Novembre 2025 – version étudiante

C++



loic.yon@isima.fr
https://perso.isima.fr/loic

CLERMONT AUVERGNE INP UCA Isima

⊗ 0 ⚠ 0

ZZ 2 ?

- Cours de C++ sur 2 années
- Attention aux TPs
  - Mise en œuvre du cours
  - Manipulation d'outils : *git*, *valgrind*, *makefile*
  - Tests unitaires

Catch2

- Évaluation par cc
  - **Sur machine** pour la partie C++ à la fin du cours
  - Surprise ?

Plan



1. ++C 5
2. Concepts objets & C++ "de base" 25
3. Subtilités 83
4. Pratique : la chaîne de caractères 110
5. Exceptions 136
6. Généricité 161
7. Bibliothèque standard 188
8. Ça ne se dit pas … 239

Ces transparents ont été conçus originellement par Bruno-Laurent GARCIA, ISIMA
Améliorations successives de David HILL et Christophe DUHAMEL, ISIMA
Remarques éclairées de Violaine ANTOINE et Bruno BACHELET

Caractéristiques générales

- Stroustrup, 1983
- Langage orienté objet
  - Classes au sens du langage SIMULA
  - Variables non objet
  - Fonctions
- Support de la généricité
  - Fonctions
  - Classes
- Traitement d'exceptions
  - Mécanisme évolué de gestion des erreurs



– Normes : 1999 (2003), 2011 (2014), 2017, 2020, 2023, 2026

gcc-12 supporte C++17

NO OBject

C / C++ / ++C ?





CLERMONT AUVERGNE INP UCA Isima

Ajouts par rapport au C

- ✓ Surcharge des fonctions
- ✓ Valeur par défaut de paramètres
- ✓ Paramètres muets
- ✓ Constantes
- ✓ Type référence
- ✓ Nouvelle gestion de la mémoire

- × Opérateurs de transtypage efficaces
- × Gestion des types à l'exécution
- × Généricité avancée
- × Norme 2011 avancée (mouvements)
- × Bibliothèque standard avancée

} vu en ZZ3

Premier programme

```
#include <iostream>

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

int main(int argc, char ** argv)
{
  std::cout << "Hello ZZ" << 2;
  std::cout << std::endl;

  return 0;
}
```

Autres flux : std::cin,std:: cerr, std::clog …

Compilation

- Chaine de compilation identique à celle du C
- Extensions de fichier différentes pour différentiation

.h	.H	.hpp	.hxx	.C	.cxx	.cpp

- Préprocesseur / compilateur / éditeur de lien

g++	clang++

- Compilation à la main ou outil

make	cmake

Exécution

- Résultat de la compilation = un binaire
- Pas de machine virtuelle
  - Pas de portabilité à l'exécution
  - Pas de ramasse-miettes

- Portabilité ?
  - Utilisation de C++ « standard »
  - Compilation par système/architecture

Surcharge (1)

- Réaliser une même action avec des éléments différents

```
void afficher (quelque chose);
```

```
void afficher_entier   (int i);
void afficher_reel     (double d);
void afficher_pointeur (void * p);
```

C

```
int zz = 2;
afficher_entier   (zz);
afficher_reel     (zz * 50.);
afficher_pointeur (&zz);
```


Surcharge (2)

- Forme faible du polymorphisme
  - ⇨ Fonctions de mêmes noms mais avec signatures différentes

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

C++

```
int zz = 2;
afficher (zz);
afficher (zz * 50.);
afficher (&zz);
```


Paramètres avec valeur par défaut

Omettre la valeur courante d'un paramètre

- Déclaration

```
void afficher(string txt, int coul = BLEU);
```

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

- Définition / implémentation

```
void afficher(std::string txt, int coul)  {
   // code
}
```

- Appel

```
afficher("loic");
afficher("loic", ROUGE);
```

Paramètres muets

- Paramètre non nommé
  - Compatibilité d'appel d'une version sur l'autre
  - Eviter les warnings de compilation
  - Respect de conventions (main, par exemple)
- Syntaxe

```
type3 fonction(type1 p1, type, type2 p2);
```

```
v3 = fonction(v1, valeur_inutile, v2);
```

- Exemple :

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

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


Constante littérale en C

```
#define TAILLE 10
int tab[TAILLE];
```

Statique

Je promets de ne pas modifier

```
const int TAILLE=10;

int tab[TAILLE];
```

VLA

```
enum {TAILLE=10};

int tab[TAILLE];
```

Statique

- Rapide
- Vérification de type
- Entier seulement

const à la C



« Je promets de ne pas modifier »

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


Constante littérale en C++

```
#define TAILLE 10
int tab[TAILLE];
```

```
enum {TAILLE=10};

int tab[TAILLE];
```

```
constexpr int TAILLE=10;

int tab[TAILLE];
```

```
const int TAILLE=10;

int tab[TAILLE];
```

- Nouvelle définition du const
- Évaluation à la compilation ou à l'exécution
- Facile à enlever / mettre
- Tout type
- Vérification de type

Référence (1)

```
int  a = 1;
int &r = a;
```

```
std::cout << a << " " << r;
```

```
a = 2;
std::cout << a << " " << r;
```

```
r = 3;
std::cout << a << " " << r;
```

```
std::cout << &a << " " << &r;
```


Référence (2)

```
#include<iostream>

void fv(int a) {
  a = 5;
}

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

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

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

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

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

  return 0;
}
```


Référence (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;
}
```


Référence « constante »

```
int  a = 1;
const int &r = a;
```

r est une référence "constante" sur a

```
std::cout << a << " " << r;
```

```
a = 2;
std::cout << a << " " << r;
```

```
r = 3;
```

Référence en paramètre (1)

- Uniquement le passage par valeur en C

Paramètres en *out* ou *in / out* :
Passer un POINTEUR

Code peu lisible

Source d'erreur

- Référence
  - Pointeur masqué
  - Simule le passage par référence
  - S'utilise comme une variable

Naturel

Référence en paramètre (2)

Efficacité

Notion facile à appréhender ?

Code très lisible

Syntaxe ambiguë avec &

Moins d'erreurs

Appel facilité

Pointeur ?

Une référence est toujours liée à une variable

La référence nulle n'existe PAS !

Gestion de la mémoire dynamique

```
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;
```


Inférence (déduction) de type

2011

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

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

!

Newbie

PROGRAMMATION ORIENTEE OBJET BASIQUE & C++





CLERMONT AUVERGNE INP UCA Isima

POO en C++



- Transposition des classes
- Relations entre classes
  - Agrégation
  - Héritage
  - Association ?
- Exceptions
- Généricité
  - Fonctions
  - Classes

Transposition des classes



- Syntaxe générale
- Déclaration
  - Attributs et prototypes des méthodes
- Définition/implémentation des méthodes
- Cycle de vie des objets
  - Construction
  - Appel de méthodes
  - Destruction

Exemple simplissime

```
#include <iostream>

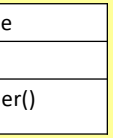
class Exemple {
  public:
  void afficher() {
     std::cout << "hello" << std::endl;
  }
};

int main(int, char **)
{

  Exemple e;

  e.afficher();

  return 0;
}
```





Classe en C++

Point
-x : réel -y : réel - compteur : entier
+ getX() : réel + setX(réel) + deplacerDe(dx : réel, dy : réel) + deplacerVers(px : réel, px : réel) + getCompteur() : entier

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

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

GUIDE DE STYLE ?

TOUJOURS OBLIGATOIRE !

Déclaration (Fichier Point.hpp)


```
#include "Point.hpp"

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

// Setter, méthode set
void Point::setX(double px)
{
  x = px;
}

// méthode
void Point::deplacerDe(double dx, double dy)
{
  x+=dx;
  y+=dy;
}

void Point::deplacerVers(double px, double py)
{
  setX(px);
  setY(py);
}
```

Définition (Fichier Point.cpp)

Manipulation basique (1)

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


int main(int, char **)
{

  Point p;

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

  p.deplacerVers(2., 4. );

  return 0;
}
```

```
> 0.0 0.0
>
```





Constructeur

- Initialiser les attributs d'un objet
  - Même nom que la classe
  - Pas de type de retour
  - Par défaut si non fourni

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

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

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



Valeurs par défaut ?




```
class Point
{
   double x;
   double y;

 public:
   Point();
   Point(double = (
         double = (
};
```

```
class Point
{
   double x = .0;
   double y = .0;

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


Manipulation basique (2)

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

int main(int, char **)
{
  Point   p1;
  Point   p2(5., 6.);
  Point   p3(4);
  Point * pp = new Point(10.,11.);

  Point p4 {};
  Point p5 {2., 3.};

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

  delete pp;

  return 0;
}
```


Membres de classes

Point
-x : réel -y : réel - compteur : entier
+ Point() + getX() : réel + setX(réel) + deplacerVers(px : réel, px : réel) + getCompteur() : entier

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

 private:
   double x;
   double y;
   static int compteur;
};
```

Mot-clé **static**
Membre de classe


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

// attribut en lecture seule
// de l'extérieur
int Point::getCompteur()
{
  // cout << getX();
  return compteur;
}
```

Définition
(Fichier cpp)

```
// définition - initialisation
int Point::compteur = 0;
```



```cpp
#include <iostream>

int main(int, char **)
{

  std::cout << Point::getCompteur();

  Point p(10., 10.);
  std::cout << p.getCompteur();

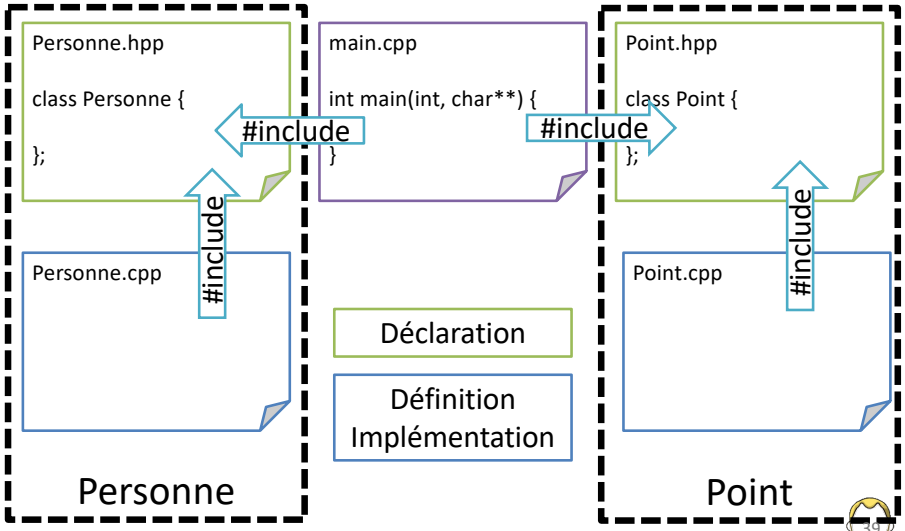


  std::cout << Point::compteur;

  return 0;
}
```


Bonne pratique





On compile et on exécute…

- « Comme en C »
- Tous les fichiers

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

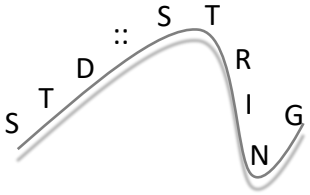
On ne compile JAMAIS un fichier entête (hpp)

- Un makefile !!!

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


Une classe prédéfinie : std::string

- Entête : string
- Chaîne de caractères
  - Encapsulation d'une chaîne C
  - D'autres types (UTF-8 par exemple)
- Relativement fournie
- Manipulation intuitive



  - On apprendra à les coder plus tard

Exemples complets sur https://perso.isima.fr/loic/cpp/string.php

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

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

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



```
if (s2==s3) cout << "egal" ;
if (s2!=s3) cout << "different" ;
if (s2< s3) cout << "plus petite »;
if (s2.compare(s3)==0) cout << "egal" ;
```

Tests

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

Concaténation

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

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

Chaîne C

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


Conversion implicite ?

```
const char *
```

```
"exemple"
```

```
char *
```

```
char [256]
```

```
std:string
```


Flux / *stream*

- Permet de manipuler des entrées / sorties
- Source ou destination de caractères

- Flux standards `cin cout cerr clog`
- Flux fichiers `fstream`
- Flux de chaînes de caractères `stringstream`

Manipulation en TP

Conversion

```
#include <sstream>
```

- Flux de chaînes de caractères

```
std::ostringstream oss;
oss << valeur;

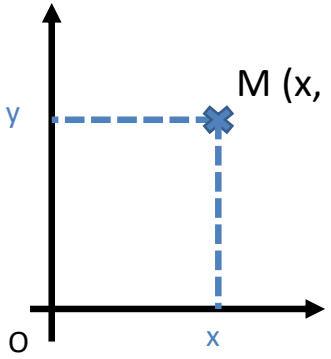
Résultat : oss.str();
```

10.5 -> "10.5"

```
std::istringstream iss(chaine_a_decrypter);
iss >> entier;
```

"8" -> 8

Encapsulation (1)



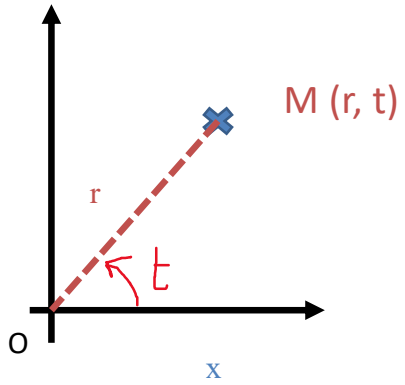
Point
+ x : réel + y : réel
+ Point() + getX() : réel + setX(réel) + getY() : réel + setY(réel)

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

```
Point p;
p.x = 7;
std::cout << p.getX();
```

Code utilisateur

Encapsulation (2)



Point
+ r : réel + t : réel
+ Point() + getR() : réel + setR(réel) + getT() : réel + setT(réel)

```
Point p;
p.x = 7;
std::cout << p.getX();
```


Encapsulation (3)

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

interface inchangée ou augmentée

```
Point p;
std::cout << p.getX();
```

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

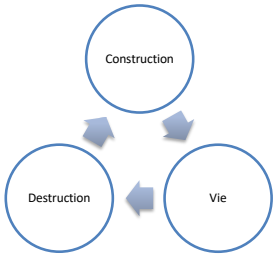
```
// Getter, méthode get
double Point::getX()
{
  return r*cos(t);
}
```

Cycle de vie des objets

① Construction
- Allocation de mémoire
- Appel d'un constructeur

② Vie
- Appel de méthodes

③ Destruction
- Appel du destructeur
- Désallocation : rendu mémoire



Liste d'initialisation

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

- Souple
  - Constantes ou valeurs immédiates
  - Fonctions
  - Expressions arithmétiques
- Dans l'ordre de la déclaration de la classe
- Traitée avant le code

Liste & ordre des initialisations

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

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

Liste & expression complexe

- Ajout d'un attribut calculé *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:
   // Pas d'autre constructeur
   Point(double, double);
};
```

Et si le constructeur n'est pas public ?

```
int main(int, char**)
{
   Point p1(3., .4);
   Point p2;
   Point t[10];
   Point * p = new Point[7];
}
```



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

> Compilation
> Exécution
> ?

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

this

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


Liste & constructeur

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

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

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


Rappel : types d'allocation

Destruction "automatique"

1. Statique
   1. variables globales
   2. variables locales déclarées statiques
2. Automatique : variables locales sur la pile
3. Dynamique : variables allouées sur le tas
   1. Allocation avec **new** + constructeur
   2. Destruction avec **delete**
4. Un autre …

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


int fonction(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[fonction(l)+2];
   fonction(6);
   delete [] m;
   return 0;
}
```



```
class Tableau
{
   int * tab;
   int taille;

 public:
   Tableau():tab(nullptr),taille(10)
   {
       tab = new int[taille];
   }

};

int main(int, char **)
{
   Tableau t;

   return 0;
}
```

> Compilation
> Exécution
>

Valgrind ?



Destructeur ?

- Porte le nom de la classe précédé de ~
- N'est jamais appelé directement
  - Destruction automatique
  - Appel par `delete`
- Libérer les ressources critiques



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

```
Tableau::~Tableau()
{
   delete [] tab;
}
```

TP — Vérification de l'ordre de destruction


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

   return r;
}
```

```
b = fonction (a);
```

Optimisation possible du compilo

Copie or not copie ?

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

   return r;
}
```

```
b = fonction (a);
```

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

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

   fonction(p);

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

> 5.0 2.0
>

1. Copie
2. Constructeurs connus pas appelés

Bilan

```
const int TAILLE = 10;

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

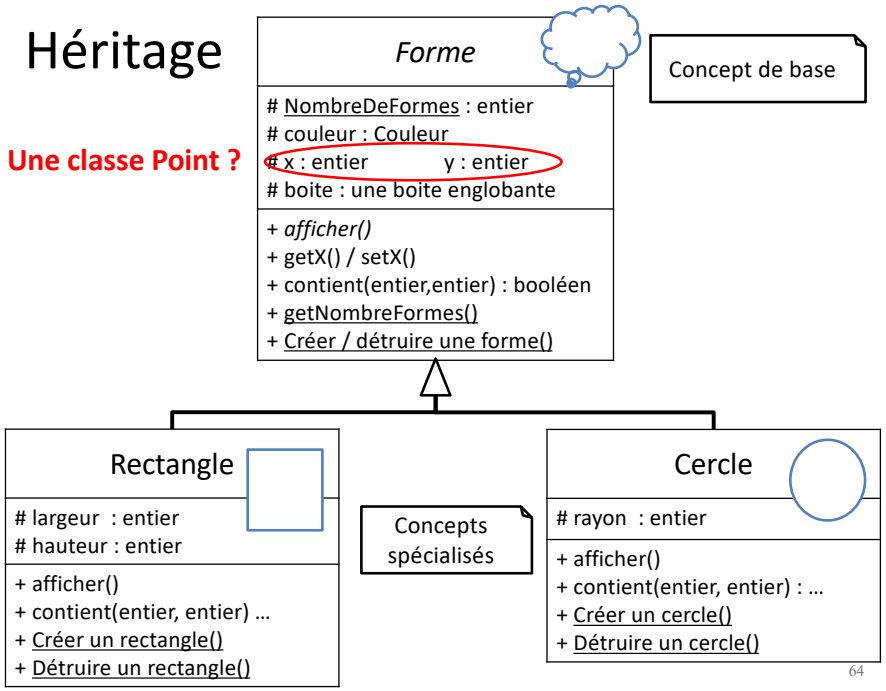
  Point *p;
  p = new Point(10.,25.);
  delete p;

  return 0;
}
```

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


Héritage

Une classe Point ?



Héritage

- Simple ou multiple
- Visibilité des membres
- Plusieurs types d'héritage

public
protected
private

```
class Rectangle : public Forme
{

};
```

Héritage simple public

Visibilité / Accessibilité

- 3 niveaux de protection

Membre mère	Fille	Extérieur
public		
protected		
private		

- Héritage classique : *protected*
- Passer tous les attributs de *private* à *protected* ?


```
class Parent {
 private:
  ✗int portemonnaie;
 protected:
  ✓C c; // chaussure

 public:
    void direBonjour();
    C    voirChaussures();
    int  demanderMonnaie(int);

 protected:
  ✓int  consulterMonnaie();
  ✓void changerChaussure(C);

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



AUVERGNE NOUVEAU MONDE

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



Instanciation d'un objet

```
Cercle::Cercle()
{
}
```

```
Cercle::Cercle():Forme()
{
}
```

```
Cercle::Cercle(int px, int py):
   Forme(px,py), rayon(0)
{
}
```

```
Cercle c1;
Cercle c2(1,2);
```


Polymorphisme

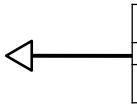
- Une même méthode prend plusieurs formes
- Forme faible
  - Surcharge de méthode – *overloading*
  - Méthodes de signatures différentes

Forme
+ déplacer(x, y) + déplacer(Point)

- Forme forte
  - Redéfinition – *overriding*
  - Actions différentes pour des classes d'une même hiérarchie

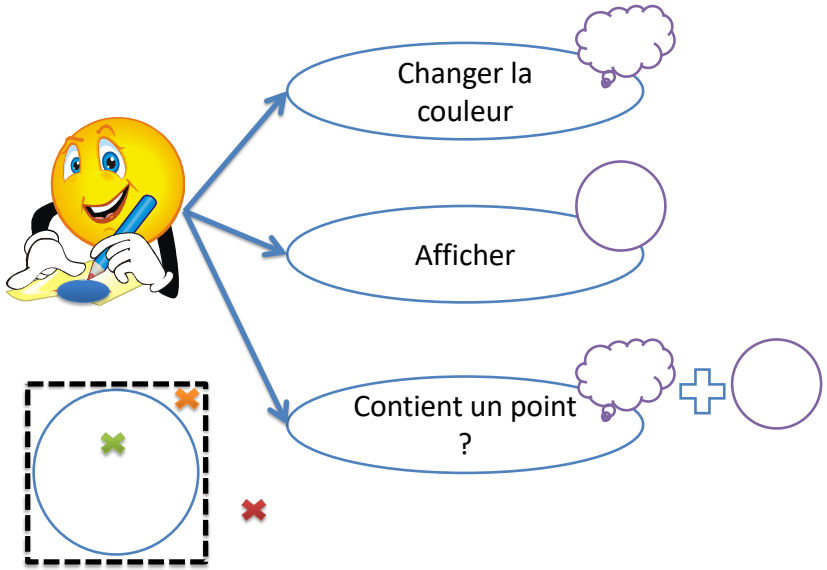


Forme
+ afficher();



Cercle
+ afficher();

Cercle & polymorphisme




```
void Forme::afficher() {
  cout << "Forme" << endl;
}
```

```
void Cercle::afficher() {
  cout << "Cercle" << endl;
}
```

```
int main(int, char**) {
  Forme  f;
  Cercle c;

  f.afficher();
  c.afficher();
  return 0;
}
```

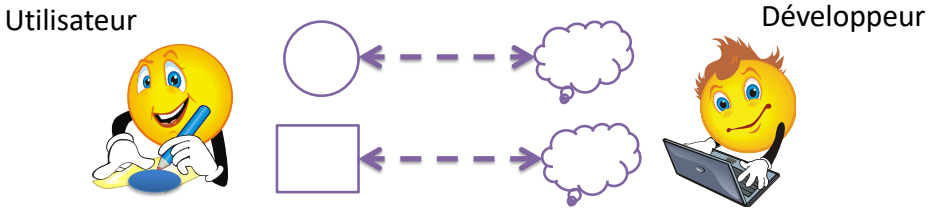



```
bool Forme::contient(int x, int y)
{
  // calculer la boite englobante
  return r;
}
```

```
bool Cercle::contient(int x, int y)
{
  bool r = Forme::contient(x,y);

  if (r)
  {
     // affiner le calcul
  }
  return r;
}
```

bool : true or false




```
int main(int, char **) {
  Forme  * f = new Cercle;

  f->setCouleur(2);

  f->setRayon(10.0);


  f->afficher();

  return 0;
}
```

Méthode virtuelle

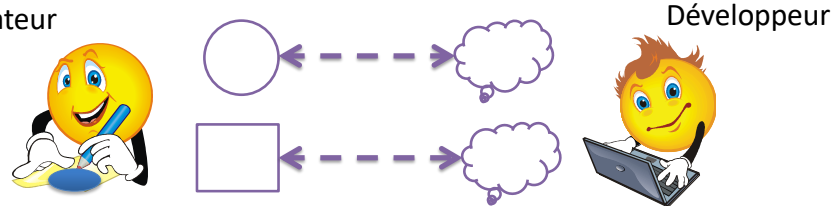
- Détermination à l'exécution
  - Table des méthodes virtuelles
  - Surcoût
- « Virtuelle un jour, virtuelle toujours »



```
class Forme {
   virtual void afficher();
};
```

```
class Cercle : public Forme {
   [virtual] void afficher() [override];
};
```

2011




```
Forme   t1[100000];
Forme * t2[100000];
```

```
t1[0] = Cercle;
t1[0].afficher();
```

```
t2[0] = new Cercle;
t2[1] = new Rectangle;
// …

for(int i=0; i<nombre; ++i)
   t2[i]->afficher();
```

Ajout de Triangle
= pas d'impact



Downcasting

- Classe générale ➡ classe spécialisée
  - Upcasting : pas de problème

Coûteux
si vérification

```
int main(int, char **) {
 Forme   * f = new Cercle();
 f->setRayon(2.0)
 ((Cercle    *)f)->setRayon(2.0);
 ((Rectangle *)f)->setRayon(2.0);
 ((Rectangle *)f)->setHauteur(5.0);
  f->afficher();
  return 0;
}
```

C	E

- Vérification à l'exécution
  - Opérateurs qui empêchent les incohérences en ZZ3

Classe abstraite

- Représente un « concept » très général
- Non instanciable

cannot declare variable 'x' to be of abstract type 'X'

- Au moins une méthode abstraite
  - « Pas d'implémentation »
  - Déclarée virtuelle pure

```
virtual retour methode(paramètres) = 0;
```

  - Code par défaut possible pour appel par les classes filles

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

```
bool Forme::contient(int px, int py) {
  bool r = false;
  if (px>=x) && (px<=x+largeur) &&
     (py>=y) && (py<=y+hauteur))
     r = true;
  return r;
}
```

```
bool Rectangle::contient(int px, int py) {
  return Forme::contient(px,py);
}
```

```
bool Cercle::contient(int px, int py) {
  return r; // déjà vu transparent 72
}
```

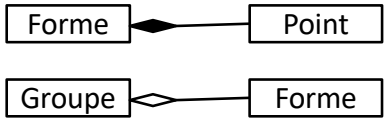
Interface

- N'existe pas en C++
- Classe abstraite pure
  - Pas de méthode concrète
  - Même le destructeur
- Attributs ?
  - Normalement non
  - Constantes ?



Agrégation / Composition

- Un ou plusieurs objets dans un autre
- Implémentation
  - Objet direct
  - Référence
  - Pointeur



- Provenance ou construction de l'objet
  - Objet construit par son agrégateur
  - Objet en provenance de l'extérieur
    - Recopié — Vu plus tard
    - Prise de pointeur ou de référence


```
class Forme
{
  Point point;
};
```

```
Forme::Forme ):point()
{
}
```

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

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

Autres relations entre objets

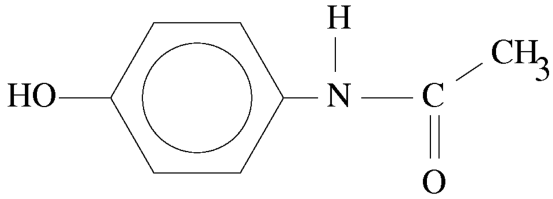
- Même implémentation agrégation/composition
  - Objet
  - Référence
  - Pointeur

Noms significatifs

- Qui se charge de la destruction ?
  - Propriétaire de l'objet
  - Attention aux objets non automatiques

SENSIBLE

SUBTILITÉS C++







CLERMONT AUVERGNE INP ESIGELEC Isima

Objet constant

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

```
r.setLargeur(40);
```

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

Bon, OK, la méthode change l'état de l'objet

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

!! Même punition ??

Méthode constante

- Appelable sur un objet **constant**

```
void Forme::afficher() const
{
    std::cout << "Forme" << std::endl;
}
```

- Mot-clé `const`
  - Garantit que la méthode ne modifie pas l'état de l'objet
  - Appartient à la signature
  - À mettre **dès que possible**

```
void Rectangle::setLargeur(int l) const {
  largeur = l;
}
```

error: assignment of data-member 'Rectangle::largeur' in read-only structure

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

OPTIMISATION ?

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



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

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

Optimisation de *getter* ?

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

```
Std::cout << r.getLargeur();
```

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


CPP *old style*

```
class NinjaWarrior
{
   double x_;

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

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

>>

Retour valide - Attribut modifiable


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

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

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

fonction(10);
```

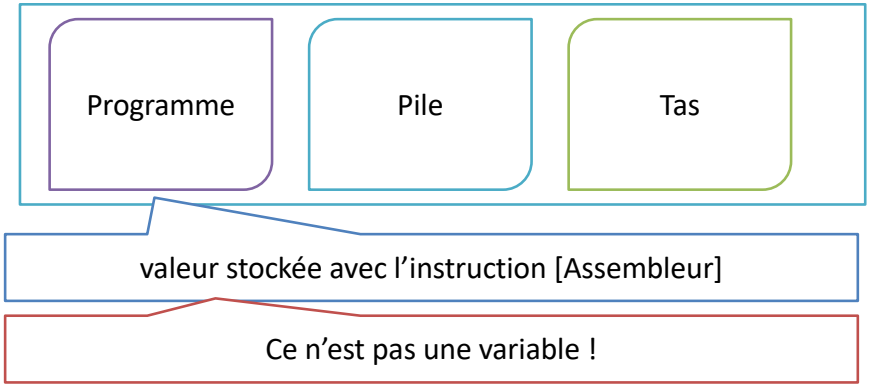
Références & valeurs immédiates

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

   return r;
}
```

Où se trouve la valeur immédiate ?

Mémoire




```
fonction(10);
```

```
int fonction (const int & p)
```


Références & Surcharge ?

```
class NinjaWarrior
{
   double x_;

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

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



```
class Forme {
  public:
   virtual void afficher() { std::cout << "Forme"; }
};
```

```
class Cercle : public Forme {
  public:
     void afficher() override { std::cout << "Cercle"; }
};
```



```
void afficher1(Forme f) {
   f.afficher();
}
```

```
void afficher2(Forme& f) {
   f.afficher();
}
```

```
void afficher3(Forme* p) {
   p->afficher();
}
```

```
int main(int, char**) {
 Cercle cercle;

 afficher1(cercle);
 afficher2(cercle);
 afficher3(&cercle);

 return 0;
}
```

```
>
>
```

Troncature de type


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

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

   fonction(p);

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

```
>
>
```



Constructeur de copie

- Création d'un nouvel objet par clonage
  - Explicite
  - Passage par valeur
  - Retour de fonction (compilo ?)

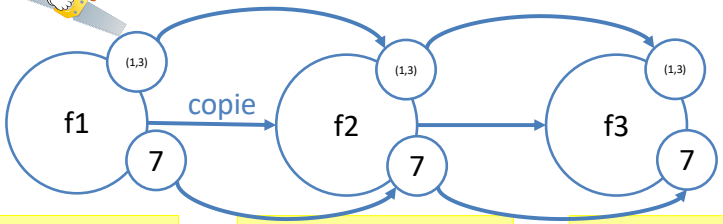
```
Forme f1;
```

```
Forme f2(f1);
Forme f3 = f1;
```

```
Forme::Forme(Forme) {
}
```




Copie par défaut d'un composé




```
Forme f1;
```

```
Forme f2(f1);
```

```
Forme f3 = f2;
```

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

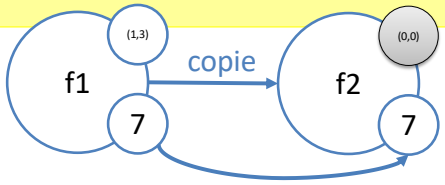



Copie d'un composé

TP

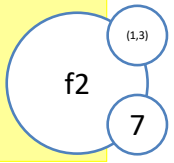
Forme — Point

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




```
Forme f2(f1);
```

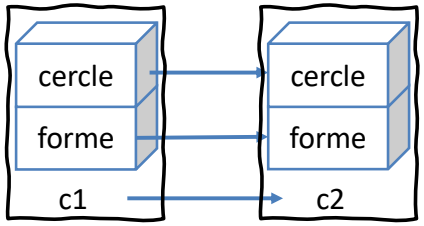
```
Forme::Forme(const Forme & f):
  p(f.p) {
  //
}
```





Copie par défaut et héritage

```
class Cercle : public Forme {
};
```




```
Cercle c1;
```

```
Cercle c2(c1);
```



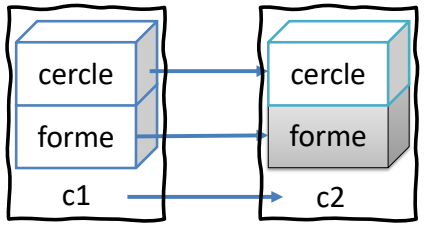
Copie et héritage

TP

```
Cercle::Cercle(const Cercle & c) {
  // code
}
```

```
Cercle c1;
```

```
Cercle c2(c1);
```



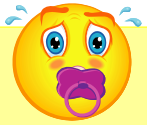

```
Cercle::Cercle(const Cercle & c) :
  Forme(c) {
  // code
}
```

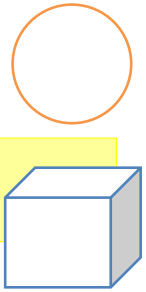



Copie - bilan

- Copie binaire quand les éléments sont simples
- Le constructeur de copie par défaut fait ce qu'il faut
  - Agrégation/composition
  - Héritage
- Constructeur de copie « personnel »
  - Construction par défaut sauf mention contraire

Conversion





```
Cercle c;
Rectangle r = c;
```

- Constructeur spécifique

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

- Éviter la conversion implicite

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

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


Affectation

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

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

c3 = c2 = c1;
```

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

Affectation par défaut

Si vous n'écrivez pas d'opérateur d'affectation :

- Copie binaire pour les types non objets
- Appel par défaut de l'opérateur= de la classe mère
- Appel par défaut à l'opérateur= des objets agrégés / composés

Dire ce qu'il y a à faire si vous écrivez l'opérateur …

L'objet courant existe déjà



Synthèse : références & objets

- Toujours passer les paramètres objets par référence
  - Évite un appel au constructeur de copie
  - Garantit les appels polymorphiques
- Référence constante pour objets non modifiables
- Ne pas renvoyer de référence lorsque l'on doit renvoyer un objet !

Pour les types primitifs constants,
la copie reste acceptable !

Eviter la duplication de code ?



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





```
class Essai {
   static int i ;
   string nom;

 public:
   const string & getNom(int j) const {
      i+=j;
      return nom;
   }
  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; }
};
```

Vocabulaire

- Déclaration
- Prototype

- Définition
- Implémentation

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

extern B b;
```

.hpp

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

B b;
```

.cpp

Déclarations anticipées / *forward*

```
void f();

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

void f(){
}
```

```
class B;

class A {
  B * b;
};

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


Déclaration de classe anticipée

- Utiliser un pointeur
- Utiliser une référence

- Utiliser un objet
- Dériver une classe

```
class B;

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

class C : public B {
};

class B {
};
```



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

Combien de fois est lu A.hpp lors de la compilation ?

A.cpp

A.hpp

#include

Gardiens =

main.cpp

B.cpp

B.hpp

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

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

Pratique

IMPLÉMENTER LA CHAÎNE DE CARACTÈRES





Plan



- TD : la chaîne de caractères
  - Subtilités du C++
- Les opérateurs par l'exemple
  - Affectation
  - Crochets
  - Redirection
  - Incrémentation
  - Tests

  } Amitié
- Synthèse : la forme normale de Coplien

Classe `Chaine`

- Gestion dynamique de la "longueur" de la chaîne de caractères C
- Entête **`cstring`** utilisable

Chaine
- capa : entier - tab : tableau de char
+ Chaine() + Chaine(taille : int) + Chaine(s : char *) + Chaine(s : Chaine) +~Chaine() + opérateur d'affectation + extraction de caractère + affichage + accesseurs qui vont bien


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

   Chaine s1;
   Chaine s2(6);
   Chaine s3("essai");
   Chaine s4(s3);
   Chaine s5 = s3;
   s2 = s3;
   s3.remplacer("oups");
   s3.afficher(std::cout);
   s2.afficher();
   std::cout << s3.toCstr();
   std::cout << s3[0];
   std::cout << s3;

   return 0;

}
```



```
Chaine s1;
Chaine s2(6);

s1.afficher(std::cout);
```

0 par défaut ?

```
Chaine::Chaine (int pCapa) :
    capa(pCapa>0 ? pCapa+1 : 0) {
  if (capa) {
    tab = new char [capa];
    tab[0] = 0;
  } else tab=nullp
}
```

```
void Chaine::afficher(std::ostream & o) const {
  if (tab) o << tab;
}
```

```
Chaine::~Chaine() {
   delete [] tab;
}
```



```
Chaine s3("essai");
```

```
Chaine::Chaine (const char *cstr) {


}
```

```
Chaine s2 = "autre essai";
```

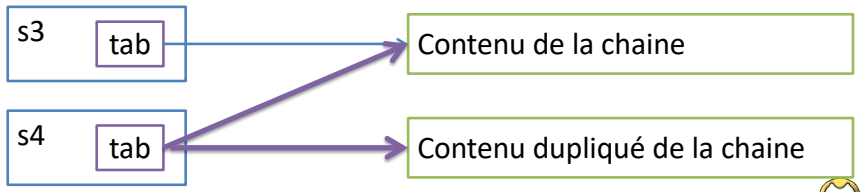


```
Chaine s4(s3);
Chaine s5 = s3;
```



```
Chaine (const Chaine &uC) {


}
```




```
cout << s5.toCstr();
s3.remplacer("zz");

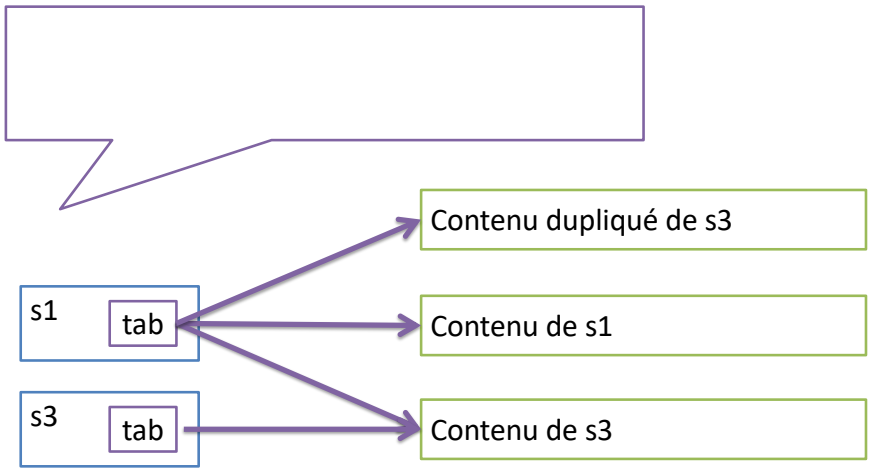
cout << s3.getCapacite();
```

```
const char * Chaine::toCstr() const {
  return tab;
}
```

```
int Chaine::getCapacite() const {
  return tab?capa-1:0;
}
```

```
void Chaine::remplacer(const char * c) {
  strncpy(tab, c, capa);
  tab[taille-1] = 0;
}
```

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




```
s1 = s3;
s1.operator=(s3);
```

```
Chaine& Chaine::operator=(const Chaine &uC) {

  return *this;
}
```



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

at ()

```
char& Chaine::at(int index)
{



  return tab[index];
}
```



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

at ()const

```
const char& Chaine::at(int index) const
{



}
```

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

[ ]

```
char& Chaine::operator[](int index)
{




}
```



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

[ ] const

```
const char& Chaine::operator[](int index) const
{




}
```



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

<<

```
std::ostream& operator<<
            (std::ostream &o, const Chaine &c)
{
  if (c.toCstr()) o << c.toCstr();
  // OU c.afficher(o)
  return o;
}
```

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



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

FONCTION

```
Chaine operator+(const Chaine &a,const Chaine& b)
{
   char *tab =
    new char [a.getCapacite()+b.getCapacite()+1];
   strcpy(tab, a.toCstr());
   strcat(tab, b.toCstr());
   Chaine temp(tab);
   delete [] tab;

   return temp;
}
```

Synthèse : opérateurs

- Usage habituel des objets
- Tous sont redéfinissables même si ce n'est pas souhaitable
- Méthode
  - Si l'objet courant a un rôle important
  - Opérateurs unaires
  - Affectation
- Fonction
  - Rôle symétrique des paramètres
  - Opérateurs binaires
  - Déclaration de classe non modifiable



Aller plus loin : https://en.cppreference.com/w/cpp/language/operators



Amitié (2)

- Avantages
  - Possibilité de rendre amies des classes fortement liées (notion de package)
  - Efficacité du code
- Inconvénients
  - Violation du principe d'encapsulation
  - Possibilité de faire n'importe quoi au nom de l'efficacité
  - Attention à l'intégrité de l'objet lorsque l'on accède aux attributs !

Quelques opérateurs en mode *friend*



Amitié (1)

- Accéder aux membres privés directement ?
- Classe ou fonction friend
- Déclarée dans la classe
- Non transitive et non héritable

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



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



```
class Chaine;

ostream& operator<<(ostream&, const Chaine&);

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

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

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



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

```
Chaine operator+(const Chaine &a,const Chaine& b)
{
   Chaine res(a.taille+b.taille-1);
   strcpy(res.tab, a.tab);
   strcat(res.tab, b.tab);
   return res;
}
```




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

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

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

Opérateur d'égalité
FONCTION

On peut écrire tous les opérateurs de comparaison

Forme Canonique de Coplien & Règle des 3



```
class C {
     C();

     C(const C&);

     [virtual]~C();

     C& operator=(const C&);

};
```

FNC 2011



- FCC 2003 ⊊ FCC 2011
  - Sémantique de déplacement (ZZ3)
  - Règle des 5

- Opérateur / méthode par défaut
- Suppression d'opérateur ou de méthode

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

```
int main(int, char **) {
      C    c1;
      C    c2 {};
      C    c3(10);
      C    c4 {20};
      C    c5(c1);
      C    c6 = c2;
      C    c7();
      C();

      c1 = c4 = c2;

      return 0;
};
```



Who's who ?



Pour conclure…

- Classe « complexe » : tableau dynamique de char
- Utilisation intensive
  - Surcharge d'opérateurs
  - Concepts fins du C++
  - Forme Canonique de Coplien / règle des 3
- Base pour les « conteneurs »
  - Objet qui en contiennent d'autres
  - Vecteur = tableau dynamique d'éléments
  - Liste chaînée
  - Tableau associatif
- Gestion des erreurs ?

Error free

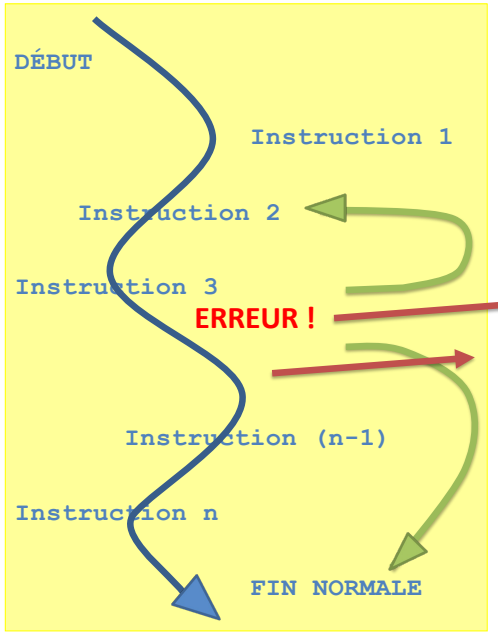
GESTION DES ERREURS : EXCEPTIONS







CLERMONT AUVERGNE INP Isima

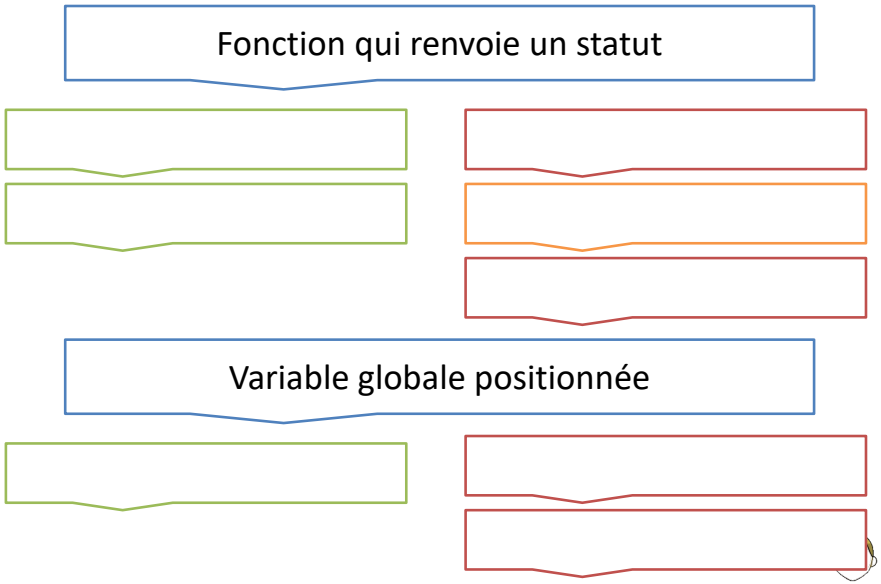


Détection ?

Sortie ?

- Retour en arrière
- Immédiate ?
- Différée !
- "Normale"

Détection classique !

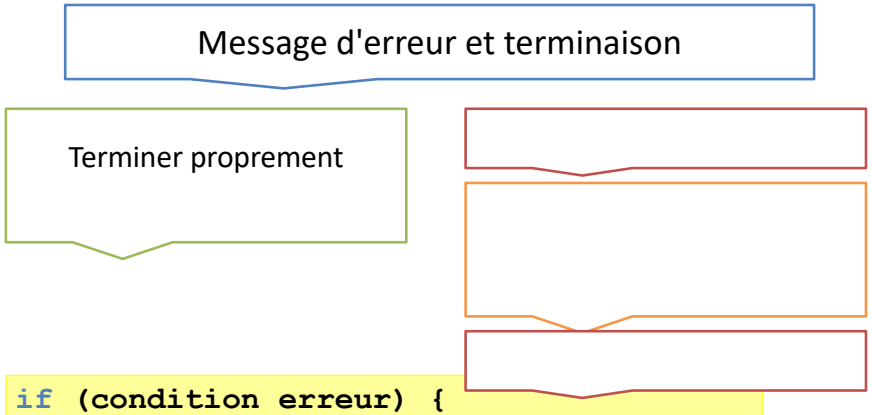



```
int fonction(parametres formels) {
  if (ConditionErreur1)
    return CONSTANTE_ERREUR1;
  if (ConditionErreur2)
    return CONSTANTE_ERREUR2;
  …
  return CONSTANTE_SUCCES;
}
```

```
switch (fonction(parametres effectifs)) {
  case CONSTANTE_ERREUR1:
    …
    break;
  case CONSTANTE_ERREUR2:
    …
   break;
  case CONSTANTE_SUCCES:
   …
   break;
}
```

Exemple de mécanisme classique

Sortie de programme ?




```
if (condition erreur) {
  std::cerr << "Message d'erreur"
            << std::endl;
  std::exit(1);
}
```


Fin du programme

- « Propre »
  - Fichiers fermés
  - Ressources libérées
  - Destructeurs statiques appelés

fin naturelle ou `exit()`

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

- Non prévue : exception non traitée
  - Aucune ressource libérée
  - `abort()` appelée par défaut
  - `std::terminate()` appelable directement
  - Changeable

`terminate()` `unexpected()`

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

```
class Bavarde {
   std::string nom;
 public:
   Bavarde(std::string n):nom(n) {
      std::cout << "constructeur " << nom << std::endl;
   }
   ~Bavarde() {
      std::cout << "destructeur "  << nom << std::endl;
   }
};

Bavarde g("global");

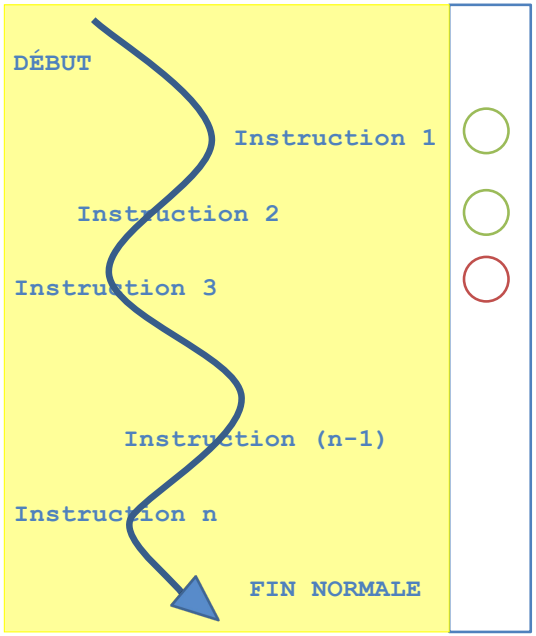
int main(int, char **) {
   Bavarde        t("local");
   static Bavarde s("statlocal");

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


Exceptions

- Gestion d'erreurs ou de comportements exceptionnels
- Lancement d'exceptions
  - Théoriquement tout type de données
  - Habituellement des classes spécialisées
- Traitement des exceptions
  - Blocs de codes surveillés
  - Gestionnaires d'exceptions : code spécialisé





Création de l'exception (**lancer**)

Traitement (attraper) Ou relance

Sortie du programme ?

Exemple

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

  tab = new int [BEAUCOUP];

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

  delete [] tab;

}
```

Attraper une exception…

```
void f() {
  int * tab = nullptr;
  try {
     tab = new int [BEAUCOUP];
     for(int i =0; i< BEAUCOUP; ++i)
        std::cout << tab[i] << " ";
     delete [] tab;
  } catch (std::bad_alloc &e) {
     std::cerr << e.what();
  }

}
```


Classe imbriquée

```
class ClasseEnglobante
{
  public:
    class ClasseInterne{};
};
```

- A l'intérieur de la classe
  - Usage normal
- A l'extérieur de la classe

```
ClasseEnglobante::ClasseInterne instance;
```




Autre exemple

```
bool redo = true;
do {
  cout << "entrer indice valeur" << endl;
  cin >> i >> v;
  try {
      chaine[i] = v;
      redo = false;
  } catch(const Chaine::ExceptionBornes &e) {
      cerr << "ERREUR : recommencez :";
  }
} while (redo);
```



Conseil :
référence constante



Exception standard

- Classe spécifique de la bibliothèque standard

```
#include <exception>
```

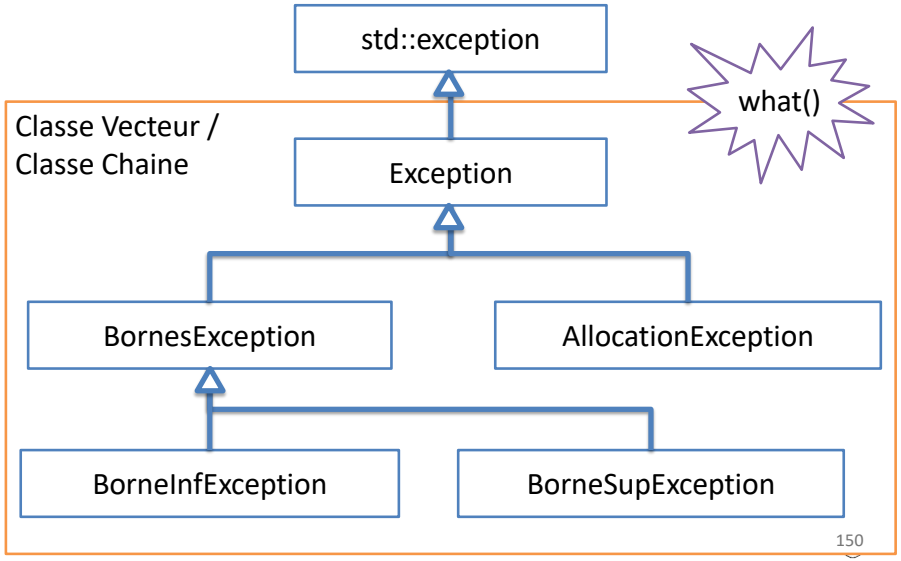
```
std::
```

- Classe mère de toute exception
  - à spécialiser
  - Exceptions à hiérarchiser

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

Description incomplète

Hiérarchie des exceptions (1)



Traitement des hiérarchies d'exception

- Toujours traiter les exceptions les plus spécialisées d'abord
- Utiliser un objet passé par référence constante
  - Pour éviter une recopie
  - Respect du polymorphisme
- Prévoir un traitement pour `std::exception`
- Ramasser tout ce qui reste

"auto" n'est pas utilisable

Spécialisé

Général

```
try {
 // Code susceptible de lancer une exception
}
catch (const Vecteur::AllocationException &e)
{
  // Traitement spécialisé
}
catch (const Vecteur::BorneInfException &e) {
  // Traitement spécialisé
}
catch (const Vecteur::Exception
  // Traitement plus général
}
catch (const Vecteur::BorneSupException &e) {
  // Traitement plus général
}
catch (const exception& e) {
  // Traitement sommet hiérarchie
} catch (...) {
      // Fourre tout
}
```


Chaine exemplaire

```
class Chaine {
  public:

    class ExceptionBornes {};
                  : public std::exception {};

    char& operator[](int index) {

      if ((index < 0) || (index > taille))
        throw ExceptionBornes();

      return tab[index];
    }
};
```

Relancer une exception

- Retour à la normale impossible
  - Traitement sur plusieurs niveaux
  - Terminaison du programme ?



```
try {
   c = chaine[3];
}
catch (const Chaine::ExceptionBornes &e) {
   // traitement
   if (…) throw;
}
```

Eviter la troncature !

Code garanti sans exception

noexcept

```
type f().    noexcept;
type C::m()  noexcept;
```

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

Description complète



Quelques règles

- Constructeur
  - Peut lever une exception
  - Alloue les ressources
- Destructeur
  - Ne devrait jamais lever d'exception
  - Rend les ressources

Ressource
Acquisition
Is
Initialization

- Catch « fourre tout »
- Comportement du new

new()

- Comportement normal

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

- Modification de comportement
  - Renvoie **nullptr** en cas d'erreur

#include <new>

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

```
class AllocConvenable {
  public:
    AllocConvenable(int n):t1(nullptr), t2(nullptr) {
        try {
            t1 = new double[n];
            t2 = new double[n];
        }
        catch ( const std::bad_alloc & ) {
            delete [] t1;
            throw;
        }
    }
    ~AllocConvenable() {
        delete [] t2;
        delete [] t1;
    }

 private:
    double * t1, *t2;

};
```

RAII

```
try {
  AllocConvenable a;
  // utilisation de a
} catch (const bad_alloc& e) {
   cerr << "objet non construit";
}
```



```
class C {
   int * att;
 public:
   C(const C1);
   C& operator=( … );
   void swap(C&);
}
```

RAII

```
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;
}
```




Conclusion exceptionnelle

Exception = mécanisme de gestion des erreurs

Rigoureux

Ne peut être ignoré !

Coûteux en CPU

Taille de l'exécutable



Fausse impression de sécurité

Toute exception non prise en compte
entraîne la fin du programme

template <typename T>

GÉNÉRICITÉ









Généricité

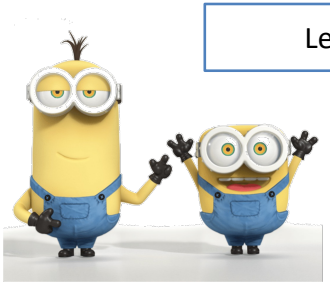
- BUT : écrire du code "générique"
  - Mécanisme orthogonal au paradigme objet
- Abstraction
  - Structures de données
  - Algorithmes
- Quoi ?
  - Fonctions paramétrées
  - Classes paramétrées
- Par quoi ?
  - Des types
  - Des constantes



Patron / gabarit / modèle

Template / generics

Calculer le maximum



Le plus grand de 2

Max (1) : fonctions dédiées

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

Surcharge

Vérification de type

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


Max (2) : macro

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

Max (3) : fonction paramétrée

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

Pas de conversion automatique de type

Code réutilisé

Performance possible grâce à l'inlining

Instanciation de *template*

```
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);
```

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

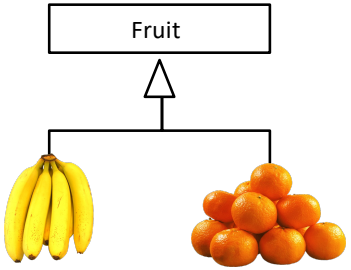
std::max existe !



```
class CaisseBanane {
  Banane * bananes;
  CaisseBanane(){}
  void ajouter(Banane b){}
  Banane enlever() {}
};
```

```
class CaisseOrange {
  Orange * oranges;
  CaisseOrange(){}
  void ajouter(Orange o){}
  Orange enlever() {}
};
```







```
class CaisseFruit {
  Fruit* fruits;
  CaisseFruit(){}
  void ajouter(Fruit f){}
  Fruit enlever() {}
  void porter()
};
```



```
template class Caisse<typename T> {
  T* f;
  Caisse(){}
  void ajouter(F f){}
  F enlever() {}
};
```

```
Caisse<Banane> caisseBananes;
```





Comment et où écrire une classe générique ?

- Écrire la classe non paramétrée
  - 1 type donné
  - 2 fichiers : déclaration (hpp) et définition (cpp)
  - **Tester** cette classe
- Fusionner les deux fichiers pour obtenir un seul **fichier d'entête**
  - Pas de fichier d'implémentation
  - Pas d'erreur de définition multiple



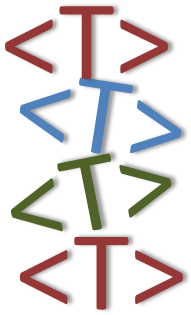
- Remplacer le type précis par le paramètre *template*
- Instancier le *template*
  - Seul un *template* instancié est compilé

Pile générique ?

- Structure de données classique
- Simplification : tableau de taille fixe



Pile
- tab : tableau d'entiers - taille : entier - capacité : entier
+ Pile(capacité : entier) + ~Pile() + top() : entier + isEmpty() : booléen + pop() + push(entier)



Utilisation de la pile

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

using namespace std;

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

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

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

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

  private:
    int   taille;
    int   capacite;
    int * tab;
};
```

Pile.hpp

```
Pile::Pile(int capa) : taille(0),
      capacite(capa),tab(nullptr) {
  tab = new int[capacite];
}
const int& Pile::top() const {
  return tab[taille-1];
}
bool Pile::isEmpty() const {
  return (taille==0);
}
void Pile::pop() {
      --taille;
}
void Pile::push(const int& el) {
  tab[taille]=el;
  ++taille;
}
```

Pile.cpp

```
~Pile() {
  delete [] tab;
}
```

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

  private:
    int   taille;
    int   capacite;
    T * tab;
};
```

Mention *template*

Pas là !

PileGen.hpp

Supprimer le type fixe où il apparaît

Pas là !

```
template<typename T >
PileGen<T> ::PileGen(int capa) :
      capacite(capa),taille(0) {
  tab = new T [capacite];
}

template<typename T >
const T & PileGen <T> ::top() con
  return tab[taille-1];
}

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

PileGen.hpp (suite)

~PileGen()
pop()
isEmpty()
à faire

Ce code est un modèle de classe = non compilé

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

int main(int, char **) {
  using PileEntiers = PileGen<int>;

  PileEntiers p1;

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

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

INSTANCIATION
⇒ Le template instancié est compilé

Tester avec d'autres types !!

PileGen<Point>
PileGen<Forme *>

Constante en paramètre template !

```
#ifndef __PILEGENSTAT_HPP__
#define __PILEGENSTAT_HPP__

template <class T, const int TAILLE = 256 >
class PileGenStatique
{
  public:
    PileGenStatique() : taille(0) {}
   ~PileGenStatique() {}
  private:
    T tab[TAILLE];
};
#endif
```

Tout le reste est identique !

Instanciation & contrôle de type

```
PileGenStatique<double>    pgs1;
PileGenStatique<int,  512> pgs2;
PileGenStatique<int,  512> pgs3;
PileGenStatique<int, 1024> pgs4;
```

```
pgs2 = pgs3;
```

```
pgs1 = pgs2;
```

```
pgs2 = pgs4;
```

```
pgs4 = pgs2;
```


PILE DYNAMIQUE	PILE STATIQUE
Réallocation possible	Réallocation impossible
Taille de l'exécutable	Taille de l'exécutable
Lenteur inhérente au `new()`	Rapidité d'exécution car la mémoire est allouée dans l'exécutable

Un ou plusieurs paramètres ?

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

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

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

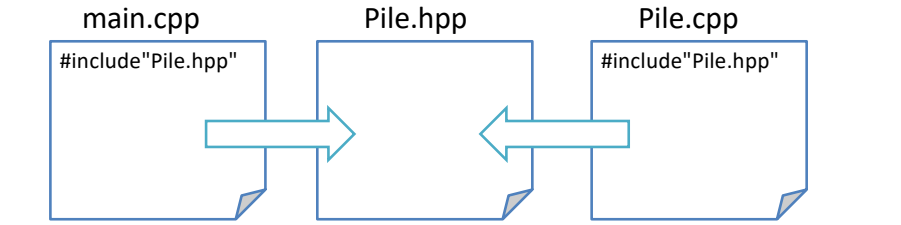
```
Bureau< Livre, PileGen<Livre> >
```


Les opérateurs de comparaison : des fonctions très pratiques…

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

Avec == et < ou > , on peut tout écrire !

Template et compilation




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

Version non générique

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

Version générique

Un fichier entête mais comment ?

- 1 fichier

Fouillis ?

Pratique

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

Méthodes déportées

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

Méthodes dans la classe « inline»

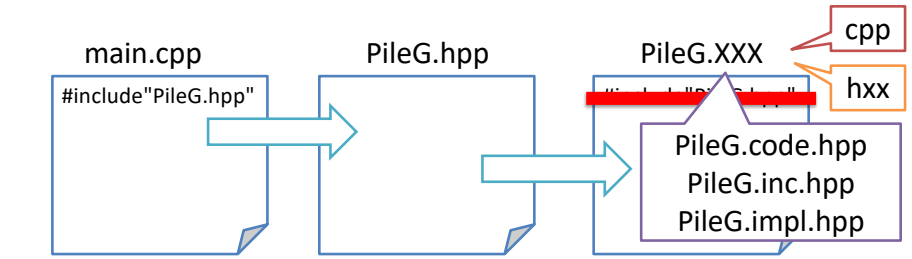
- 2 fichiers

Propre

Compliqué

Usuel

Template avec 2 fichiers



PileG.cpp — Extension reconnue par l'éditeur — compilé automatiquement par make ?

PileG.hxx — Extension pas toujours reconnue Editeur à configurer — Pas compilé automatiquement

Code dans l'entête MAIS pas d'alerte de définition multiple



Introduction

- Besoins récurrents :
  - structures de données : tableau statique ou dynamique, vecteur, pile, file, ensemble…
  - algorithmes : chercher, trier, insérer, extraire…
- Sans réinventer la roue
  - temps perdu (codage, débogage, optimisation)
  - utiliser l'existant (bibliothèques)
- Tous les langages modernes ont une bibliothèque
  - java, C#, perl, python
  - C++

Pourquoi utiliser la bibliothèque standard ?

- Fiabilité
  - collection de classes largement utilisées
- Portabilité
  - totale => standard
- Efficacité
  - Utilisation intensive de la généricité
  - Structures de données optimisées
- Compréhensibilité
  - Conventions d'utilisation uniformes

Contenu

- C standard
- Classes de base
  - Flux, string, stringstream, exception
  - Traits : caractères et numériques
  - unique_ptr, shared_ptr, weak_ptr
- Conteneurs et itérateurs
- Algorithmes et foncteurs

} "STL"



Exemple

- Déclarations et initialisations
- Lecture d'entiers sur l'entrée standard
- Tri de la saisie
- Affichage de la liste triée

- Conteneur
  - Contenir des objets
- Itérateur
  - Parcourir un conteneur
  - Début, fin élément courant, passer à élément suivant



Exemple en C/C++

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




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

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

  qsort (array, n, sizeof(int), cmp);             // Tri

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

 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;                             // Déclarations
  int input;

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

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

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

  return 0
}
```

```
#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;
}
```



Déclarations

Saisie

Tri

Affichage


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

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

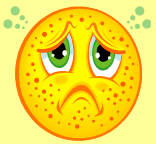
  std::istream_iterator<int> start(cin);
  std::istream_iterator<int> end;
  std::back_insert_iterator<std::vector<int>> dest(v);

  std::copy (start, end, dest);

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

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

  return 0;
}
```



Déclarations

Saisie

Tri

Affichage

Trois types de conteneurs

- Séquences élémentaires
  - Tableau, vecteur, liste et file à double entrée
- Adaptations des séquences élémentaires
  - pile, file et file à priorité
- Conteneurs associatifs
  - ensemble avec/sans unicité
  - association avec clé unique/multiple
  - Avec ou sans ordre sur les clés/éléments

`std::`

Utilisation intensive de la généricité

Fonctionnalités communes

- Forme Canonique de Coplien (étendue au C++2011)
- Dimensionnement automatique de la capacité
  - lorsque l'insertion d'un élément viole la capacité
  - doublement de la capacité
  - adaptation rapide à la taille « finale »

Sauf `array`

`size_type` souvent `std::size_t`

```
st   C::size() const      // nombre d'éléments
st   C::max_size() const  // nombre max
bool C::empty() const     // prédicat de vacuité
void C::swap(C &)         // échange de contenu
void C::clear()           // purge
```

Commun aux séquences élémentaires

- 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)
```

- Suppression

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

- Accès en bordure de la séquence

```
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
```


Array)- tableau de taille fixe

2011

```
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
```

Accès en O(1)

Non redimensionnable

Homogénéisation d'interface

Vecteur : tableau de taille variable

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

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

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

Accès en O(1)

Ajout en O(1) amorti

Insertion / suppression en O(n)


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

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

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

  std::cout << (v1 == v2) ? "Ok" : "Pas Ok"
            << std::endl;

  return 0;
}
```

Liste(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
```



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

  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

  // affiche le début et la fin
  std::cout << l.front() << ' ' << l.back()
            << std::endl;

  return 0;
}
```


File à double entrée

Vecteur

```
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
```

Insertion / suppression en O(1)

Tous les avantages de vector



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

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

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

  std::cout << (v1 == v2) ? "Ok" : "Pas Ok"
            << std::endl;
  return 0;
}
```

Problèmes ➔ itérateurs

- Parcourir un conteneur SOUVENT

Pointeur sur élément courant dans le conteneur

Un seul parcours à la fois

Encapsuler le pointeur à l'extérieur

- Parcours différents (avant, arrière) PARFOIS

Définir la stratégie dans le conteneur

Un seul parcours à la fois

Encapsuler le pointeur à l'extérieur

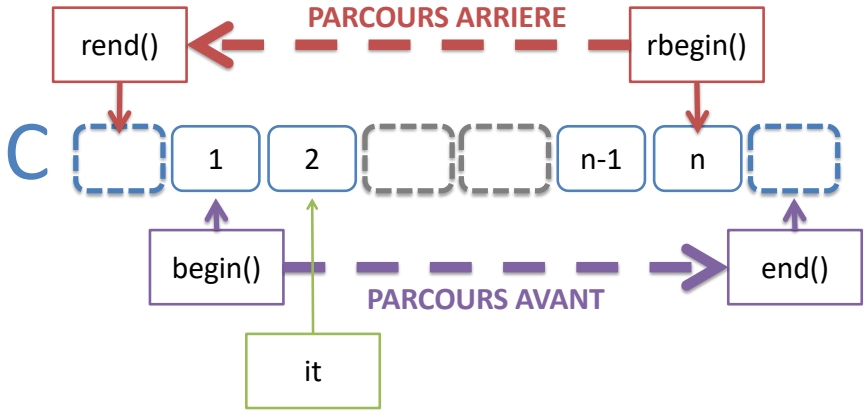
Itérateur

- Définit l'accès à un élément
  - Un pointeur le plus souvent
  - Implémentation du conteneur connue
  - Classe imbriquée
- Définit la stratégie de parcours

```
class conteneur {
 public:
  class iterateur {
  };
};
```

```
conteneur::iterator
conteneur::const_iterator
conteneur::reverse_iterator
conteneur::const_reverse_iterator
```

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

FNC	Incrément ++ pré et post	Déréférencement	Comparaison == !=
it=c.begin()	++it;	*it	it!=c.end()

Utilisation des itérateurs

- Parcours d'un conteneur

```
conteneur c;
conteneur::iterator it;
for (it = c.begin(); it != c.end(); ++it)
  // faire quelque chose avec (*it);
  // it->afficher(std::cout);
```

- Un élément particulier (résultat de recherche)

```
conteneur::iterator it =
    find(c.begin(), c.end(), elt);
cout << (it!=c.end())?"trouve":"inconnu";
```

*it == elt

Écrire sur un flux (1)

```
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();
// auto it  = source.begin();
while(it != source.end() ) {
   o << *it << s;
   ++it;
}
```

std::copy()

Écrire sur un flux (2)

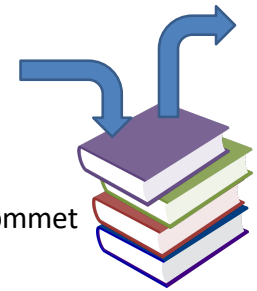
```
#include<iterator>
#include<algorithm>
```

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

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


Conteneur "adapté"

- Conteneur de base
- Avec une interface différente



Pile
Accès au sommet

Tableau
Accès à toutes les places



Conteneur adapté : pile

```
#include <stack>
```

C<T> : deque par défaut
vector,list possibles

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

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

```
int main(int, char **) {
  std::stack<int> is;
  std::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;
}
```


Conteneur adapté : file

```
#include <queue>
```

C<T> : deque par défaut
vector,list possibles

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

```
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 ();
```



```
int main(int, char **) {
  std::queue<int> iq;
  std::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;
}
```


Conteneur adapté : file à priorité

```
#include <queue>
```

C<T> : vector par défaut
deque,list possibles

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

```
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 ZZ {
  string nom, prenom;
  double note;
  // …
};
```

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

```
using  vzz = std::vector<ZZ>;
vzz zz;
// zz.push_back(ZZ(…));

std::priority_queue<ZZ> tri;
for(vzz::iterator it = zz.begin();
    it!=zz.end(); ++it)
 tri.push(*it);

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

Conteneur de pointeurs

```
using vzz = std::vector<ZZ *>;

vzz zz;

for(int i=0; i< 100; ++i)
   zz.push_back(new ZZ());
```

?? Vider le vecteur ??

```
zz.clear();
```

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


Conteneurs associatifs triés

- Eléments triés sur une clé
  - Ordre naturel
  - Comparateur
  - Arbres
  - Clé non modifiable
- Une seule clé
  - Unique
  - Multiple
- Une paire clé-valeur
  - Clé unique
  - Clé multiple

Opérations en O(log n)

```
std::set<cle>
std::multiset<cle>
```

```
std::map<cle, valeur>
std::multimap<cle, valeur>
```


Map



paire

pair< , >

clé / valeur → first / second

```
using mss = std::map<string, string>;
mss m;

m.insert(
   std::pair<string,string>("secours","42"));
m.insert(std::make_pair("loic"  , "405042"));
m["standard"]=  "405000";
```

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

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

 map<string, string> liste;

 // insertions d'éléments

 map<string, string>::const_iterator it
      = liste.begin();
 // auto it = liste.begin();


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

 return 0;
}
```



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


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


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

 return 0;
}
```



Conteneurs associatifs non triés

2011

- Eléments non triés
  - Rangement par "paniers"
  - Tables de hachage
  - Clés/éléments non modifiables

Opérations en O(1)

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

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

Problèmes ➔ foncteurs

- Appliquer un algorithme sur conteneur ?

Ajouter une méthode au conteneur

Pollution de l'API Conteneur
Toujours possible ?

Algorithme dans des classes dédiées

- Différentes versions de l'algorithme ?

Utiliser la surcharge / autre méthode…

Bonne version ?

Héritage

Foncteurs

- Classe qui implémente un algorithme sur le conteneur
  - Surcharge de l'opérateur ()

```
retour F::operator() (paramètres)
```

```
retour F::operator() ( ∅ )
```

  - Avec ou sans état interne
- Une sous-classe par variante
- Accès aux éléments du conteneur par les itérateurs

Une fonction dans un objet

Comparaison d'objets

- Opérateur <

```
bool operator<() (const A&, const A&);
```

```
bool A::operator<() (const A&) const;
```

- Fonction

```
bool comparaison(const A&, const A&);
```

- Foncteur
  - Prédéfini (*template*)
  - Perso

```
std::less<A>
```


Comparateur

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

Pas d'état interne

Les 2 objets à comparer

std::less<A>

cmp.operator()(a1,a2)

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

Comme une fonction !

Générateur de nombres pairs

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

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

std::cout << gen();

Quelques algorithmes…

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

Initialisation / copie

```
transform()
sort()
shuffle()
```

Modification

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

Recherche

Classe de traits (1)

- Classes *templates*
  - Propriétés/caractéristiques associées à un type
  - Types, constantes, méthodes statiques
  - "Au lieu de paramètres" (Myers)
- Spécialisation de *template* + paramètres par défaut
- Différents types
  - Caractères
  - Numériques
  - Itérateurs

Classe de traits (2)

- Algorithme de recherche de MAX dans un tableau
  - tab[0]
  - INT_MIN, FLT_MIN, DBL_MIN dans <climits>
  - numeric_limits

```
template<typename T>
T findMax(const T* data, int numItems) {
  T 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 {
  using type = T;
};
```

Exemple, pas dans la structure

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

Spécialisation de *templates*

Pointeurs intelligents



- unique_ptr
  - Un propriétaire unique
  - Transmission (déplacement) mais pas copie
  - Rendu hors de portée
- shared_ptr
  - Plusieurs pointeurs sur un objet
  - Libération quand plus utilisé
  - Compteur de références
- weak_ptr
  - Propriété temporaire
  - A caster en shared_ptr pour l'utiliser
  - Eviter les cycles

auto_ptr

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


unique_ptr



- Un seul propriétaire d'un pointeur

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

Encore mieux : renvoyer 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



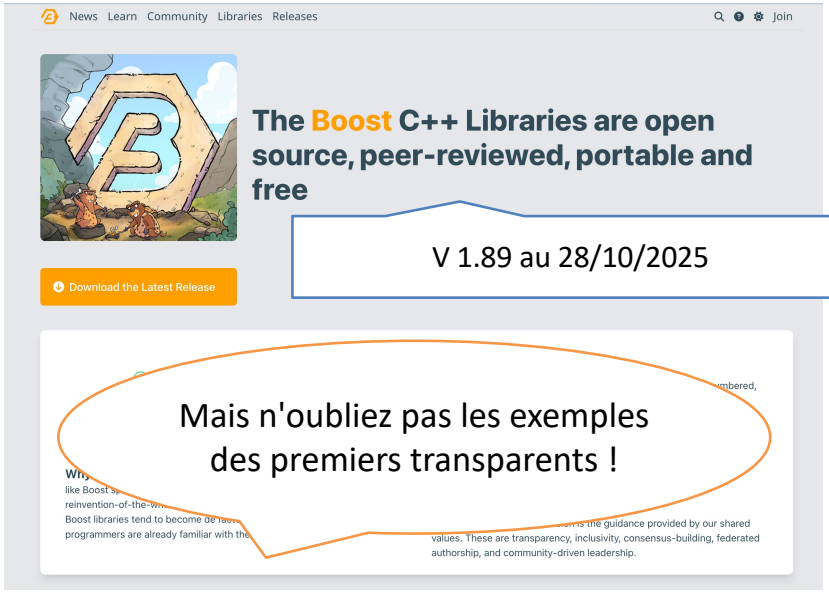
- Ensemble de bits : 0 ou 1, true ou false
  - Optimisé
  - Conversion vers/d'entier ou de chaine
  - Taille fixée à la compilation // vector<bool>

```
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…





Compléments



- Héritage
  - multiple, à répétition, virtuel
  - privé et protégé
  - précisions sur la virtualité
- Divers
  - syntaxes *templates*
  - Opérateur ++
  - méthodes inline
  - Namespace

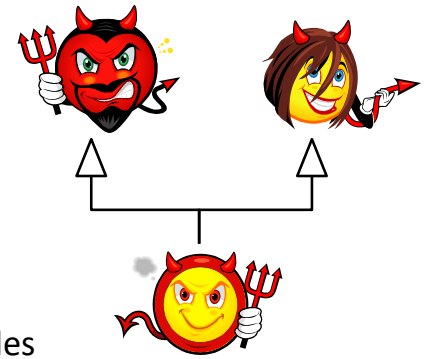
Héritage : rappel

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

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

```
class D : public A {
  protected:
    double attrib;
};
```


Héritage multiple



- Naturel
- Tous les attributs et méthodes des classes mères
  - Collision de nom = lever l'ambiguïté !

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

};
```

Collision de nom

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

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

B::attrib

C::attrib

```
class D : public B, public C {

  void m() {
    cout << attrib;
  }
};
```


Using

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

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

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

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


Héritage à répétition

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

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

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

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


Héritage virtuel (1)



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

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

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

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

Héritage virtuel (2)

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

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


Héritage privé / protégé (1)

- Testament = qui connaît l'héritage ?
- public
  - Toutes les classes qui connaissent la mère
- protected
  - Les descendants seulement
- privé
  - La classe seule
  - Amis éventuels

Héritage privé / protégé (2)

	Héritage		
Membre	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	✖	✖	✖
Mère	Fille		

L'héritage en `private` ?

"est implémenté sous forme de"

Enfin pas tout à fait …

Exemple par B.S.

```
template<typename T> class vector {
   // code complet
};
```

```
template<> class vector<void *> {
   // code complet
};
```

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



Exemple classique …

```
class Moteur {
  public:
    void demarrer() {}
};

class Voiture : private Moteur {
  public:
    using Moteur::demarrer;
};
```

```
Voiture v;
v.demarrer();
```


… mais pas satisfaisant

```
class Moteur {
  public:
    void demarrer() {}
};

class Voiture {
  public:
    void demarrer()
       { moteur.demarrer(); }
  private:
    Moteur moteur;
};
```


Spoil ZZ3

Principe de substitution de Liskov

On doit pouvoirs substituer une classe mère par ses classes filles sans avoir à faire d'autres modifications

Sémantique de l'héritage

Héritage contre duplication de code seulement

Programmation par contrat

Composition
Agrégation

Et l'héritage privé dans tout çà ?

```
class Mere {
 public:
  Mere() {
    std::cout << "Mere::Mere()" << std::endl;
  }
 ~Mere() {
    std::cout << "Mere::~Mere()" << std::endl;
  }
};
class Fille : public Mere
 public:
  Fille() {
    std::cout << "Fille::Fille()" << std::endl;
  }
 ~Fille() {
    std::cout << "Fille::~Fill
  }
};
```

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

Destructeur virtuel pur ?

- Rendre une classe abstraite
- Ne pas trouver de méthode abstraite sauf …

```
class Mere {
 public:
  Mere() {
    std::cout << "Mere::Mere()" << "\n";
  }
  virtual ~Mere() = 0;
};
```

Perturbe l'héritage !

Ajouter une définition par défaut pour les classes filles

Empêcher l'héritage ?

- Ne pas utiliser de destructeur virtuel
  - Les conteneurs de la bibliothèque standard
- Bien lire la documentation des classes utilisées

- Mot-clé `final`
  - Classe
  - Méthode

2011

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

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


Constructeur et polymorphisme

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

```
class Fille : public Mere {
 public:
  Fille() { m(); }
  void m() override
   { std::cout << "mf" << std::endl; }
};
```

```
Fille f;
```

>

>

Constructeur "virtuel" (1)

```
M * f = new Fille;
```

Créer un objet du même type que f ?

Mécanisme de detection de type à l'execution … OU

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


Constructeur "virtuel" (2)

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

```
Mere * m1 = new Fille();
Mere * m2 = m1->create();
```

```
class Fille : public Mere {
 public:
  Fille() {}
 ~Fille() {}
  Fille * create() const;
  Fille * clone()  const;
};
```

new Fille()

new Fille(*this)

Références

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




```
int main(int, char **) {
  MemePasPeur * p = new MemePasPeur;
  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;
}
```


Méthode *inline*

- Dommage d'utiliser une fonction/méthode
  - Pour un traitement simple
  - L'accès à un attribut

INTERDICTION de mettre un attribut en public

- Solution : méthode *inline*
  - Développée comme une macro

-O2

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

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

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

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


Inline mais quand ?

- Suggestion
- Le compilateur fait ce qu'il veut …



- Certains mécanismes empêchent l'*inlining*

Rappel : redéfinition

Exemple.hpp

```
#ifndef __EXEMPLE
#define __EXEMPLE

int VAR_GLOBALE;



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

double maximum(double, double );
#endif
```



```
class Etudiant {
 public:
  std::vector<Cours *>& getCours();
  void echange(std::vector<Cours *>&);
 private:
  std::vector<Cours *> liste;
};
```

```
void afficher(Etudiant& e) {
  std::vector<Cours *>::const_iterator it;


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

Classe ou type « interne »

```
class Etudiant {
 public:
  using cours_t = std::vector<Cours *>;
  cours_t& getCours();
  void     echange(cours_t&);
 private:
  cours_t liste;
};
```

Hors de la classe

```
cours_t& Etudiant :: getCours() {}
```

Dans la classe

```
void Etudiant::echange(cours_t&) {}
```


Usage externe à la classe

```
cours_t& Etudiant::getCours() {}
```

```
void afficher(Etudiant& e) {
  Etudiant::cours_t::const_iterator it;

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


Classe interne et *template*

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

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

}
```

Indication pour le compilateur :

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


Opérateurs++

```
++a;
```

Opérateur préfixé
Incrémente a **avant** utilisation

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

```
a++;
```

Opérateur postfixé
Incrémente a **après** utilisation

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

Copie

a++++;

Paramètre "bidon"
pour différentiation

Namespace

- Eviter les conflits de nom
  - Portée (*scope*) nommée

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

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


Persistance ?

- Capacité d'un objet à survivre à la fin du programme
- Sauvegarde / Restauration
- Fichier / Base de donnée

Conclusion

- On a vu beaucoup de choses
  - Objet et modélisation
  - C++ de base et intermédiaire
  - Mais pas tout !
- Vous ne comprenez pas ?
  - C++ 2017, voire 2020/2023/ 2026



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