global variables
   - Local variables declared as static
2. Automatic
   - local variables on the stack
   - parameter
3. Dynamic : variables allocated on the heap
   - Allocation with **`new`** + ctor
   - Destruction with **`delete or delete[]`**



```
#define T 3
int i = T;
const double PI = 3.14;


int function(int j)
{
    static int k = 0;
    int e        = 7;
    k += j;
    cout << ++i << j << k << ++e;
    return k;
}

int main(int argc, char **)
{
    int l = argc + 1;
    double * m = new double[function(l)];
    function(6);
    delete [] m;
    return 0;
}
```

Macros usage discouraged in C++

Constant

```
class Array
{
    int * arr;
    int   size;

 public:
    Array():arr(nullptr), size(10)
    {
        arr = new int[size]; // if not possible ?
    }

};

int main(int, char **)
{
    Array a;

    return 0;
}
```

> Compilation ✓
> Running ✓
>

Valgrind ?



Destructor ?

- Name of the class with a ~
- Never explicitely called
  - Automatic destruction
  - Called by `delete`
- Frees the critical resources



```
class Array
{
 public:
    Array();
   ~Array();
};
```

```
Array::~Array()
{
    if (arr) // opt
      delete [] arr;
};
```

TP

Destruction order checking

```
int function (int p)
{
    int r = p * p;

    return r;
}
```

```
b = function (a);
```

Copy or not copy

```
int  function (int& p)
{
    int r = p *p;

    return r;
}
```

```
b = function (a);
```



Copy preventing

```
void function (Point& p)
{
    p.setX(5.);
    cout << p.getX() << " ";
}

int main(int, char **)
{
    Point p(2.0, 3.0);

    function(p);

    cout << p.getX();
    return 0;
}
```

```
> 5.0 2.0
>
```

1. Copy
2. Known ctors not called

Overview

```
const int SIZE = 10;

int main(int, char **)
{
   Point p1(12., 15.);
   Point p2;
   Point p3[SIZE];
   Point p4();

   Point *p;
   p = new Point(10.,25.);
   delete p;

   return 0;
};
```

Not an object

```
Point p6 { 1., 2. };
Point p7 {};
```

Unit testing



- To test small pieces of code
  - Before integrating them to the whole program
- By hand or **automatic ?**
  - Can you repeat ?
  - Regression ?
- More or less complete
- More or less realistic
- Test Driven Development (TDD)

Factorial computing

$$\begin{cases} n! = n * (n-1)! \\ 0! = 1 \end{cases}$$

```
 0! = 1
 3! = 6
 7! = 5040
13! = 6227020800
```

```
TEST_CASE("factorial")
{
    REQUIRE(    1 == f(0) );
    CHECK   (    6 == f(3) );
    CHECK   ( 5040 == f(7) );
}
```



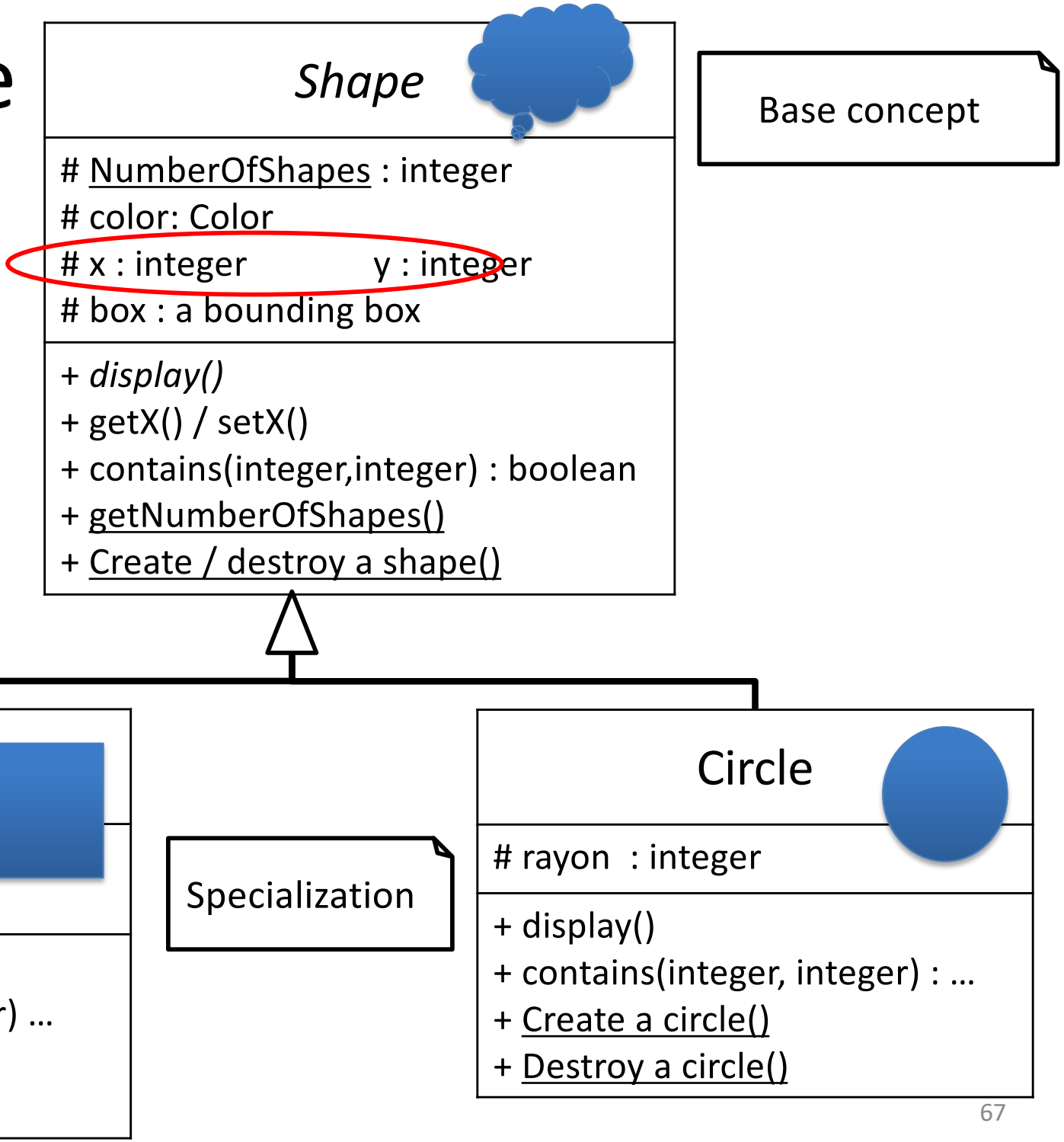
```
int fact(int n) {
    int r = 1;

    if (n>1)
        r = n * fact(n-1);
    return r;
}
```

```
int f(int x)
{
    int r = 1;

    if (x==3)
       r=6;
    else if (x==7)
       r = 5040;

    return r;
}
```

```
 0! = 1
 3! = 6
 7! = 5040
13! = 6227020800
```

Inheritance

Class Point ?



Inheritance

- Simple or multiple
- Member visibility
- Several inheritance types

} public, protected, private

```
class Rectangle : public Shape
{

};
```

Simple public inheritance

Visibility / Access

- 3 levels

Mother member	**Daughter**	**Outside**
public	✓	✓
protected	✓	✗
private	✗	✗

- Classical inheritance : *protected*
- All the members from *private* to *protected* ?



```
class Parent {
 private:
  ✗int purse;
 protected:
  ✓S s; // shoe

 public:
    void sayHello();
    C    seeShoe();
    int  askForMoney(int);

 protected:
  ✓int  seeMoney();
  ✓void changeShoe(C);

 private:
  ✗void takeMoney(int);
};
```



```
class Child:
    public Parent
{
};
```



Objet instanciation

```
Circle::Circle()
{
}
```

Implicit call

1. Shape creation
2. Circle creation

```
Circle::Circle():Shape()
{
}
```

```
Circle c;
```

```
Circle c(1,2);
```

```
Circle::Circle(int px, int py):
    Shape(px,py), radius(0)
{
}
```

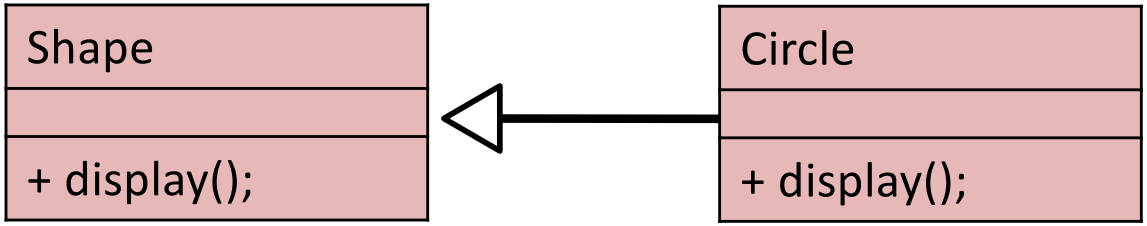
Reverse destruction : Circle then Shape

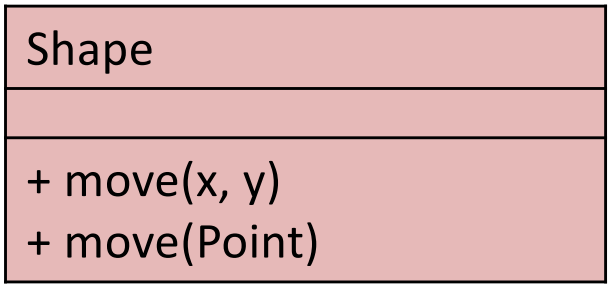
Polymorphism

- The same method can have multi shape
- Weak version
  - Overloading – *Compile Time*
  - Methods with different signatures

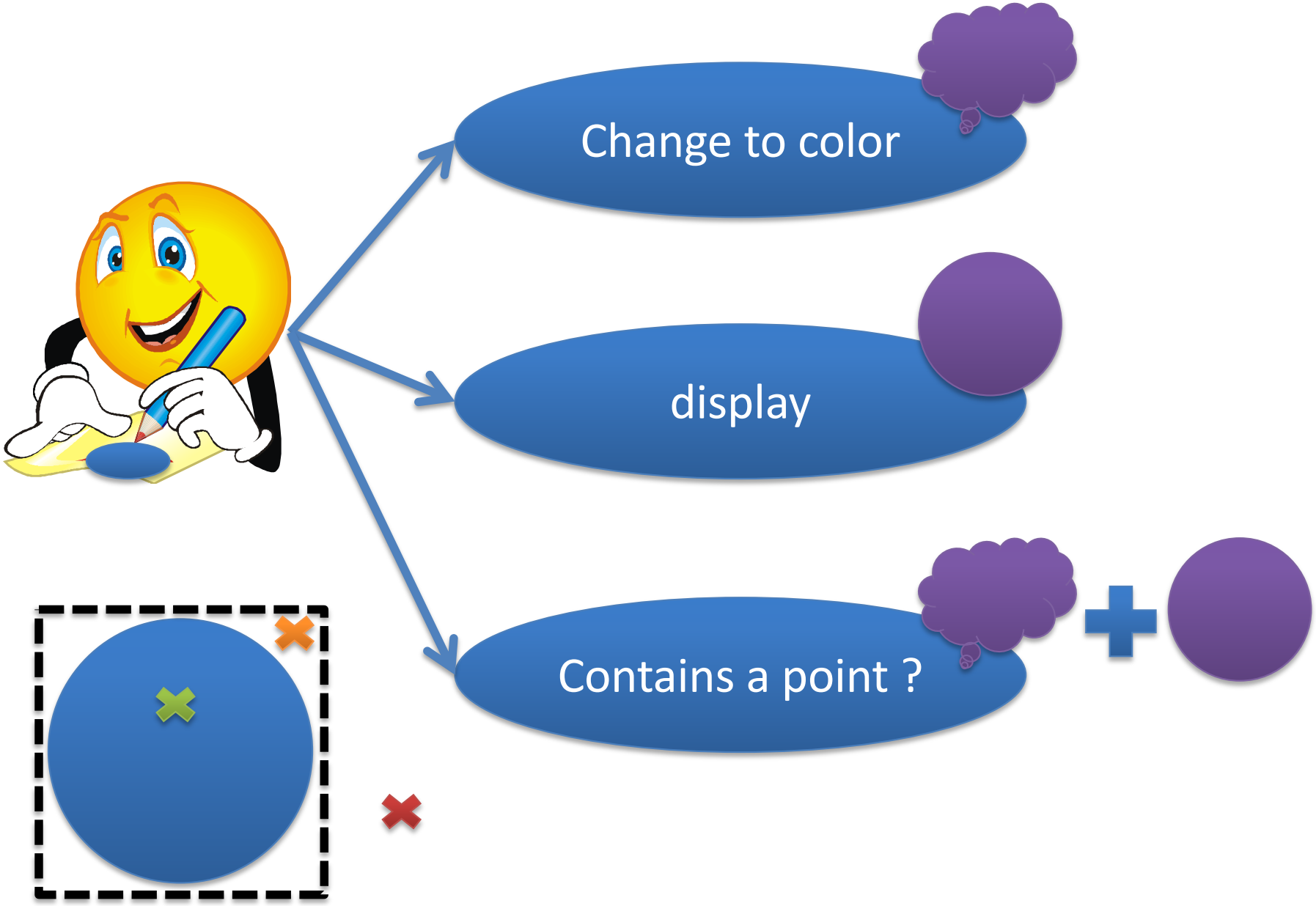


- Strong version
  - Overriding – *Runtime / dynamic*
  - Different behaviors along hierarchy





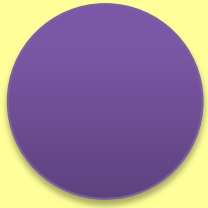
Circle & polymorphism



```
bool Shape::contains(int x, int y)
{
  // computing the bounding box
  return r;
}
```



```
bool Circle::contains(int x, int y)
{
  bool r = Shape::contains(x,y);

  if (r)
  {
     // more precise computation
  }
  return r;
}
```

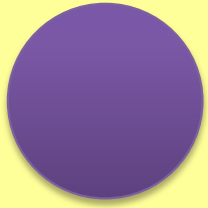


bool : true or false

```
void Shape::display() {
   cout << "Shape" << endl;
}
```



```
void Circle::display() {
   cout << "Circle" << endl;
}
```

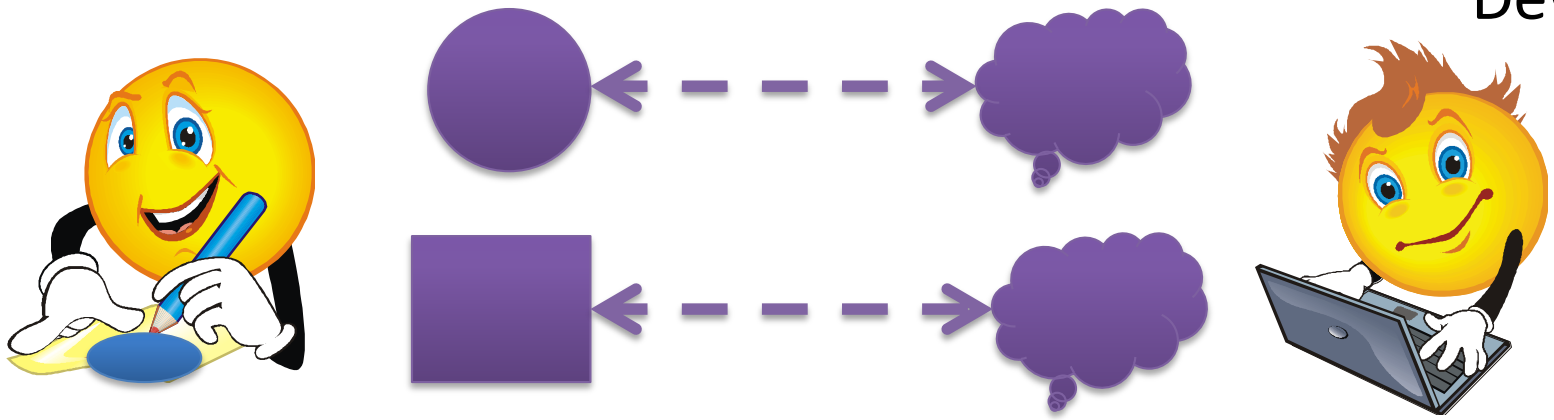


```
int main(int, char**) {
   Shape  s;
   Circle c;

   s.display();
   c.display();
   return 0;
}
```



User

Developer



```
int main(int, char **) {
   Shape  * s = new Circle;

   s->setColor(2);

   s->setRadius(10.0);


   s->display();

   return 0;
}
```

the circle is a Shape

OK

The shape is not a circle

displays « Shape » !!!

Virtual method

- Dynamic / Runtine choice
  - Virtual methods table
  - Overhead
- « Virtual one day, virtual always »





```
class Shape {
    virtual void display();
};
```

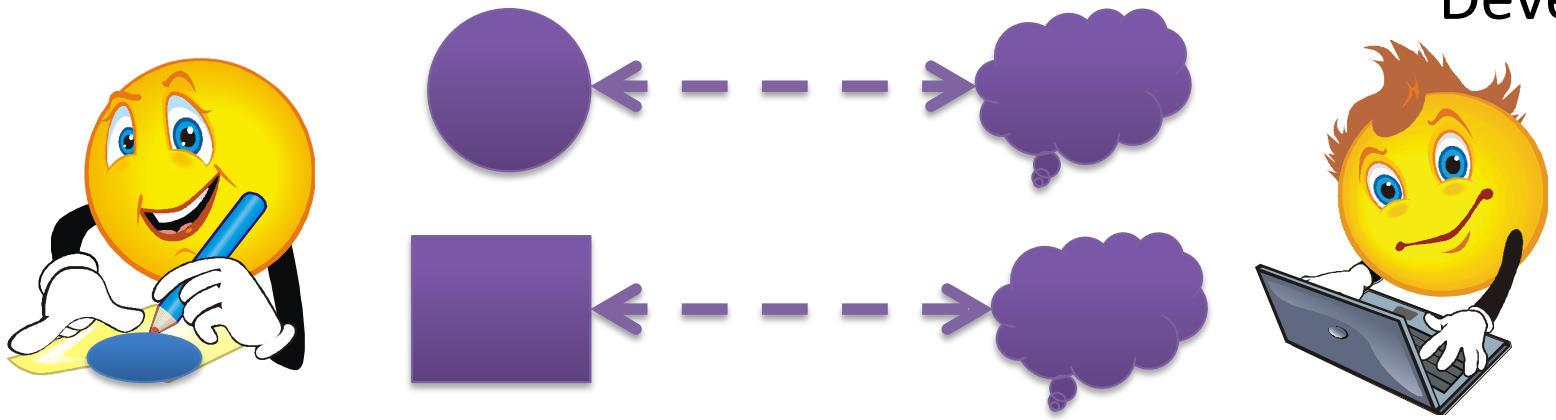
2011

```
class Circle : public Shape {
    [virtual] void display() [override];
};
```

Any class with a vitual method
SHOULD have a virtual destructor

User

Developer



```
Shape    a1[100000];
Shape * a2[100000];
```

```
a1[0] = Circle;
a1[0].display();
```

Adding a triangle
= no impact



```
a2[0] = new Circle;
a2[1] = new Rectangle;
// …

for(int i=0; i<number; ++i)
    a2[i]->display();
```

Downcasting

- Base class ➡ specialized class
  - Upcasting : no problem



Expensive if checking

```
int main(int, char **) {
 Shape  * s = new Circle();
 s->setRadius(2.0)
 ((Circle    *)s)->setRadius(2.0);
 ((Rectangle *)s)->setRadius(2.0);
 ((Rectangle *)s)->setHeight(5.0);
  s->display();
  return 0;
}
```

C	E
✗	
✓	✓
✗	
✓	✗

- Runtine checking
  - Dedicated operators to avoid inconstency (not seen)

Abstract class

- Very general concept
- Cannot create instance

cannot declare variable ‘x’ to be of abstract type ‘X’

- At least an abstract method
  - « no code »
  - Declared as pure virtal

```
virtual return_type method(parameters) = 0;
```

  - A default code can be provided to derived classes

```
virtual bool contains(int, int) = 0;
```

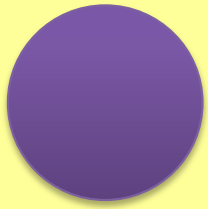
```
bool Shape::contains(int px, int py) {
  bool r = false;
  if (px>=x) && (px<=x+width) &&
     (py>=y) && (py<=y+height))
     r = true;
  return r;
}
```



```
bool Rectangle::contains(int px, int py) {
  return Shape::contains(px,py);
}
```



```
bool Circle::contains(int px, int py) {
  return r; // see slide 60
}
```



Interface

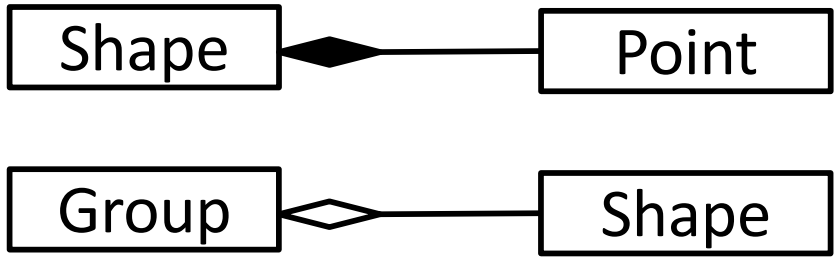
- Does not exist in C++
- Pure abstract class
  - No method with code
  - Even destructor
- Attributes ?
  - No but …
  - Constants ?



Multiple inheritance

Aggregation / Composition

- One or several objects in another
- Implementation
  - Direct Objet
  - Reference
  - Pointer



- Who constructs the objet ?
  Where does it come from ?
  - Object instancied by outer object
  - Object from outside
    - Copied
    - Pointer or reference

To come

```
class Shape
{
   Point point;
};
```



```
Shape::Shape():point()
{
}
```

Optional

```
Shape::Shape(Point pp):point(pp)
{
}
```

```
Shape::Shape(int px, int py):point(px, py)
{
}
```

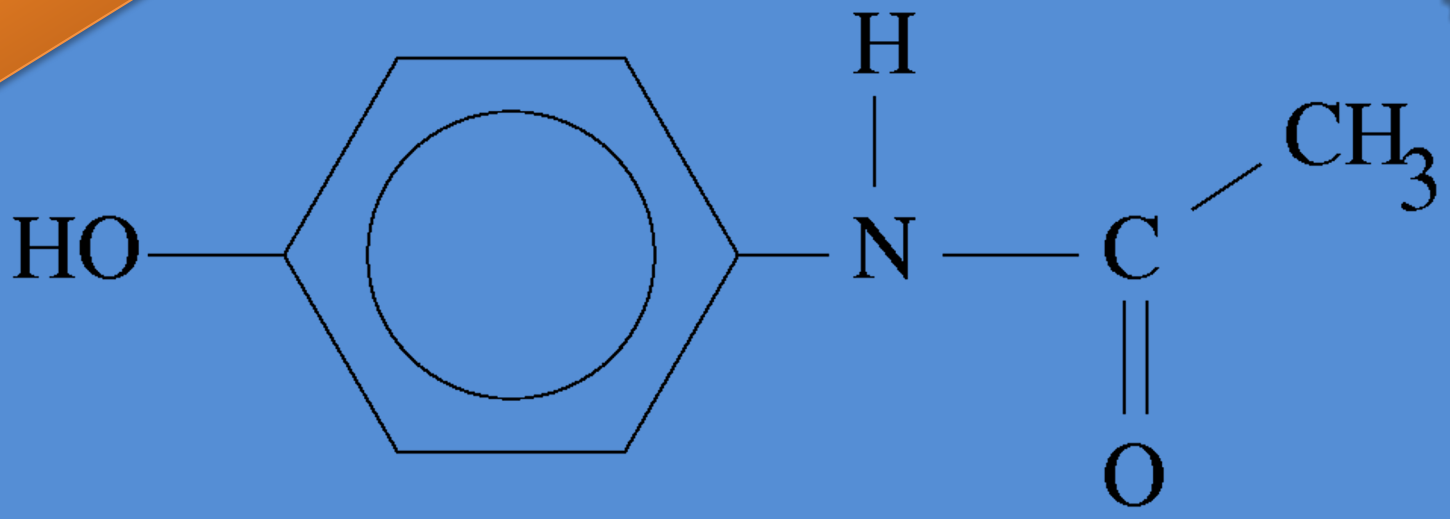
Composition : the point is destructed with the shape

Other relationships between objects

- Same implementation as aggregation/composition
  - Object
  - Reference
  - Pointer
- Who deals with the destruction ?
  - Owner ?
  - Be careful to non automatic objects

Significant names

SENSIBLE



SUBTLETIES C++



ISIMA

Constant Object

```
const Rectangle r(0, 0, 10, 20);
```

```
r.setWidth(40);
```

error: no matching function for call to ‘Rectangle::setWidth(int) const’
note: candidates are: void Rectangle::setWidth(int) <near match>

Well, OK, the method updates the state of the object

```
int l = r.getWidth();
```

!! Same thing ??



Constant method

- Callable on a constant object

```
void Shape::display() const
{
    std::cout << "Shape" << std::endl;
}
```

- `Const` **keyword**
  - Insures that the method does not modify the object state
  - Belongs to prototype
  - Used whenever it is possible
  - « Constness »

```
void Rectangle::setWidth(int l) const {
  width = l;
}
```

error: assignment of data-member ‘Rectangle::width’ in read-only structure

```
int Rectangle::getWidth() const {
  std::cout << "const ---" << std::endl;
  return width;
}
int Rectangle::getWidth()          {
  std::cout << "non const" << std::endl;
  return width;
}
```

OPTIMIZATION ?

```
const Rectangle r1;
Rectangle       r2;
r1.getWidth();
r2.getWidth();
```

```
> const ---
  non const
>
```

const int & ou int

```
int & Rectangle::getWidth() const {
   std::cout << "const ---" << std::endl;
   return width;
}
```

error: invalid initialization of reference of type ‘int&’ from expression of type ‘const int’

getter optimisation ?

```
int & Rectangle::getWidth() {
   std::cout << "non const" << std::endl;
   return width;
}
```

```
cout << r.getWidth();
```



```
r.getWidth() = 3;
```

CPP *old style*

```
class NinjaWarrior
{
    double x_;

 public:
    double & x() {
        return x_;
    }
};

int main(int, char **)
{
    NinjaWarrior p;
    cout << p.x() << " ";
    p.x() = 7;
    cout << p.x() << " ";
    return 0;
}
```

```
> 0.0 7.0
>
```

Valid return
Updatable attribute

References Return

```
int function (int p)
{
    int r = p * p;

    return r;
}
```

```
b = function (a);
```



warning: reference to local variable ‘r’ returned

```
int& function (int& p)
{
    int r = p * p;

    return r;
}
```

```
b = function (a);
```

References & immediate values

```
int function (int & p)
{
    // …
    return r;
}
```

error: invalid initialization of non-const reference of type ‘int&’ from a temporary of type ‘int’

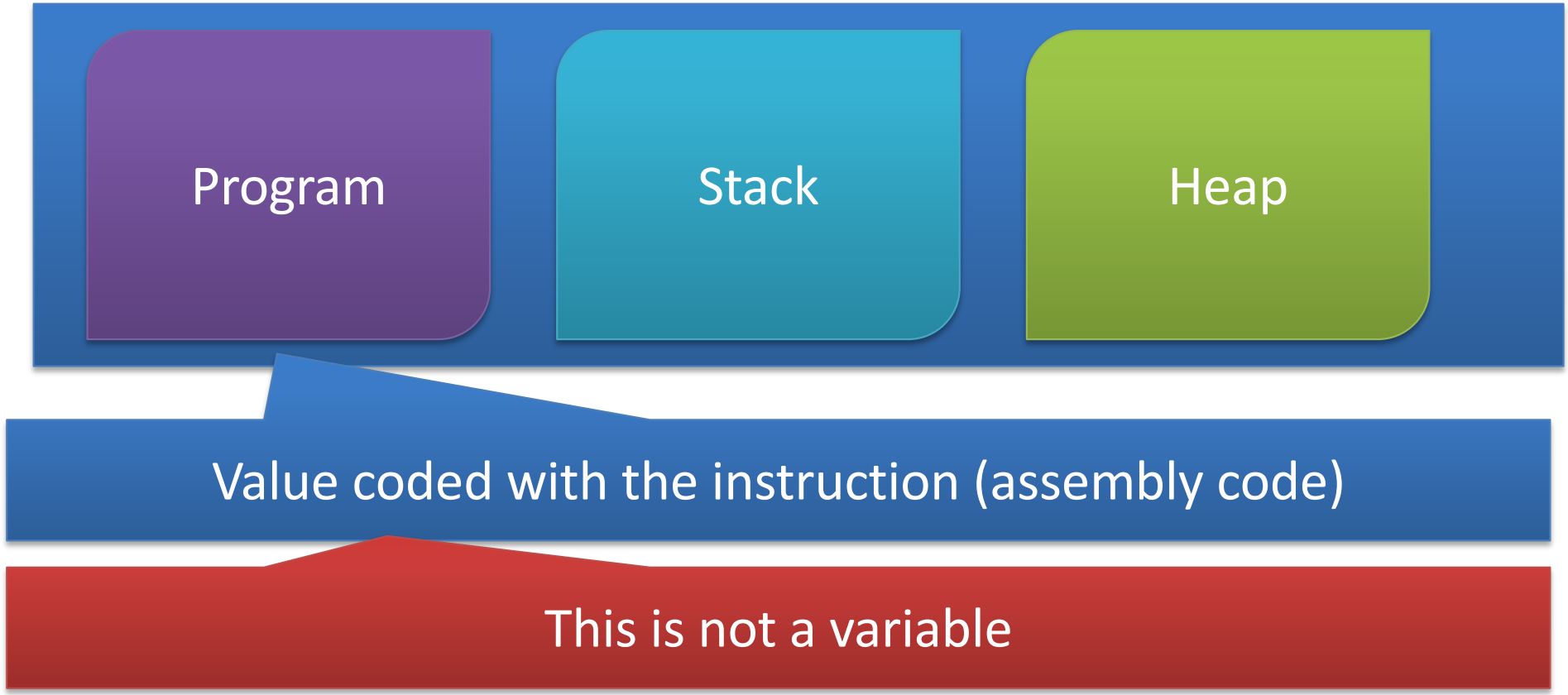
```
int i = 5;
function(i);

function(10);
```

Immediate value

```
int function (const int & p)
{
    // …

    return r;
}
```

Where are stored the immediate values ?

Memory




```
function(10);
```

```
int function (const int & p)
```

References & Overloading ?

```
class NinjaWarrior
{
   double x_;

 public:
   double & x() // SET ?
   {
      return x_;
   }

   double x()    // GET ?
   {
      return x_;
   }
};
```

error: ‘double NinjaW::x()’ cannot be overloaded



```
class Shape {
  public:
   virtual void display() { cout << "Shape"; }
};
```

```
class Circle : public Shape {
  public:
   virtual void display() { cout << "Circle"; }
};
```



```
void display1(Shape f) {
    f.display();
}
```

```
void display2(Shape & f) {
    f.display();
}
```

```
int main(int, char**) {
    Circle circle;

    display1(circle);
    display2(circle);

    return 0;
}
```

```
> Shape  Circle
>
```

Type truncating

```
void function (Point  pp)
{
    cout << pp.getX() << " ";
    pp.setX(5.);
    cout << pp.getX() << " ";
}

int main(int, char **)
{
    Point p(2.0, 3.0);

    function(p);

    cout << p.getX();
    return 0;
}
```

```
> 2.0 5.0 2.0
>
```

1 usual construction
2 destructions
➔ pp has to be a copy



Copy constructor (1)

- Creation of a new object as a clone / copy
  - Explicit
  - Parameter by value
  - Return of a function (compilo ?)

```
Circle c1;
Circle c2(c1);
Circle c3 = c1;
```

```
Circle::Circle(Circle) {
}
```

```
Circle::Circle(const Circle &) {
}
```

```
Circle::Circle(Circle &) {
}
```

WARNING : No recursive definition



Copy constructor (2)

Copy with a ctor call

```
class Shape {
    Point p;
    int   couleur;
};
```

Binary copy
Shallow copy

Double delete of pointers



Copy & *compagreg*

PW

- The default copy constructor calls the default copy constructors of nested objects
- Otherwise the default constructor is called

```
Shape::Shape(const Shape & f) {
   //
}
```

```
Shape::Shape(const Shape & f):
point(f.point) {
   //
}
```





Copy & inheritance

PW

```
class Circle : public Shape {
};
```

Copy by call of a constructor

The default copy constructor of the daughter class calls the copy constructor of the mother

Call of the default constructor otherwise

```
Circle::Circle(const Circle & c):
Shape(c) {
   // some code
}
```

Assignment

```
Circle c1(3.0);
Circle c2(5.0);
Circle c3;

c1 = c2;
c1.operator=(c2);

c3 = c2 = c1;
```

operator = as a method

Link possible

```
Circle& Circle::operator=(const Circle &uC) {
    if (this != &uC) {
        // …
    }
    return *this;
}
```

Avoiding c1 = c1;

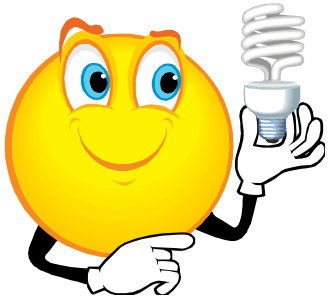
Default assignment

If you do not write the operator =

- Binary copy for non object
- Default call of the mother's operator =
- Default call of the nested objects

If you write it, write what need to be done

The current objet exists

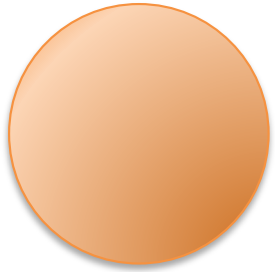


Summary : references & objects

- Always passing object parameters by reference
  - To avoid copy ctor call
  - To ensure polymorphic calls
- "const" reference for non "mutable" objects
- Do not return reference when an object is required

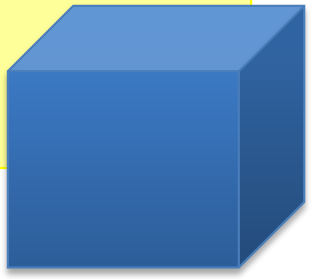
For non object – scalar – types, copy is acceptable

Cast



```
Circle c;
Rectangle r = c;
```





- Specific constructor

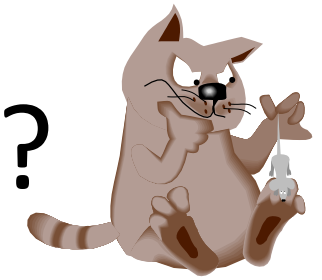
```
Rectangle::Rectangle(const Circle &) {
}
```

- Avoiding the implicit cast

```
explicit Rectangle(const Circle &);
```

```
Rectangle r = (Rectangle) c;
```

Avoiding code duplication ?



```
Circle::Circle(const Circle & c)
{
   *this = c;
}
```

Common code in a private method

Ctor delegation and initialization list

```
class GiveItATry {
    static int i ;
    string name;

 public:
    const string & getName(int j) const {
        i+=j;
        return name;
    }
   void m(const int  j) { i = j; }
   void m(       int  j) { i = j; }

   void n(const int& j) { i = j; }
   void n(       int  j) { i = j; }

   void o(const int& j) { i = j; }
   void o(       int& j) { i = j; }
};
```



A **const** method can modify a class attribute

Redefinition

Compil OK
Ambiguous call

OK

Vocabulary

- Déclaration
- Prototype

- Définition
- Implementation

Saying it exists

Creation

```
class B {
  int a;
 public:
  void m();
};

extern B b;
```

.hpp

```
void B::m() {
}

B b;
```

.cpp

B instanciation
Default initialization

Forward declaration

```
void f();

int main(int, char**) {
   f();
}

void f(){
}
```

```
class B;

class A {
   B * b;
};

class B {
   int b;
 public:
   void m() {};
};
```

Declaration of forward class

- With a pointer
- With a reference



- With an objet
- Spezialization



Uncomplete type

```
class B;

class A {
   B *   pb;
   void m1(B  & b);
   void m2(B    c);
};

class C : public B {
};

class B {
};
```

Muliple definition

extern

```
#ifndef WATCH_A
#define WATCH_A
class A {
  B * b;
};
int globale;
#endif
```

How many times A.hpp is red when compiling ?

A.cpp

A.hpp

#include

Gardiens = avoiding recursiv inclusion

main.cpp

B.cpp

B.hpp

```
#ifnde
#define WATCH_B
class B {
  A * a;
};
#endif
```

```
int main(int, char**
  A a;
  B b;
}
```

Practice



HOW TO CODE A STRING CLASS



ISIMA

Plan



- The String as an helper to learn
  - C++ subtleties
- Operators by example
  - Assignment
  - Bracket
  - Redirection
  - Increment
  - Tests

} Friendship

- Summary : the canonical form

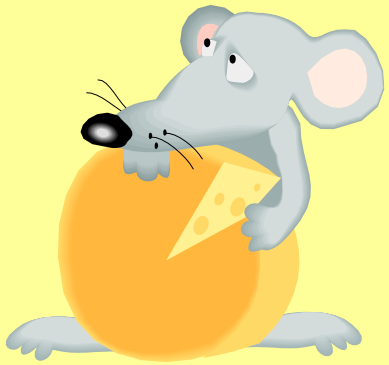
String class

- Dynamic length and size
- Encapsulates a C string
- **`cstring`** can be used

String
- capa : integer - arr : array of char
+ String() + String(size : int) + String(s : char *) + String(s : String) +~String() + assignment operator + character extraction + display + relevant accessors

```
class String {
 private:
   int     capa;
   char * arr;
 public:
    // …
};
```

```
using namespace std;

int main(int, char **) {

    String s1;
    String s2(6);
    String s3("try");
    String s4(s3);
    String s5 = s3;
    s2 = s3;
    s3.replace("done");
    s3.display(cout);
    s2.display();
    cout << s3.c_str(); // nested C string
    cout << s3[0];
    cout << s3;

    return 0;

}
```



```
String s1;
String s2(6);

s1.display(cout);
```

0 as default ?

```
String::String (int pCapa) :
    capa(pCapa>0 ? pCapa+1 : 0) {
  if (capa) {
    arr = new char [capa];
    arr[0] = 0;
  } else arr=nu
}
```

Constructor

Always possible ?

cout as default ?

```
void String::display(ostream & o) const {
   if (arr) o << arr;
}
```

arr == 0 ?

const ?

```
String::~String() {
   delete [] arr;
}
```

Destructor

```
String s3("try");
```

```
String::String (const char *cstr) {
  if (cstr) {
    capa = strlen(cstr)+1;
    arr  = new char [capa];
    strcpy(arr,cstr);
  } else {
    capa = 0;
    arr  = nullptr;
  }
}
```

Always possible ?

Another ctor

```
String s2 = "another try";
```

```
String s4(s3);
String s5 = s3;
```

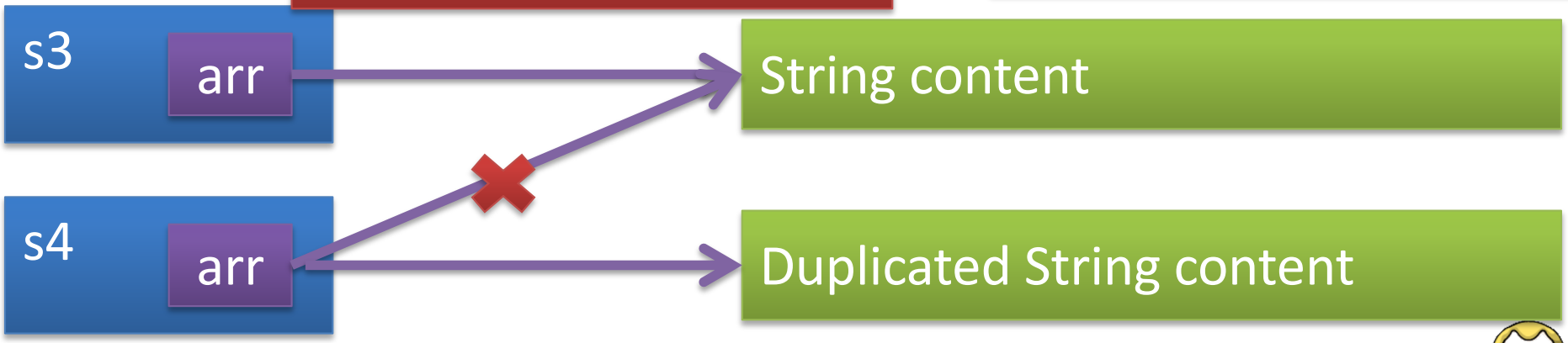


```
String::String (const String &uC) {
   capa = uC.capa;
   if (capa)    {
      arr = new char [capa];
      strcpy(arr, uC.arr);
   } else arr = nullptr;
}
```

getter not mandatory (same class)

Always possible ?

Copy constructor



```
cout << s5.c_str();
s3.replace("texte");

cout << s3.getCapacity();
```

```
const char * String::c_str() const {
  return arr;
}
```

Read only accessors

```
int String::getCapacity() const {
  return arr?capa-1:0;
}
```

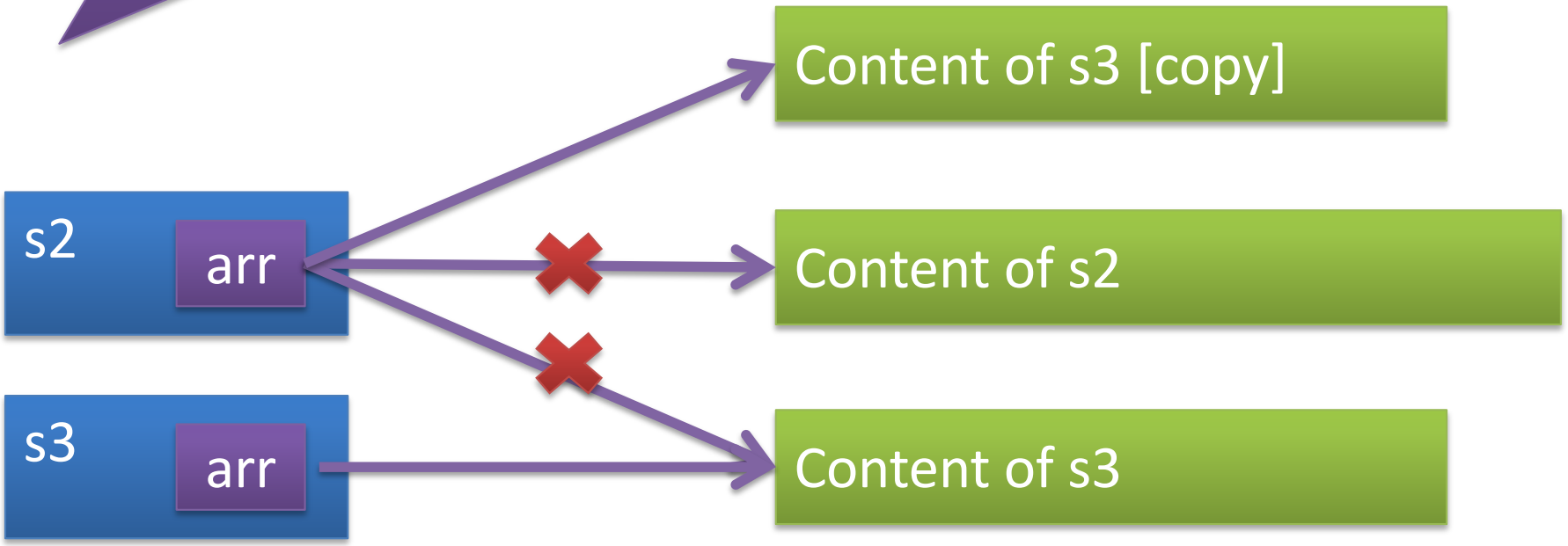
Violates the encapsulation

```
void String::replace(const char * c) {
  strncpy(arr, c, capa);
  arr[capa-1] = 0;
}
```

```
s2 = s3;
s2.operator=(s3);
```



Binary copy
- Memory leak
- Double delete at destruction



```
s2 = s3;
s2.operator=(s3);
```



```
String& String::operator=(const String &uC) {
   if (&uC != this) {
      delete []  arr;
      capa = uC.capa;
      if (capa) {
         arr=new char [capa];
         strcpy(arr, uC.arr);
      } else arr = nullptr;
   }
   return *this;
}
```

To avoid s2 = s2;

s2 exists !!

Even if arr == nullpr

Always possible ?

Assigment operator

Assigments can be chained s1 = s3 = s2

at ()

```
std::cout << s1.at(0);
s1.at(2) = 'L';
```

Allows read & write

```
char& String::at(int index)
{
    if ((index < 0) || (index >= capa)
                    || !arr ) {

      std::cerr << "ERROR index" << std::endl;
      std::exit(1); // Not so fi
    }

    return arr[index];
}
```

Seen later

```
const String s7 (s1);
std::cout << s7.at(0);
```

at ()const

Previous version not working

char

```
const char& String::at(int index) const
{
   if (all tests) {
      std::cerr << "ERROR index" << std::endl;
      std::exit(1);
   }
   return arr[index];
}
```

```
s1[2] = 'L';
std::cout << s1[0];
s1.operator[](2) = 'L';
std:cout << s1.operator[](0);
```

[ ]

Allows read & write

```
char& String::operator[](int index)
{
   if (all tests) {
      std::cerr << "ERROR index" << std::endl;
      std::exit(1);
   }
   return arr[index];
}
```

```
return at(index);
```

[ ] operator
Element Extraction
Method

```
const String s7 (s1);
std::cout << s7[0];
```

Previous version not adapted

[ ] const

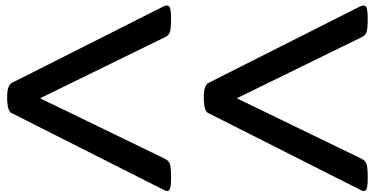
char

```
const char& String::operator[](int index) const
{
   if (all tests) {
      std::cerr << "ERROR index" << std::endl,
      std::exit(1); // Moisi …
   }
   return arr[index];
}
```

```
return at(index);
```

Const [ ] operator

```
cout << s1;
operator<<(cout, s1);
```



Operator can be chained

```
ostream& operator<<(ostream &o, const String &c)
{
    c.display(o);
    // OR if (c.c_str()) o << c.c_str();
    return o;
}
```

FUNCTION

c.arr not working ☺

```
cout << s1 << s2;
operator<<(operator<<(cout, s1), s2);
```

```
s1 = s2 + s3;
s1 = operator+(s2, s3);
```



FUNCTION

```
String operator+(const String &a,const String& b)
{
    char *arr =
     new char [a.getCapacity()+b.getCapacity()];
    strcpy(arr, a.c_str());
    strcat(arr, b.c_str());
    String temp(arr);
    delete [] arr;

    return temp;
}
```

Estimate

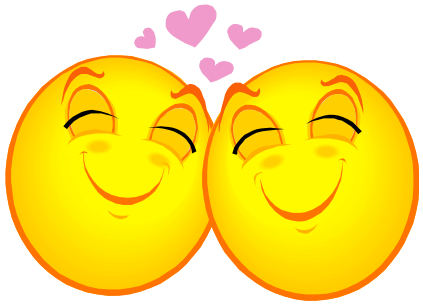
multiples copies (if no move)

Summary : operators

- Classical use of object
- All of them can be overloaded even if it can be tricky
- As a method
  - Current objet is important
  - Unary operator
  - Assignment
- As a function
  - Symmetrical use of parameters
  - Binary operators
  - When the class cannot be modified



Reference : https://en.cppreference.com/w/cpp/language/operators



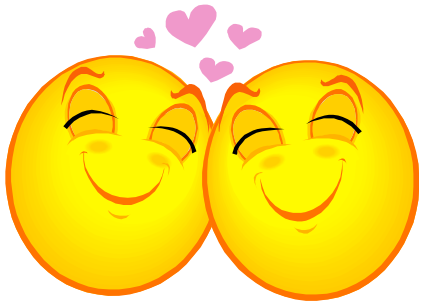
Friendship (1)

- Using private members directly
- friend class or friend function
- The class has to give the right / access
- Not transitive

```
class A
{
   …
   friend class B;
   friend void function(const A&, const C&);
};
```

A gives B the authorization to access its private members

Not the opposite : B cannot declared itself ….

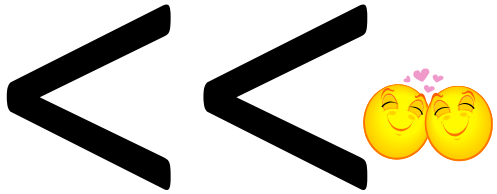


Friendship (2)

- Pros
  - Classes can be friend if they are stronly connected (like in a package)
  - Code efficiency
  - Simple ?
- Cons
  - Violates the principle of encapsulation
  - Doing anything for efficiency
  - Integrity matters when accessing attributes

Some operators with *friendship*

```
cout << s1;
operator<<(cout, s1);
```



```
class String {
    friend ostream&
        operator<<(ostream&, const String&);
};
```

Friendship declaration

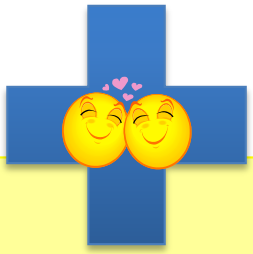
Existency declaration

```
ostream& operator<<(ostream&, const String&);
```

```
ostream& operator<<(ostream &o, const String &c)
{
  if (c.arr) o << c.arr;
  return o;
}
```

Implementation

```
s1 = s2 + s3;
```



```
class String {
    friend String
        operator+(const String&, const String&);
};
```

Modification of the class declaration

```
String operator+(const String &a,const String& b)
{
    String res(a.capa + b.capa -1);
    strcpy(res.arr, a.arr);
    strcat(res.arr, b.arr);
    return res;
}
```

Less dynamic allocations

Have to know the guts of the class



```
if (s1==s3) cout <<  "equal";
if (operator==(s1,s3))
    cout << "equal";
```

```
bool operator==
   (const String &a, const String &b) {

   return !strcmp(a.c_str(), b.c_str());
}
```

Equals operator FUNCTION

We can write all the comparison operators

Canonical Normal Form & Big 3

```
class C {
      C();

      C(const C&);

      ~C();

      C& operator=(const C&);

};
```

Construction by default

Clone
By value parameter
Return of a method

MANDATORY for complex object

To free resources

Assignment operator

Normal form

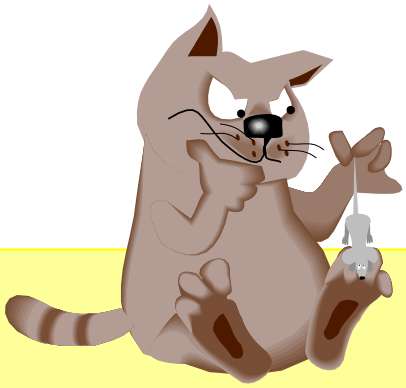


- NF 2003 ⊊ NF 2011
  - Move semantics (not seen)
  - Big 5

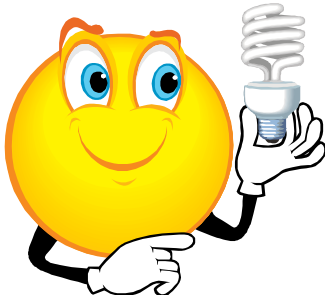
- Use of default operator or method operator
- Operator or method can be suppressed

```
class C {
    C()           = default;
    C(const C&) = deleted ;
};
```

Who's who ?



```
int main(int, char **) {
      C     c1;
      C     c2(10);
      C     c3(c1);
      C     c4 = c2;
      C     c5();
      C();

      c1 = c4 = c2;

      return 0;
};
```



To conclude

- Complex class as a dynamic array of characters
- Intensive use
  - Operator overloading
  - Cpp subtleties
  - Normal form / Big 3
- As a base for containers
  - Object that contains others objects
  - vector = dynamic array of elements
  - Linked list
  - Hashmap
- Errors handling ?

Error free

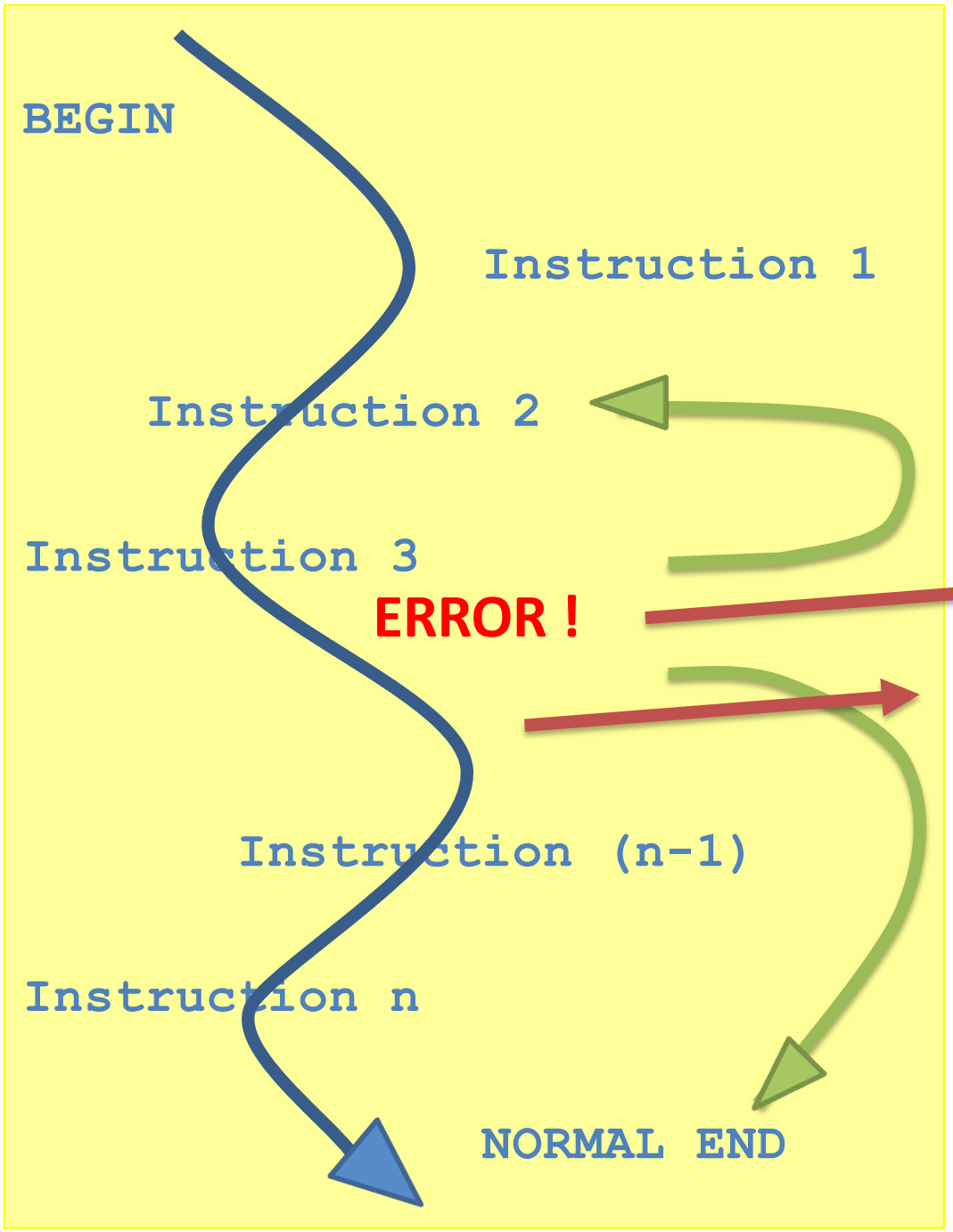




ERROR HANDLING: EXCEPTIONS



ISIMA



Detection ?

Go back ?

Exit ?

- Immediate ?
- Differed !
- "Normal"

Usual detection

Function with return

Easy to put

Advanced diagnosis

Monopolized return

switch

Not mandatory to check

Global variable

Can use the return freely

Values conflicts ?

Conflict name ?

A lot of values

```
int function(formal parameters) {
  if (ConditionError1)
    return CONSTANT_ERROR1;
  if (ConditionError2)
    return CONSTANT_ERROR2;
  …
  return CONSTANT_SUCCESS;
}
```

```
switch (function(real_parameters)) {
  case CONSTANT_ERROR1:
    …
    break;
  case CONSTANT_ERROR2:
    …
   break;
  case CONSTANT_SUCCESS:
   …
   break;
}
```

Example of classical mecanism

Program exit

Error message and EXIT

Clean end
with `std::atexit()`

Always ended

Hard to adapt the program

Dynamic allocation ?

```
if (condition_for_error) {
   std::cerr << "Message"
             << std::endl;
   std::exit(1);
}
```

Program end

- « Clean »
  - Closed files
  - Freed resources
  - Static destructors called

Natural end
ou `exit()`

```
std::atexit(f);
```

- Not expected : not handled exception
  - No freed resources
  - Default call of `abort()`
  - Directly callable `std::terminate()`
  - Updatable

`terminate()`
`unexpected()`

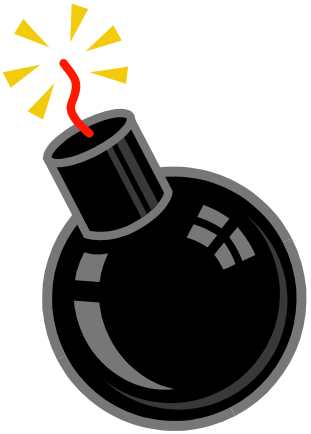
```
void f(void) {}
std::set_terminate(f);
```

```
class Gossip {
   std::string name;
 public:
   Gossip(std::string n):nom(n) {
      std::cout << ”constructor " << name << std::endl;
   }
   ~Gossip() {
      std::cout << ”destructor "  << name << std::endl;
   }
};

Gossip g("global");

int main(int, char **) {
   Gossip         t("local");
   static Gossip s("statlocal");

   // std::exit(1);
   // std::terminate();
   return 0;
}
```

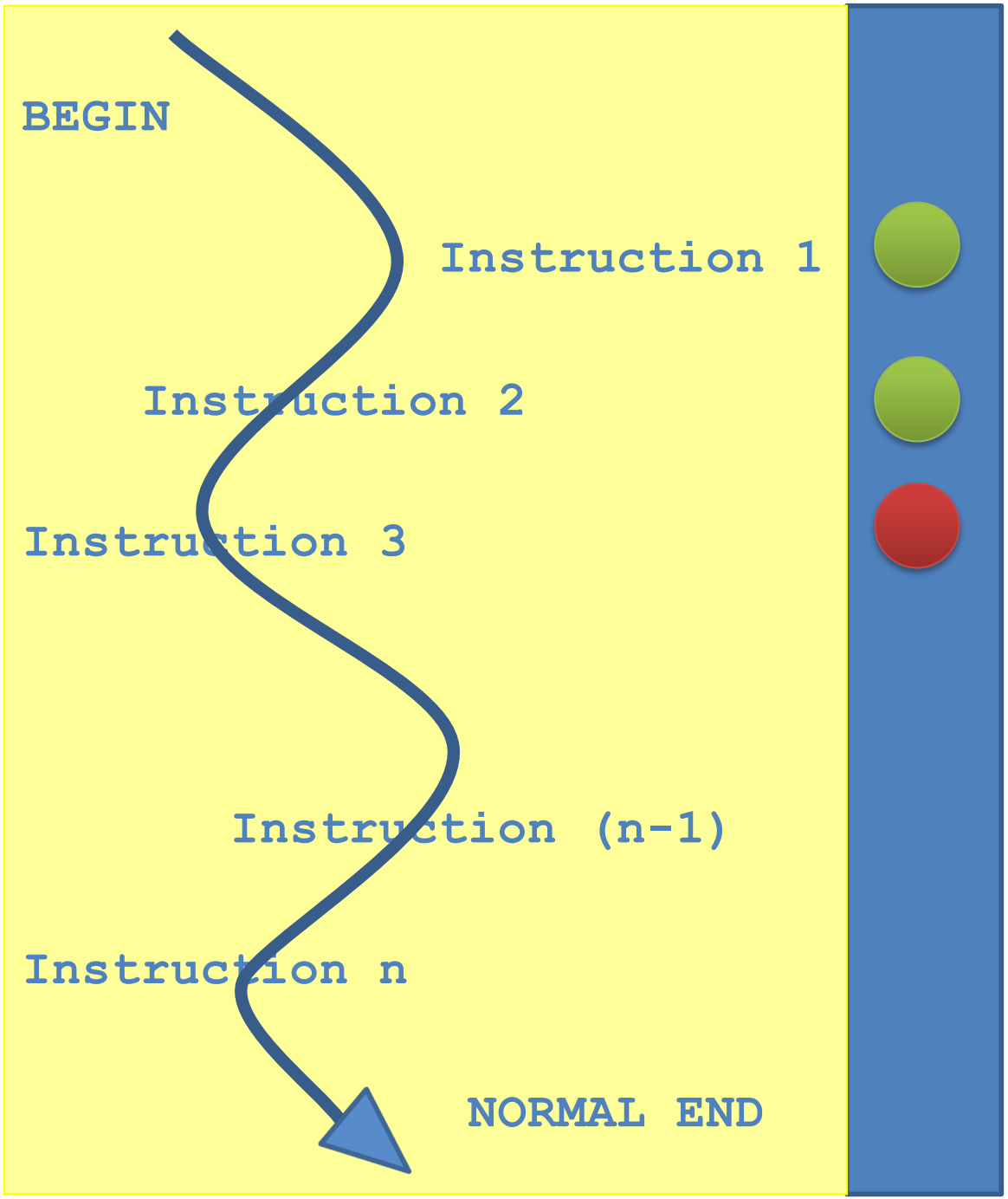
Static Objects destroyed by `exit()`

Exceptions

- Errors handling or exceptional behaviors
- Can be thrownded
  - Any kind of data, theoritically
  - But specialized classes usually
- Handling
  - Blocks under surveillance
  - Handlers with specialized code



Exception not handled =
Program EXIT



Exception creation
(**throw**)

Handling
(catch)
Or thrown again

Program exit ?

Example

```
void f() {
   int * arr = nullptr;

   arr = new int [ALOT];

   for(int i =0; i< ALOT; ++i)
      std::cout << arr[i] << " ";

   delete [] arr;

}
```

IMPOSSIBLE ALLOCATION I ?

Returns nullptr ?

std::bad_alloc

Catching an exception…

```
void f() {
   int * arr = nullptr;
   try {
      arr = new int [ALOT];

      for(int i =0; i< ALOT; ++i)
         std::cout << arr[i] << " ";

      delete [] arr;
   } catch (std::bad_alloc &e) {
      std::cerr << e.what();
   }

}
```

NEVER EXECUTED

EXIT

Catch Exception

Nested class

```
class OuterClass
{
  public:
     class InnerClass {};
};
```

nested class

- Inside the class
  - Usual use
- Outside the class

Cleaner code

No name conflict

```
OuterClass::InnerClass instance;
```

Can be used liked the inner types



Another example

```
bool redo = true;
do {
   cout << "Enter a valid index" << endl;
   cin >> i >> v;
   try {
        string[i] = v;
        redo = false;
   } catch(const String::BoudingException &e) {
        cerr << "ERROR: try again :";
   }
} while (redo);
```



Advice :
Const reference



We can do it WITHOUT exception

Standard Exception

- Special class from the standard library

```
#include <exception>
```

```
std::
```

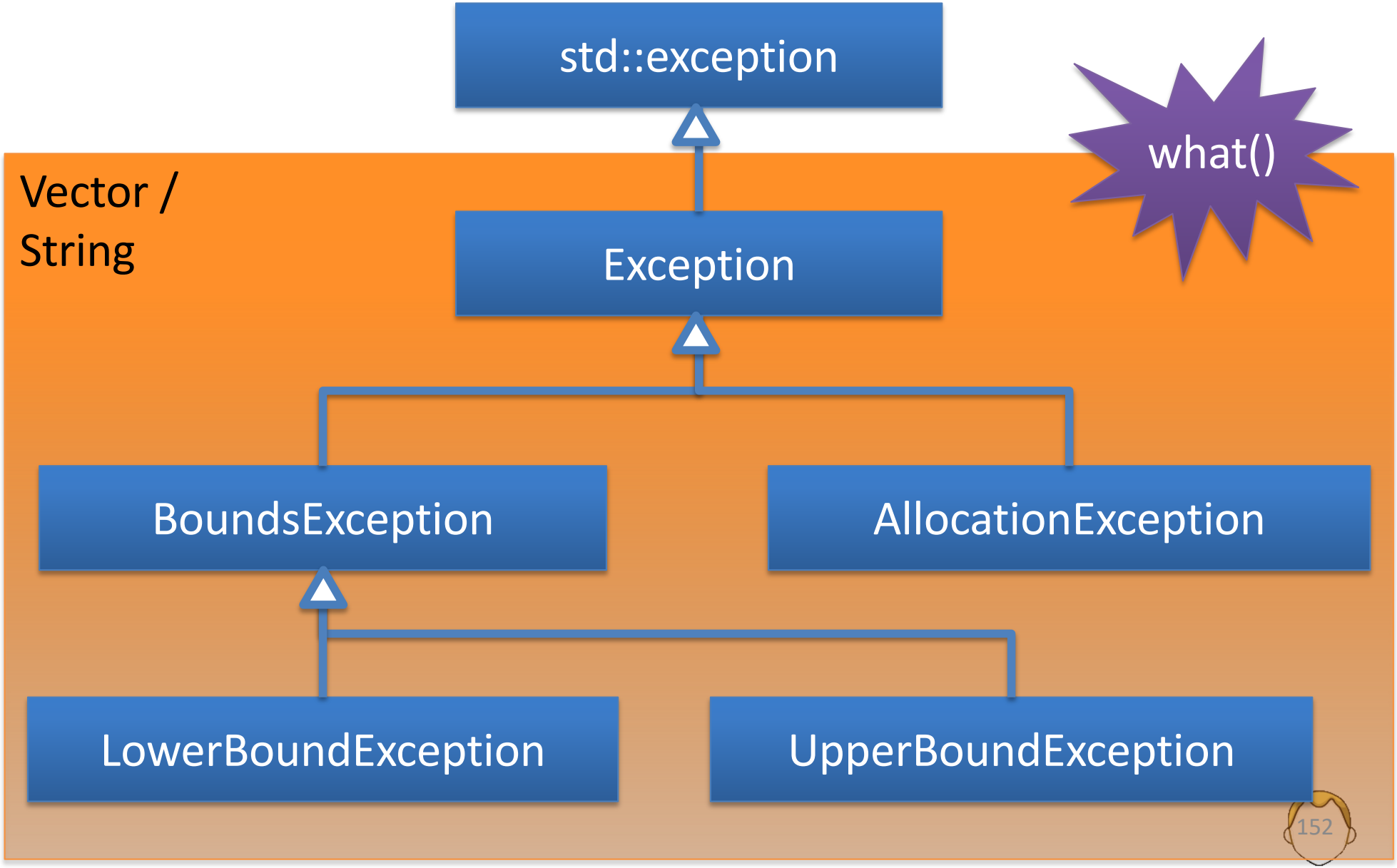
- Base class for all exception classes
  - To specialize
  - Order/Specialization to specify

noexcept **on all methods**

```
class exception {
 public:
  exception();
  exception(const exception&);
  exception& operator=(                n&);
  virtual ~exception();
  virtual const char * what() const;
};
```

To override

Exception Inheritance



Handle exception hierarchy

- Always handle the most specialized classes
- Always use a **const reference**
  - To avoid a copy
  - To ensure polymorphism
- Catching `std::exception`
- Catching everyting else `...`

Order matters …

Specialized



General

```
try {
 // Code that can throw exception
} catch (const Vector::AllocationException &e) {
  // Special handling
} catch (const Vector::LowerBoundException &e) {
  // Special handling
} catch (const Vector::Exception &e) {
  // More general handling
} catch (const Vector::UpperBoundException &e) {
 // Special handling again
}
catch (const exception& e) {
  // Handling for all kind of exceptions
} catch (...) {
  // Anything
}
```

Never executed

Exceptional String

```
class String
{
  public:
    class BoundsException {};

    char& operator[](int ind
    {
      if ((index < 0) || (index > size))
        // Code that causes trouble
        throw BoundsException();
      return [index];
    }
};
```

Nested class for the potential exception

```
: public std::exception {};
```

Instanciation

Throws the exception

Throwing again an exception

- Back to normal ? impossible
  - Multi level handler
  - End of the program ?



```
try {
   c = string[3];
}
catch (const String::BoundsException &e) {
   // handler
   if (…) throw;
}
```

Avoids truncating

Exception qualifier(s) ?

- Potential exceptions list
  - Flawed support before C++11
  - Deprecated since …. EXCEPT FOR

- No exception
  - Ensures no exception is thrown
  - Belongs to signature

```
~String() noexcept;
```



- Main feature
- Compilation checks



Some rules

- Constructor
  - Can raise an exception
  - Resources allocation
- Destructor
  - Should never throw an exception
  - Frees resources

RAII:

Resource
Acquisition
Is
Initialization

- Catch for « anything »
- New' s behavior

new()

- Normal behavior

```
try {
    t1 = new double[n];
}
catch (const std::bad_alloc &) {
}
```

Exception when an error occurs (like Java)

- Change of heart
  - returns 0 when an error occurs

#include <new>

```
t1 = new (std::nothrow) double[n];
```

```
class ProperAlloc {
  public:
     ProperAlloc(int n):a1(nullptr), a2(nullptr) {
         try {
              a1 = new double[n];
              a2 = new double[n];
         }
         catch ( const std::bad_alloc & ) {
              delete [] a1;
              throw;
         }
     }
     ~ProperAlloc() {
         delete [] a2;
         delete [] a1;
     }

 private:
     double * a1, *a2;

};
```

No problem if t1 is not allocated…

Throws the exception

Normal destruction

RAII

```
try {
   ProperAlloc pa;
} catch (const bad_alloc& e) {
    cerr << "instanciation not performed";
}
```

```
class C {
   int * att;
 public:
   C(const C&);
   C& operator=( … );
   void swap(C&);
}
```

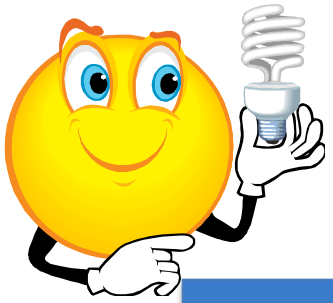
RAII



Copy & swap

```
C& C::operator=(const C & o){
    C temp(o);
    temp.swap(*this);
    return *this;
}

void C::swap(C & o) noexcept {
   std::swap(this->att, o.att);
}
```

```
C& C::operator=(C o){
    o.swap(*this);
    return *this;
}
```



Exceptional Conclusion

Exception = errors handling mecanism

Rigorous

Cannot be ignored

CPU cost

Executable size



False security feeling

Any exception can lead to the end of the program if not taken into account



template <typename T>

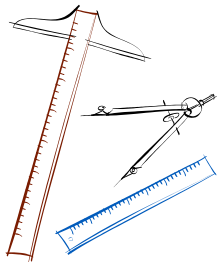
GENERICITY - TEMPLATES



ISIMA



Genericity



- Does not belong to OO paradigm
- Who ?
  - Parameterized functions
  - Parameterized classes
- Parameters ?
  - Types
  - Constants

Template / generics

- Write once = a model
- The model needs to be instanciated with type(s)

Max $_{(1)}$: dedicated functions

```
const int& max(const int& a,const int & b) {
  return ((a > b) ? a : b);
}
```

Overloading

Type checking

```
const double& max(const double&a,
                  const double& b) {
  return ((a > b) ? a : b);
}
```

Duplicated code

- Errors
- Executable size

Max (2) : macro

```
#define MAX(a, b) \
   (((a) > (b)) ? (a) : (b))
```

- No type ckecking
- Weird syntax
- Executable size
- Side effects

- Code Reuse
- Performances



Max (3) : parameterized function

typename or *class*

```
template <typename T>
const T& max(const T&a, const T&b) {
  return ((a > b) ? a : b);
}
```

No automatic type cast

Code reuse

Can be optimized with inlining

template instanciation

```
int     i, j;
double d, f;
Classe a, b, c;
```

```
cout << max(i,j);
```

`int max(int, int)`

```
cout << max(d,f);
```

`double max(double, double)`

```
cout << max(i,d);
```

Not automatic cast

```
cout << max<double>(i,d);
```

```
c = max(a,b);
```

Correct if comparison and assigment operators exist

`std::max` exists !

```
class BananaBasket {
   Banana * content;
} basketOfBananas;

class AppleBasket {
   Apple * content;
} basketOfApples;
```

Generic class ?

```
class Basket {
   Fruit * content;
} anything;
```

```
template <typename F> class GenBasket {
   F * content;
};

GenBasket<Banana>  basketOfBananas;
GenBasket<Apple>   basketOfApples;
```

When and write to write a template class ?

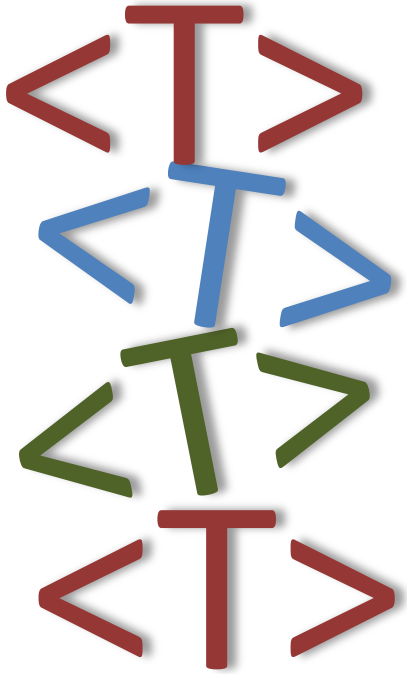
- With a normal class (not template)
  - 1 type given
  - 2 files : declaration (hpp) et definition (cpp)
  - The class is **tested**
- Merge the two files into a single **header file**
  - No implementation file
  - No multiple definition error
- Replace le given type to a template one
- Instanciation of this *template*
  - Only an instanciated template is compiled



Generic stack ?

- Classical data structure
- To simplify : a fixed sized array

Stack
- arr : array of integers - size : integer - capacity : integer
+ Stack(capacity : integer) + ~Stack() + top() : integer + isEmpty() : boolean + pop() + push(integer)

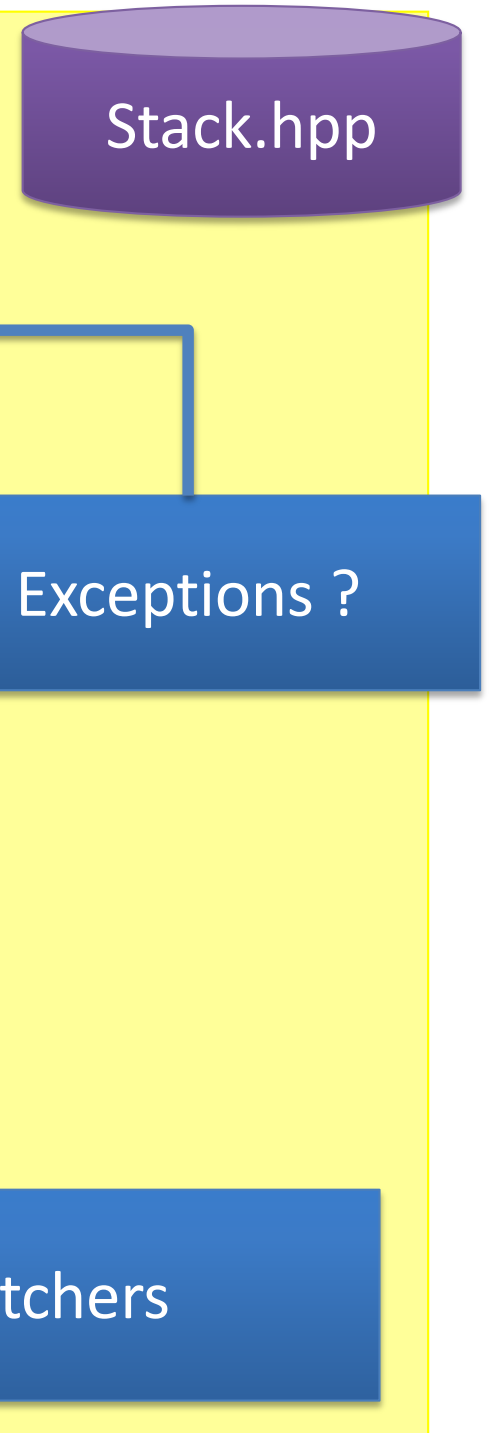


Stack use case

```
#include <iostream>
#include "Stack.hpp"

using namespace std;

int main(int, char **) {
  Stack p1;

  p1.push(10);
  p1.push(20);

  while (!p1.isEmpty()) {
    cout << p1.top() << endl;
    p1.pop();
  }
  return 0;
}
```

Use case
~~TEST~~

```
class Stack
{
  public:
    Stack(int = 256);
    const int& top() const;
    bool isEmpty() const;
    void pop();
    void push(const int&);
   ~Stack();

  private:
    int    size;
    int    capacity;
    int * arr;
};
```



Stack.cpp

```
Stack::Stack(int capa) :
      capacity(capa),size(0) {
  arr = new int[capacity];
}
const int& Stack::top() const {
  return arr[size-1];
}
bool Stack::isEmpty() const {
  return (size==0);
}
void Stack::pop() {
      --size;
}
void Stack::push(const int& el) {
  arr[size]=el;
  ++size;
}
```

```
~Stack() {
  delete [] arr;
}
```

```
template<typename T >
class GenStack
{
  public:
    GenStack(int = 256);
    const T & top() const;
    bool isEmpty() const;
    void pop()
    void push(const T &)
   ~GenStack()

  private:
    int   size;
    int   capacity;
    T * arr;
};
```

Template

Not here !

GenStack.hpp

Replace the type where it appears

Not here !

```
template<typename T >
GenStack<T>::GenStack(int capa) :
      capacity(capa),size(0) {
  arr = new T [capacity];
}

template<typename T >
const T & GenStack<T> ::top() co
  return arr[-1];
}

template<typename T >
void GenStack<T> ::push(const T & el) {
  tab[size]=el;
  ++size;
}
```

GenStack.hpp
(suite)

~GenStack()
pop()
isEmpty()
TO DO

This code is a model =
It is not compiled

```
#include <iostream>
#include "GenStack.hpp"

int main(int, char **) {
   typedef GenStack<int> IntegerStack;

   IntegerStack s1;

   s1.push(10);
   s1.push(20);

   while (!s1.isEmpty()) {
      cout << s1.top() << endl;
      s1.pop();
   }
   return 0;
}
```

INSTANCIATION
⇒ This instanciated model is compiled

```
GenStack<Point>
GenStack<Shape *>
```

Constant as a parameter

```
#ifndef __STATICGENSTACK_HPP__
#define __ STATICGENSTACK _HPP__

template <class T, const int SIZE = 256 >
class StaticGenStack
{
  public:
    StaticGenStack() : size(0) {}
   ~StaticGenStack() {}
  private:
    T arr[SIZE];
};
#endif
```

What is remaining is the same !

Instanciation & control

```
StaticGenStack<double>     sgs1;
StaticGenStack<int,   512> sgs2;
StaticGenStack<int,   512> sgs3;
StaticGenStack<int, 1024> sgs4;
```

```
sgs2 = sgs3;
```

```
sgs1 = sgs2;
```

```
sgs2 = sgs4;
```

DYNAMIC STACK



STATIC STACK

- Possible reallocation
- Executable Size
- `new()` is slow

- No reallocation
- Executable Size
- Runtime speed

One or several parameters ?

```
template <typename SUPER_LONG_NAME>
class C1 {
};
```

```
template <typename T1, typename T2>
class C2 {
};
```

```
template <typename T, typename C = C1<T> >
class C3{
};
```

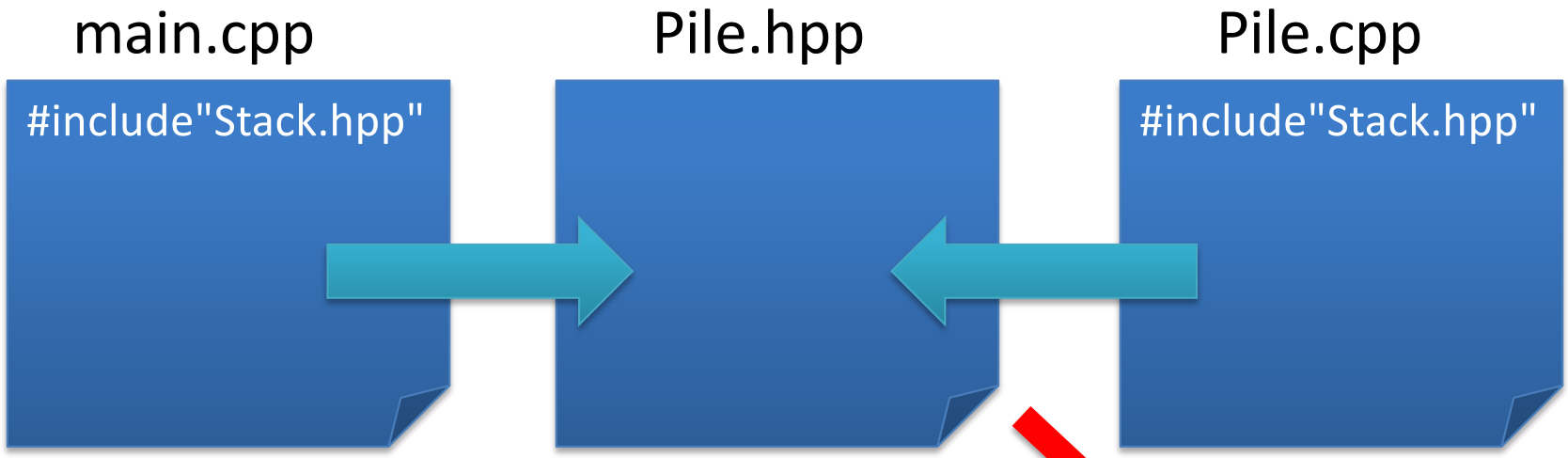
```
Desk< Book, GenStack<Book>>
```

Comparison operators : very handy template functions

```
template <typename T>
  bool operator<=(const T&a , const T&b) {
     return ((a < b) || (a == b));
}
template <typename T>
  bool operator >=(const T&a, const T&b) {
     return !(a < b);
}
```

With == and < or > , all are written !

Template & compilation




```
> g++ -c main.cpp
> g++ -c Stack.cpp
> g++ -o main main.o Stack.o
```

Non template version

```
> g++ -c main.cpp
> g++ -c GenStack.cpp
> g++ -o main main.o GenStack.o
```

Template version

A *template* is a MODEL = "no compilation"

A header file but how ?

- 1 file

Mess ?

Handy

```
template <typename T>
class C {
  void m();
};
template <typename T>
void C<T>::m() {
}
```

"Remote" methods

```
template <typename T>
class C {
  void m1() { ... }
  void m2() { ... }
};
```

Nested methods

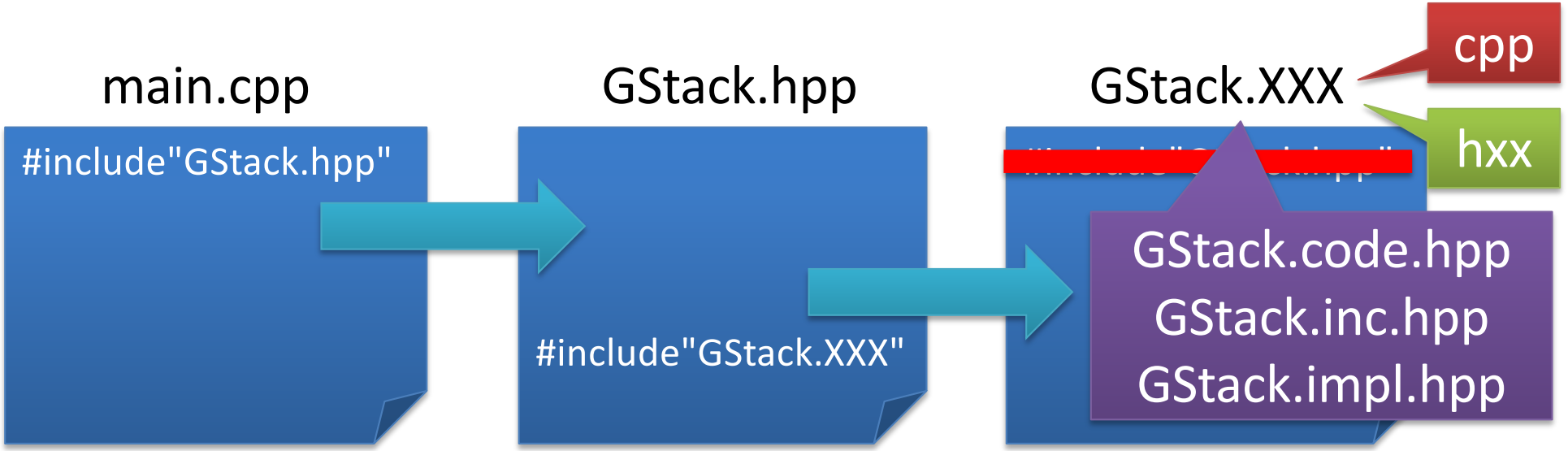
- 2 files

Clean

Usual

Complicated

Template with 2 files



GStack.cpp

Extension known by the editor

Automatically compiled by make ?

GStack.hxx

Extension not always recognized
Editor to set up

Not automatically compiled

Code in the header BUT no multiple definition error


STANDARD LIBRARY

Introduction

- Recurring needs
  - Data structures : static or dynamic array, stack, file, set…
  - algorithms : search , order, insert, extract…
- Without reinventing the wheel
  - Lost time (coding, debuging, optimization)
  - Using what is existing (libraries)
- All modern languages have their own library
  - java, C#, perl, python
  - C++



Why using the standard library ?

- Reliable
  - collection of used and reused classes
- Portable
  - total => standard
- Efficient
  - Intensive use of genericity
  - Optimized data structures (even if …)
- Comprehensive
  - Usage and Writing conventions

Content

- C standard
- Base classes
  - streams, string, stringstream, exception
  - Traits : char or numerical
  - unique_ptr, shared_ptr, weak_ptr
- Containers and iterators
- Algorithms et functors

} "STL"

Example

- Declaration and initialization
- Reading of integers from the standard input
- Sort
- Display of the sorted list

- Container
  - To store objects
- Iterator
  - Go thru a container
  - Beginning, end, current element, go to the next one



```
int cmp (const void *a, const void *b) {
  int aa = *(int *)a;
  int bb = *(int *)b;
  return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
}
```



C/C++ Example

```
int main (int, char **) {
  const int size = 1000;
  int array [size];
  int n = 0;
```

Declarations

```
  while (std::cin >> array[n++]);
  --n;
```

Input

```
  qsort (array, n, sizeof(int), cmp);
```

Sort

```
  for (int i = 0; i < n; i++)
    std::cout << array[i] << "\n";
```

Display

```
 return 0;
}
// http://www.cs.brown.edu/people/jak/proglang/cpp/stltut/tut.html
```

```
#include : iostream
           cstdlib
```

```
#include <iostream>
#include <algorithm>
#include <vector>

int main (int, char **){
  std::vector<int> v;
  int input;
```

Declarations

```
  while (std::cin >> input)
    v.push_back (input);
```

Input

```
  std::sort(v.begin(), v.end());
```

Sort

```
  int n = v.size();
  for (int i = 0; i < n; i++)
    std::cout << v[i] << std::endl;

  return 0
}
```

Display



```
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>
int main (int, char **) {
  std::vector<int> v;
  int input;

  while (std::cin >> input)
    v.push_back (input);

  std::sort(v.begin(), v.end());

  std::copy (v.begin(), v.end(),
    std::ostream_iterator<int> (std::cout, "\n"));

  return 0;
}
```



Declaration

Input

Sort

Display

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>

int main (int, char **) {
  vector<int> v;

  istream_iterator<int> start (cin);
  istream_iterator<int> end;
  back_insert_iterator<vector<int> > dest (v);

  copy (start, end, dest);

  sort(v.begin(), v.end());

  copy (v.begin(), v.end(),
     ostream_iterator<int>(cout, "\n"));

  return 0;
}
```



Declaration

Input

Sort

Display

3 types of containers

- Sequence
  - Array, vector, list, queue
- Adaptators ot sequence containes
  - stack, file and priority file
- Associatives containers
  - Set with unicity or not
  - Map with unique or multi key
  - With or without order on keys

`std::`

Intensive use of genericity

Common operations

- Canonical Shape(extended to 2011)
- Automatic capacity adaptation
  - When inserting beyond capacity
  - Capacity * 2
  - Quick Adaptation to "final" size

Except `array`

```
int  C::size() const      // # of elmts
int  C::max_size() const  // max #
bool C::empty() const     // empty ?
void C::swap(C &)         // content swap
void C::clear()           // emptying
```

Common to sequence containers

- Insertion

```
S::iterator S::insert (S::iterator, T &)
S::iterator S::insert (S::iterator, int, T &)
S::iterator S::insert (S::iterator, S::const_iterator,
                                    S::const_iterator)
```

- Removing

```
S::iterator S::erase(S::iterator)
S::iterator S::erase(S::const_iterator,
                     S::const_iterator)
```

Iterators after removing elts ?

- Access to extremities

```
void        S::push_back (T &)
void        S::pop_back ()
      T &   S::front ()
const T &   S::front () const
      T &   S::back  ()
const T &   S::back  () const
```

[fixed] Array

2011

```
#include <array>
```

```
std::array<X , int> a;
```

```
size_type A::size()     const
void      A::fill(const X&)

      X & A::operator[] (int)
const X & A::operator[] (int) const
```

O(1) access

Cannot change size

Homogeneous interface

Vector – dynamic array

```
#include <vector>
```

```
std::vector<X> v;
```

```
size_type V::capacity    ()     const
void      V::reserve     (int)

      X & V::operator[] (int)
const X & V::operator[] (int) const
```

O(1) access

Adding in amortized O(1)

Inserting / Deleting in O(n)

```
int main (int, char **) {
  typedef std::vector<int> ivector;

  ivector v1;
  for (int i = 0; i < 4; ++i)
    v1.push_back(i);

  ivector v2(4); // initial size 4
  for (int i = 0; i < 4; ++i)
    v2[i] = i;

  std::cout << (v1 == v2) ? "Ok" : "Not Ok"
            << std::endl;

  return 0;
}
```



List(s)

```
#include <list>
#include<forward_list>
```



2011

```
std::list<T>         l1;
std::forward_list<T> l2;
```

```
void L::push_front (const T &)
void L::pop_front  ()

void L::remove  (const T &)
void L::sort    ()
void L::sort    (Comparator)
void L::merge   (list<T> &)
splice, remove_if, unique
```

Inserting / deleting in O(1)

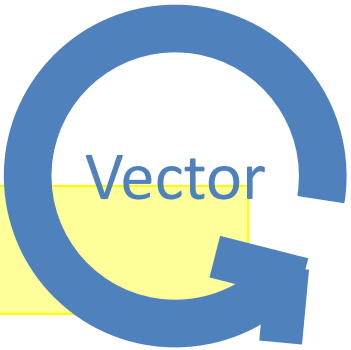
Access in O(n)

```
int main(int, char **) {
  typedef std::list<int> ilist;

  ilist l;
  for (int i = 0; i < 4; ++i)
    l.push_back((10 + 3*i) % 5);
  l.sort();    // 0 1 3 4
  l.reverse(); // 4 3 1 0

  // display between beginning and end
  std::cout << l.front() << ' ' << l.back()
            << std::endl;

  return 0;
}
```



Double ended queue

```
#include <deque>
```




```
std::deque<T> d;
```

```
int        D::capacity    () const
void       D::reserve     (int)
void       D::push_front  (const T&)
void       D::pop_front   ()
       T&  D::operator[]  (int)
const  T&  D::operator[]  (int) const
```

Inserting / Deleting in O(1)

All advantages of vector



```
int main(int, char **) {
  typedef std::deque<int> ideque;

  ideque v1;
  for (int i = 10; i >=0; --i)
    v1.push_front(i);

  ideque v2(4); // initial size 4
  for (int i = 0; i < 4; ++i)
    v2[i] = i;

  std::cout << (v1 == v2) ? "Ok" : "Not Ok"
            << std::endl;
  return 0;
}
```

Problems ➔ iterators

- Go throuh a container OFTEN

Pointing the current element in the container

One at a time

Encapsulation of the pointer from outside

- Several traversals (forth, back) SOMETIMES

Defining the strategy in the container

One at a time

Encapsulation of the pointer from outside



Iterators

- Defines the access to an elmt
  - A pointer most of a time
  - Container implementation
  - has to be known
  - Nested class
- Defines the strategy traversal

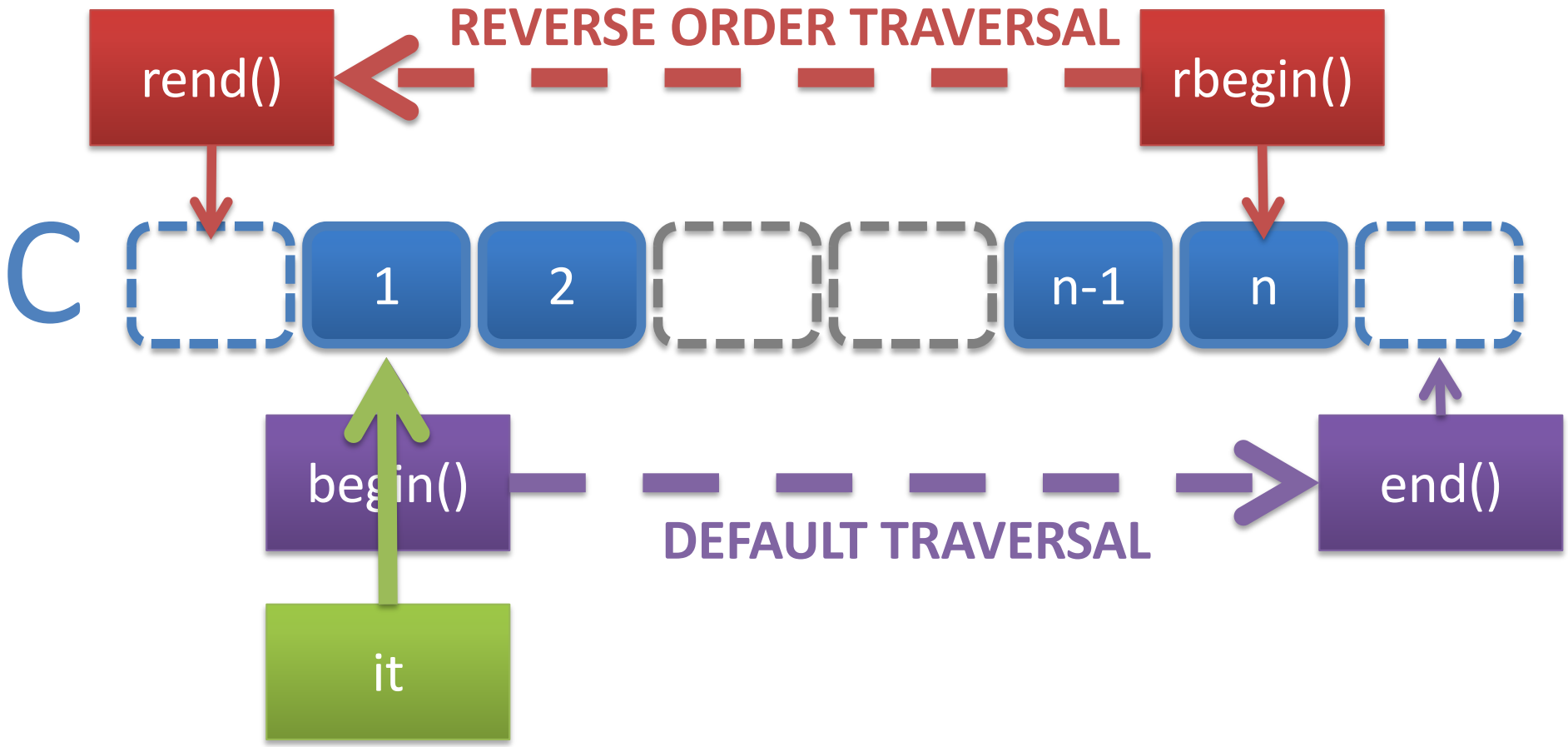
```
class container {
 public:
  class iterator {
  };
};
```

```
container::iterator
container::const_iterator
container::reverse_iterator
container::const_reverse_iterator
```

Back / Forth

Read/Write
Read Only

```
std::vector<int>::iterator
```



FNC	Incrementing ++ pré et post	Dereferencing	Comparison == !=
`it=c.begin()`	`++it;`	`*it`	`it!=c.end()`

Iterators use

- Container traversal

```
container c;
container::iterator it;
for (it = c.begin(); it != c.end(); ++it)
  // do something with (*it);
```

- A particular element (search result)

```
container::iterator it =
    find(c.begin(), c.end(), elt);
cout << (it!=c.end())?"found":"unknown";
```

*it == elt

Slides 176 and following
should be more understandable now

Writing to a stream (1)

```
typedef std::vector<int> C
// using C = std::vector<int>;
C source = { 1, 2, 3, 4, 5 };

std::ostream& o = std::cout;
const char  * s = " ";
C::iterator it  = source.begin();

while(it != source.end() ) {
    std::cout << *it << s;
    ++ it;
}
```

#include<iterator>
#include<algorithm>

Writing to a stream(2)

Container of objects T

iterators

```
std::copy(source.begin(), source.end(),
          destination );
```

Stream

Separator

```
std::ostream_iterator <T>(std::cout, " ");
```

Operator << mandatory

Element to write

"Adapted" container

- Sequence / base container
- Different interface



Stack
Access to the top

Array
All places are accessible



Adapted container:,stack

```
#include <stack>
```

C<T> : deque by default
vector,list are possible

```
std::stack<T, C<T> > s1;
std::stack<T>        s2;
```

LIFO

```
bool      S::empty() const;
int       S::size () const;
void      S::push (const T& );
void      S::pop  ();
const T&  S::top  () const;
T&        S::top  ();
```

Last arrived

```
int main(int, char **) {
  stack<int> is;
  stack<double, vector<double> > ds;

  for (int i = 0; i < 100; ++i)
    is.push(i);

  while (!is.empty()) {
     std::cout << is.top() << std::endl;
     ds.push((double)is.top()*is.top());
     is.pop();
  }

  return 0;
}
```

Adapted container: queue

```
#include <queue>
```

C<T> : deque by default
vector,list are possible

```
std::queue<T, C<T> > s1;
std::queue<T>        s2;
```

FIFO

```
bool      S::empty() const;
int       S::size () const;
void      S::push (const T& );
void      S::pop  ();
const T&  S::front() const;
T&        S::front();
const T&  S::back () const;
T&        S::back ();
```

First In

Last in

```
int main(int, char **) {
  queue<int> iq;
  queue<double, list<double> > dq;

  for (int i = 0; i < 100; ++i)
    iq.push(i);

  while (!iq.empty()) {
     std::cout << iq.front() << std::endl;
     dq.push((double)iq.front()*iq.front());
     iq.pop();
  }

  return 0;
}
```



Adapted container : priority queue

```
#include <queue>
```

C<T> : vector by default
deque,list are possible

```
std::priority_queue<T, C<T>, L<T> > s1;
std::priority_queue<T>                s2;
```

Operator < by default, or
comparator
(functor)

```
bool      S::empty() const;
int       S::size () const;
void      S::pop  ();
void      S::push (const T& );
const T&  S::top() const;
T&        S::top();
```

```
class S { // Student
   string last, first;
   double grade;
   // …
};
```



```
bool operator<
   (const S&,
    const S&)
```

```
typedef vector<S>  vs;
// using   vs = vector<S>;
vs s;
// s.push_back(S(…));
priority_queue<S> pq;
for(vs::iterator it = s.begin();
     it!=s.end(); ++it)
 pq.push(*it);

while(!ps.empty()) {
   cout << pq.top() << " ";
   pq.pop();
}
```

Changing order ?

Display with the choosen order

Containers of pointers

```
typedef std::vector<S *>  vs;
// using vs = std::vector<S *>
```

```
vs s;

for(int i=0; i< 100; ++i)
    s.push_back(new S());

s.clear();
```

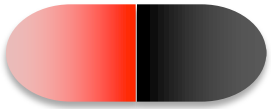
MEMORY LEAK !

```
for(vs::iterator it = s.begin();
     it!=s.end(); ++it)
    delete(*it);
s.clear();
```



Sorted associative containers

- Elements sorted on a key
  - Natural order
  - Comparator
  - Trees



Operations in O(log n)

- Only one key
  - Unique
  - Multiple

```
std::set<key>
std::multiset<key>
```

- Key value pair
  - Unique key
  - Multiple key

```
std::map<key, value>
std::multimap<key, value>
```

Map



Pair

pair< , >




```
typedef map<string, string> mss;
mss m;

m.insert(pair<string,string>("er","42"));
m.insert(make_pair("loic"  , "405042"));
m["pierre"]=  "405033";
```

Operator [] CREATES the pair

```
mss::iterator it = m.find("loic");
if (it==m.end()) cout << "and me ?";
```

```
const string& first(const pair<string,string>&
p) { return p.first; }


int main(int, char**) {
 map<string, string> liste;

 map<string, string>::const_iterator it
      = liste.begin();
 while(it!=liste.end()) {
    cout << it->first << " "
         << it->second << endl;
    ++it;
 }

 transform(liste.begin(), liste.end(),
   ostream_iterator<string>(cout, " "), first);

 return 0;
}
```

Acces to the first element



Traversal

Copy does NOT work

Unorderred associative containers



- Unsorted elements
  - Use of buckets
  - Hashtables
  - Keys/elements not mutable

Operations in O(1)

```
std::unordered_set<cle>
std::unordered_multiset<cle>
```

```
std::unordered_map<cle, valeur>
std::unordered_multimap<cle, valeur>
```

Problems ➔ functors

- Aplying an algorithm to a container ?

To add a method to the container

API Container can be polluted
Always possible ?

Algorithm in a dedicated class

- Different versions of the algorithm ?

Use of overloading / Another method

Good version ?

Inheritance



Fonctors

- Class that implements an algorithm on the container
  - Overloading of the operator ()

```
xxxxx F::operator() (parameters)
```

```
xxxxx F::operator() ( ∅ )
```

  - With or without state
- Specialized class for any variant
- Acces to the container elements by iterators

A function in an object

Comparator

```
class Comparator {
  public:
    Comparator () {}
    bool operator() (const A& a1,
                     const A& a2) const {
      return (a1.val() < a2.val());
    }
};
```

Non internal state

2 objects to compare

std::less<A>

cmp.operator()(a1,a2)

```
Comparator cmp;
A a1, a2;
std::cout << cmp(a1,a2) << std::endl;
```

Like a function !

Even numbers generator

```
class EvenGen {
  protected:
    int v;
  public:
    EvenGen () { v = -2; }
    int operator()() { v+=2; return v; }
};
```

Last even number

No parameter

```
EvenGen gen;
vector<int> v(10);
generate(v.begin(), v.end(), gen);
copy(v.begin(), v.end(),
    ostream_iterator<int>(cout, " "));
```

```
cout << gen();
```

Some algorithms

```
generate()
generate_n()
copy()
```

Initialization / Copy

```
transform()
sort()
random_shuffle()
```

Change

```
find()
min_element()
max_element()
```

Search

Trait classes (1)

- T*emplate classes*
  - Properties of a TYPE
  - Types, constants, static methods
  - "instead of parameters" (Myers)
- Specialization of *template* + default parameters
- Different types
  - Characters
  - Numericals
  - Iterators

Trait classes(2)

- How to find the max in an array
  - tab[0]
  - INT_MIN, FLT_MIN, DBL_MIN dans <climits>
  - numeric_limits

```
template<typename T>
T findMax(const T* data, int numItems) {
  largest = std::numeric_limits<T>::min();
  for (int i=0; i < numItems; ++i)
    if (data[i] > largest)
      largest = data[i]; `
  return largest;
}
```

Source : http://www.cs.rpi.edu/~musser/design/blitz/traits.html

```
template<typename T>
struct numeric_limits {
  typedef T type;
};

template<>
struct numeric_limits<int> {
  typedef int type;
  static int min() { return INT_MIN; }
};
template<>
struct numeric_limits<double> {
  typedef double type;
  static double min() { return DBL_MIN; }
};
```

Template specialization

Smart pointers



- unique_ptr
  - Only one owner
  - Ownership can be given (*move*) without copy
  - Freed when out of scope
- shared_ptr
  - Several pointers on one object
  - Freed when out of scope
  - References counter
- weak_ptr
  - Temporary own
  - To be cast as shared_ptr to use it
  - Breaking cycles

`auto_ptr`

```
XXX_pointer
make_XXX<T>(objet)
```

unique_ptr



- Only one owner for a objet/pointer

```
X* f() {
   X* p = new X;
   // throw std::exception()
   return p;
}
```

Better : return unique_ptr<X>

```
X* f() {
   unique_ptr<X> p (new X);
   // throw std::exception()
   return p.release();
}
```

http://www.stroustrup.com/C++11FAQ.html#std-unique_ptr

Bitset



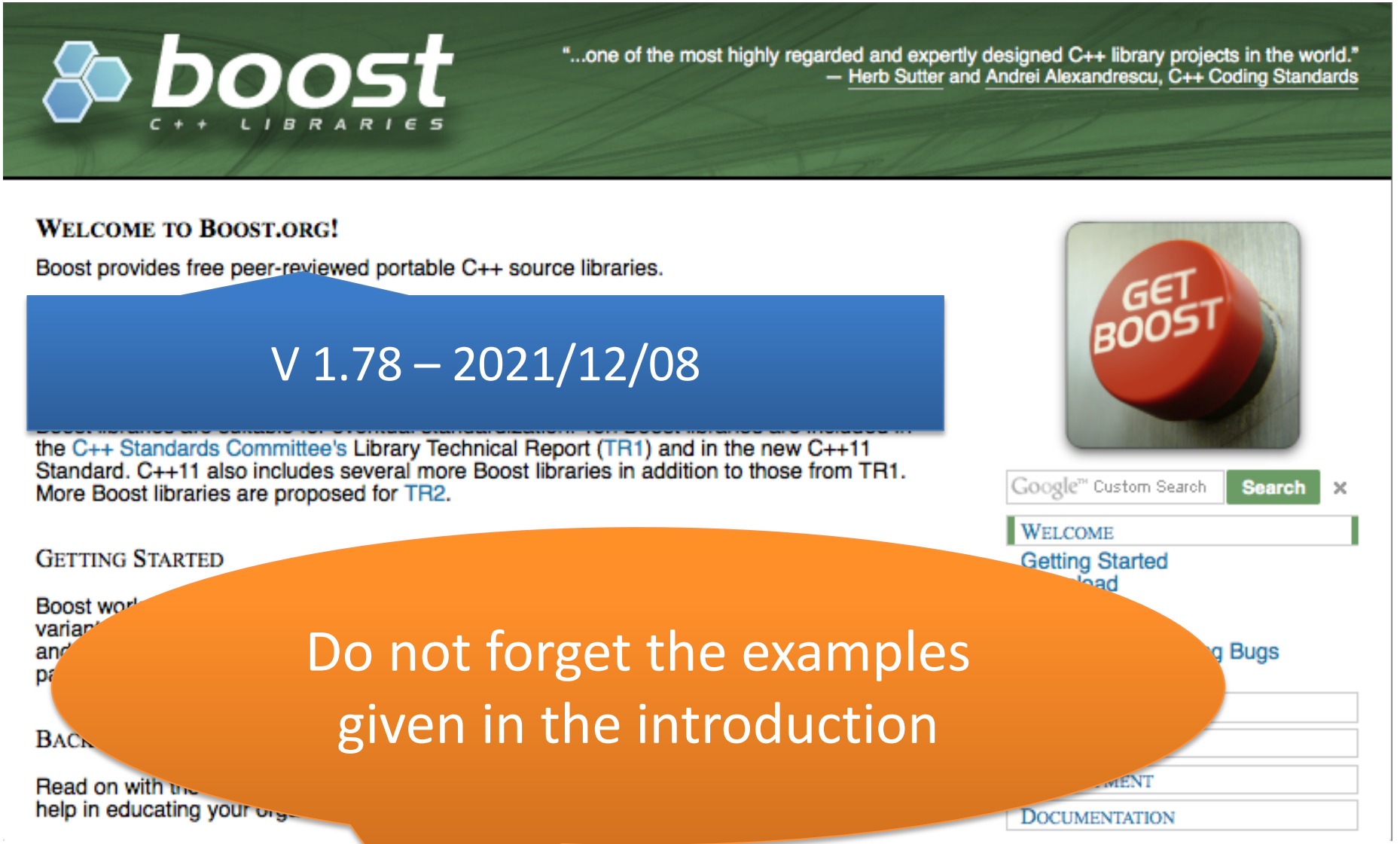
- Set of bits : 0 ou 1, true ou false
  - Optimized
  - Cast to from integer/string
  - Fixed size at compile time // vector<bool>

```
std::bitset<16> b1;
std::bitset<16> b2(33);
std::bitset<16> b3(std::string("010010"));
```

```
b1.set();
b2.reset();
b3.flip();
```

```
b1[3]
b2.set(2, 1);
b3.flip(4);
```

Pour aller plus loin…





WHAT WE DID NOT DARE TO TALK ABOUT … YET.

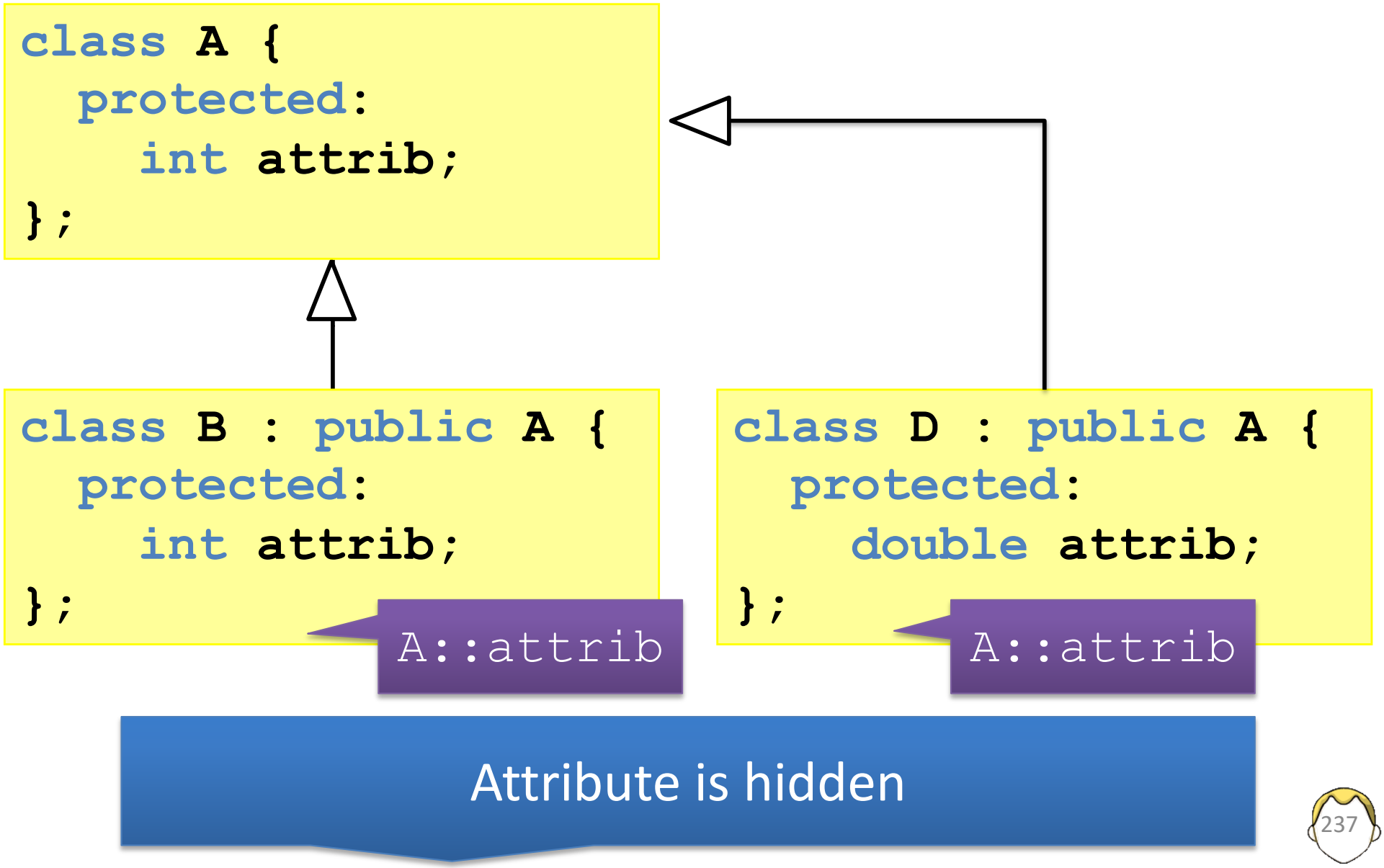


You want mre ?

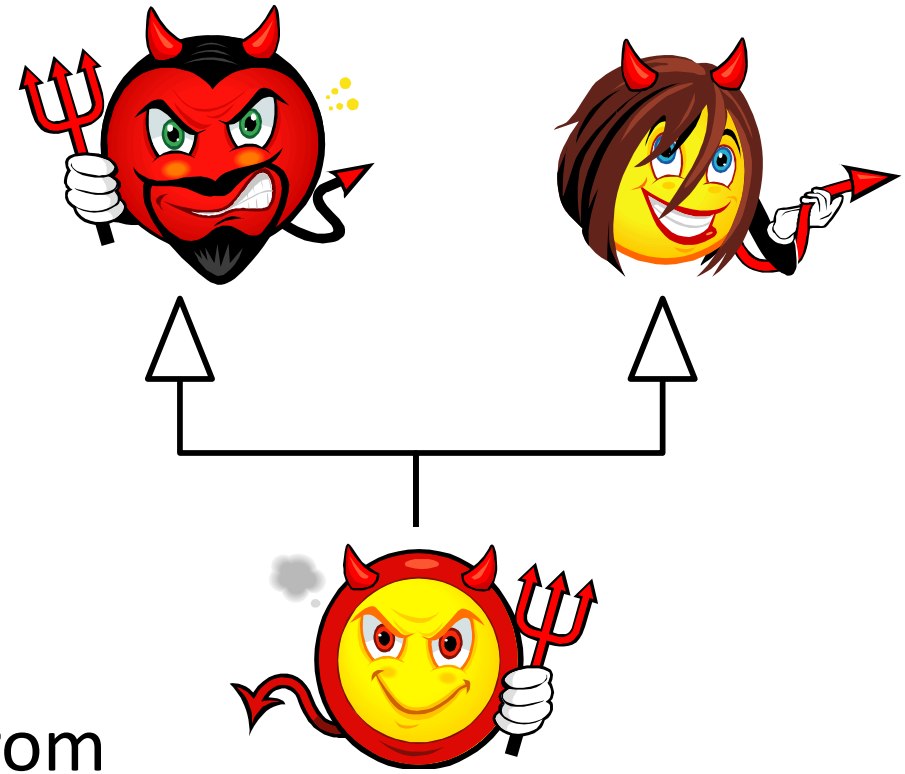


- Inheritance
  - Multiple, duplicate , virtual
  - Private and protected
  - More on virtuality
- Miscellaneous
  - *Template syntax*
  - Opérators ++
  - inline methods
  - Method pointers
  - Namespace

Inheritance : reminder



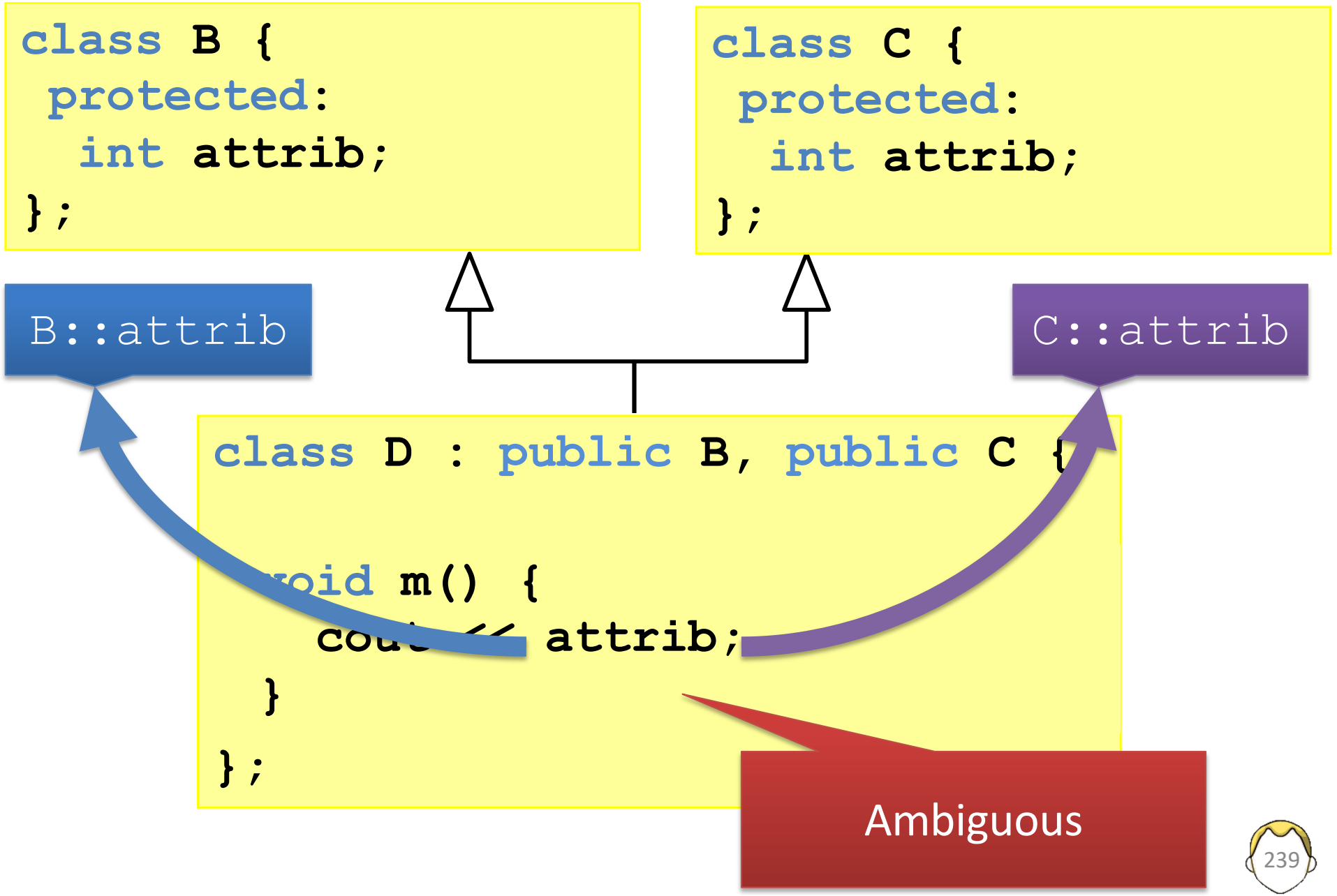
Multiple inheritance



- Natural
- All members and methods from all the base classes
  - Same name ? Ambiguous …

```
class F : public M1, public M2 {

};
```

Name ambiguity



Using

```
class B {
 protected:
  int a;
 public:
   B():a(1) {}
   void m() { cout << "B" << a;}
};
```

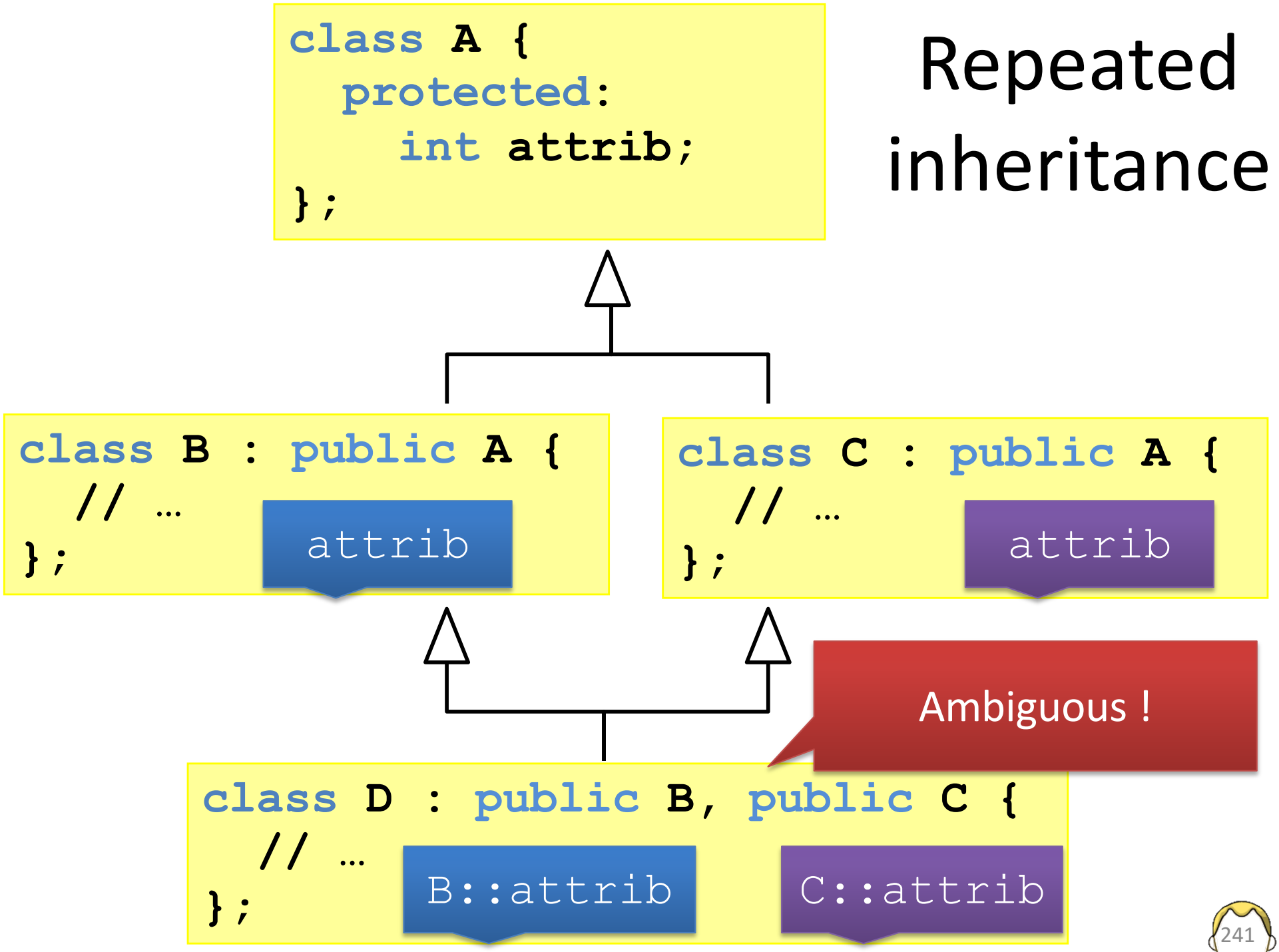
```
class C {
 protected:
   int a;
 public:
   C():a(2){}
   void m() { cout << "C" << a;}
};
```

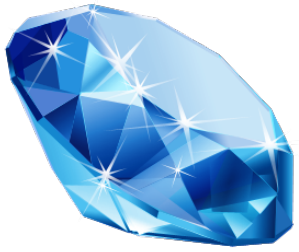
```
class D : public B, public C {
 protected:
  using B::a;
 public:
  using C::m;
  void n() { cout << a; }
};
```

```
int main(int, char **) {
   D d;
   d.m();
   d.n();
   return 0;
}
```

```
> C21
>
```

Repeated inheritance





Virtual inheritance (1)

```
class A {
  protected:
     int attrib;
};
```

No duplicate

```
class B :
 virtual public A {
   // …
};
```

attrib

```
class C :
  virtual public A {
   // …
};
```

attrib

```
class D : public B, public C {
   // …
};
```

attrib

Virtual inheritance(2)

A is not duplicated

`virtual` on B and C
And not only on D

```
class D : public B, public C {
   D(…):A(…),B(…),C(…) {}
};
```

Explicit creation of A

```
class D : virtual public A,
          public B, public C {
   D(…):A(…),B(…),C(…) {}
};
```

You can use this too !!

Private / protected inheritance (1)

- Will = who knows the relationship ?
- public
  - All the classes know the base class
- protected
  - Only descendants know the base class
- privé
  - The only class
  - Friends ?

How to cast ?
Base <-> Derived

Private / protected inheritance(2)

	Inheritance		
Member	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	✖	✖	✖
Base	Derived		

`private` inheritance ?

"is implemented

Access to the members
implementatio

vential use of
inheritance

Doable with aggregation
But with 2 objects

I am still looking...

… I have a clue

```
template<typename T> class vector {};

template<> class vector<void *> {
};

template<typename T>
class vector<T *> :
  private vector<void *> {
};
```



Example by B.S.

Classical Example …

```
class Engine {
  public:
    void start() {}
};

class Car : private Engine {
  public:
    using Engine::start;
};
```

Very strong link
Except hybrid car ☺

Classical inheritance

```
Car c;
c.start();
```

… but not satisfying

```
class Engine {
  public:
    void start() {}
};

class Car {
  public:
    void start()
        { engine.start(); }
  private:
    Engine engine;
};
```

aggregation makes more sense

Painful if 2^8 methods to write

Liskov substitution principle

We should be able to substitute derived classes objects by mother class objects without any other modification.

Inheritance semantic

Inheritance againt code copy

Contract programming

Composition aggregation

What about private inheritance ?

```
class Base {
 public:
  Base() {
     cout << "Base::Base()" << endl;
  }
 ~Base() {
     cout << "B
  }
};
class Derived : public Base {
 public:
  Derived() {
     cout << "Derived::Derived()
  }
 ~Derived() {
     cout << "Derived::~Derived()" << endl;
  }
};
```

Virtual destructor !

```
int main(int, char**) {
   Derived * m = new Derived;
   delete m;
}
```

```
> Base::Base()
  Derived::Derived()
  Base::~Base()
>
```

Pure virtual destructor ?

- Make a class abstract
- If you cannot find another one …

```
class Base {
 public:
   Base() {
      cout << "Base::Base()" << endl;
   }
   virtual ~Base() = 0;
};
```

Disrupt inheritance

Need to add a definition



Forbid inheritance ?

- Without using virtual destructorrs
  - Like the containers of the standard library
- You have to read the documentation of the classes

- Keyword `final`
  - Class
  - Method




```
class B {
   void m() final;
};
```

```
class B final {
   void m();
};
```

Constructor & polymorphism

```
class Base {
 public:
  Base() { m(); }
  virtual void m() { cout << "mm" << endl; }
};
```

Derived object not totally constructed

```
class Derived : public Base {
 public:
  Derived() { m(); }
  virtual void m() { cout << "mf" << endl; }
};
```

```
> mm
  mf
>
```

```
Derived f;
```

No call of virtual method

Construction in two steps
init() method

"Virtual" constructor (1)

```
M * f = new Derived;
```

How to create an object of the same type than Derived ?

Runtime type detection mecanismM … OR

```
o = (f->getType()== V1) ? new F1 :
    (f->getType()== V2) ? new F2 :
    …
    nullptr
```

"Virtual" constructor (2)

```
class Base {
 public:
  Base() {}
  virtual ~Base() {}
  virtual Base * create() const = 0;
  virtual Base * clone()  const = 0;
};
```

```
Base * m1 = new Derived();
Base * m2 = m1->create();
```

```
class Derived : public Base {
 public:
  Derived() {}
 ~Derived() {}
  Derived * create() const;
  Derived * clone()  const;
};
```

new Derived()

new Derived(*this)

References

```
class NotAfraid {
    int arr[4];
 public:
    NotAfraid() {
          arr[0] = 1;
    }
    const int & val() const {
           return arr[0];
    }
    void modify() {
          arr[0] = 4;
    }
};
```




```
> 1 1
  1 4
  1 4
>
```

```
int main(int, char **) {
   NotAfraid * p = new NotAfraid;
   int i = p->val();
   const int & j = p->val();
   std::cout << i << " " << j << std::endl;
   p->modify();
   std::cout << i << " " << j << std::endl;
   delete p;
   std::cout << i << " " << j << std::endl;
   return 0;
}
```

inline methods

- It is a shame to use a function/method
  - For a simple action
  - To access a data member

FORBIDDEN to define public data member

- Solution : *inline* method
  - Behaves as a macro

Fast

-O2

Increase of the size code

http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

```
class A  {
 public:
    int getX() const { return x; }
 private:
    int x;
};
```

Implicit Inline

2 ways of implementing IN the header file
Multiple definition error ?

```
class A  {
 public:
    int getX() const;
 private:
    int x;
};

inline int A::getX() const {return x;}
```

Explicit outside Inline

Inline but when ?

- Suggestion
- The compiler chooses



- *Inlining* can be incompatible with some mecanisms

A virtual method cannot be inline !

Reminder : redéfinition

Example.hpp

ODR

```
#ifndef __EXEMPLE
#define __EXEMPLE

int GLOBAL_VAR;

extern int GLOBAL_VAR;

double maximum(double a, double b) {
   return (a<b)?b:a;
}

inline double maximum(double, double ) { }

double maximum(double, double );

#endif
```

+ definition in cpp file

Prototype/declaration on hpp
Definition in cpp

Reminder : function pointer

```
double max(double, double) { return … }
double min(double, double) { return … }

typedef double(*NOM)(double, double);

double (*ptr1)(double, double) = max;
NOM ptr2 = &min;

cout << ptr1(4.0, 5.0);
cout << (*ptr2)(5.0, 12.0);
```

Standard use

Method pointer

```
class A {
   double max(double, double) { return … }
   double min(double, double) { return … }
} a;
typedef double(A::*NOM)(double, double);

double (A::*ptr1)(double, double) = A::max;
NOM ptr2 = &A::min;

cout << a.ptr1(4.0, 5.0);
cout << (a.*ptr2)(5.0, 12.0);
```

standard only

The object or class has to be mentioned …

Often replaced by functors and now by lambdas

Automatic class

```
#include <functional>
```

mem_fun
mem_fun_ref
ptr_fun …

```
vector <string *> numbers;

numbers.push_back (new string ("one"));
numbers.push_back (new string ("five"));

vector <int> lengths (numbers.size());

transform (numbers.begin(), numbers.end(),
  lengths.begin(), mem_fun(&string::size));
for (int i=0; i<5; i++)
   cout << *numbers[i] << " has "
        << lengths[i]  << " letters.\n";
```

http://www.cplusplus.com/reference/functional/mem_fun/

"Inner" class or type

```
class Student {
 public:
   typedef   vector<Class *> class_t;
   class_t& getClass();
   void      swap(class_t&);
 private:
   class_t liste;
};
```

Alias

Code easier to maintain

Outside class

```
      class_t& Student :: getClass() {}
```

Dans la classe

```
void Student::swap(class_t&) {}
```



Outside class use

```
class_t& Student::getClass() {}
```

Not recognized !

`vector<Class *>`

`Student::class_t`

```
void display(Student& e) {
  Student::class_t::iterator it;

  it = e.getClass().begin();
  // …
}
```

Inner class & *template*

```
template<typename T>
void f(const vector<T>& c) {

  vector<T>::const_iterator it = c.begin();
  // …

}
```

Uncomplete specialization

Hint for the compiler
We insure the fact that the type exists
No embiguity with class member

```
typename
vector<T>::const_iterator it = c.begin();
```



Operators ++

```
++a;
```

Prefixed operator
A is incremented **before** use

```
A& A::operator++();
```

```
a++;
```

Postfixed operator
Incréments a **after** use

```
A A::operator++(int);
```

Copy

a++++;

Dummy parameter
to distinguish them

Namespace

- To avoid names conflict
  - Names scope

```
namespace loic {
   template<typename T> class vector {
      //…
   };
}
```

`using` a variable scope :
local (bloc) or global

```
loic::vector<int> lv;
std::vector<int>  sv;
::vector<int>     gv;
```

Persistance ?

- Ability for an object to survive the end of the program
- Save / Restore
- File / SGBD

No mecanism in C++

Conclusion

- We have seen a lot of things
  - Object e& modeling
  - Base C++ and not so basic
  - Bit not everything





- If you do not understand ?
  - C++ 2011, 2014 voire 2017 ou 2020

« There are only two kinds of languages: the ones people complain about and the ones nobody uses ». B.S