Programmation orientée objets : C++

Eric Ramat
eric.ramat@univ-littoral.fr

Laboratoire d'Informatique, Signal et Images de la Côte d'Opale
Université du Littoral - Côte d'Opale

Plan

Déroulement du module

Découpage

- 2 x 6h de cours
- 7 + 3 séances de 3h de travaux pratiques (les concepts)
- 4 séances de 3h de travaux pratiques (SAé "Conception et développement d'une application distribuée") + 2 séances de travaux pratiques en Réseaux

Notation

- **chaque séance** de TP :
  - un compte rendu en fin de séance (sous forme de commits sur gitlab)
  - rendu individuel
  - une note
- notation :
  - premier semestre : 50 % examen + 50 % moyenne des notes TP
  - deuxième semestre : 50 % examen + 25 % moyenne des notes TP + 25 % note de SAé

Déroulement du module

Thèmes abordés

- concepts orientés objets
- bases du langage C++
- gestion mémoire et cycle de vie
- classes et objets
- bibliothèque standard : STL
- fonctions lambdas
- exceptions, flux et généricité

Environnements de travail

- **jetbrains** / **clion**
- vscode
- emacs / vi

Plan

LISP

- dans les années 50-60 au MIT
- notion d'objet : items avec des attributs

Simula 62

- langage pour faire de la simulation
- notion d'objet, de classe, d'héritage et "dynamic binding"

Smalltalk

- né en 1971 au Xerox Palo Alto Research Center
- inventé par Alan Kay
- notion de classe, de meta-classe, "tout objet", ..., de machine virtuelle et interface graphique

Aujourd'hui

- Python : première release en 1991 par Guido van Rossum
- Java : 1991 - sortie officielle en 1996 par Sun Microsystems
- C# : standardisé ISO en 2003 et développé par Microsoft
- PHP : première release en 1995 et support de l'orienté objet en 2004
- JavaScript / EMACScript (ES) : publié en 1995 par Sun Microsystems et Netscape et réellement objet avec ES6 en 2015

Et C++ ?

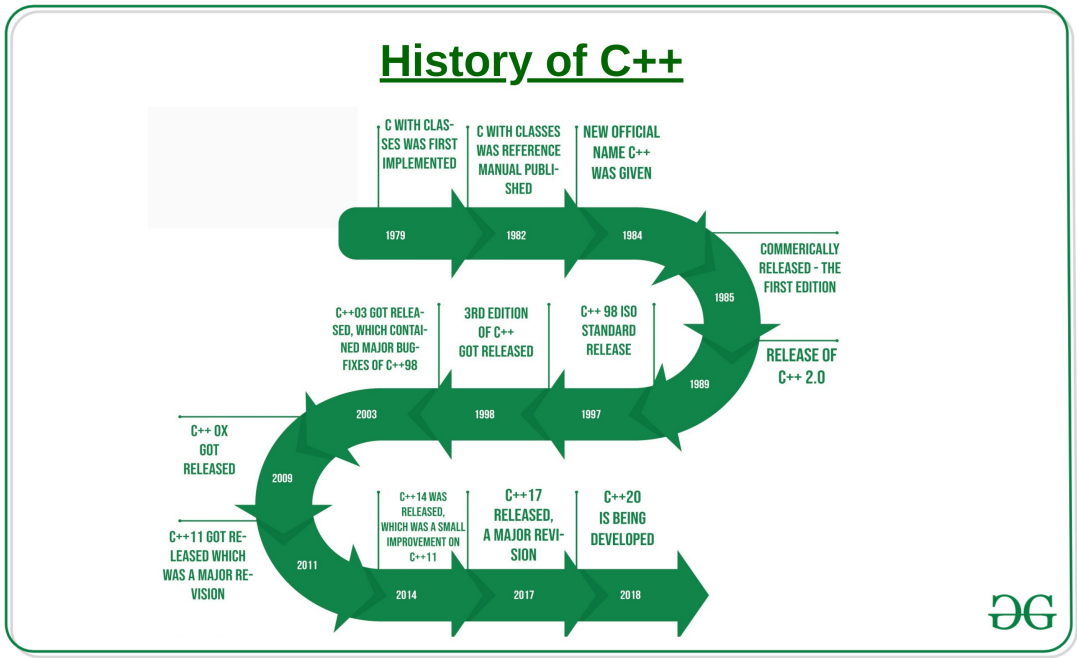
Entre Smalltalk et Python, en 1982 !

- 1982 : création du langage C++ par **Bjarne Stroustrup** (AT&T Bell Laboratories) ; objectif : greffer sur le langage C les concepts de la programmation orientée objets
- 1984 : “C with classes” devient le C++ ; ajout des notions de méthodes virtuelles, de surcharge des opérateurs, de références et de flux ;
- 1985 : première version commerciale
- 1991 : ajout de la programmation générique (templates) et des exceptions ;
- 1998 : standardisation ISO C++ ;

Introduction

Historique du C++

- 2011 : ISO C++11 (move, auto, range-for, variadic, lambda, ...) ;
- 2014 : ISO C++14 (améliorations de la STL, lambda générique, template “variadic”, ...)
- 2017 : ISO C++17 (“fold expression”, algorithmes parallèles, bibliothèque pour les systèmes de fichiers, ...)
- 2020-xxxx : ISO C++20 (module, concept, ...)



Définition (OMG - *Objet Management Group*)

L'objet, c'est :

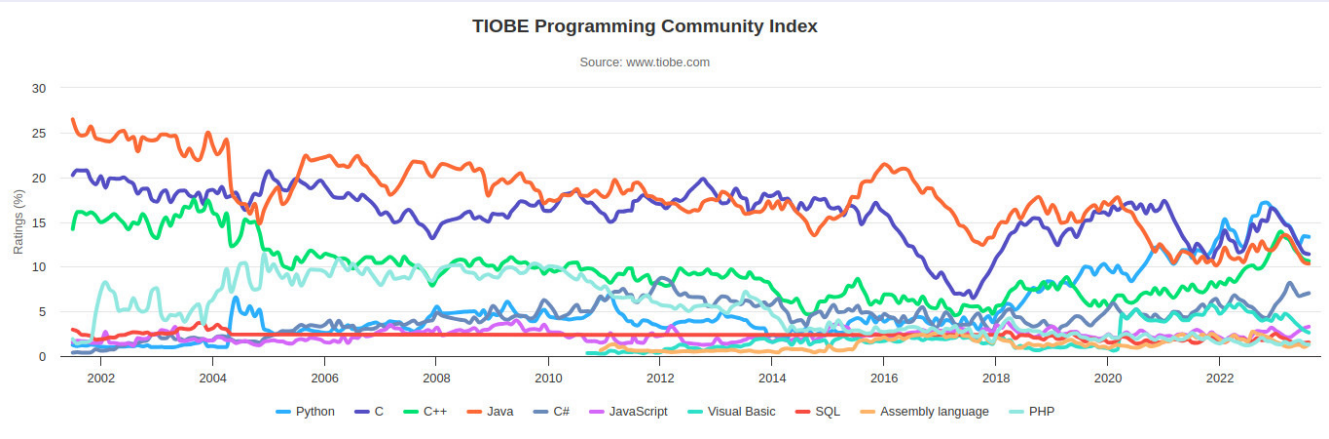
- encapsulation (données + fonctions) ;
- abstraction ;
- héritage ;
- polymorphisme.

En C++

- tout peut être classe ;
- héritage simple et multiple pour les classes ;
- les objets se manipulent via des instances, des pointeurs et des références ;
- une API objet standard est fournie : la STL (Standard Template Library) ;
- la syntaxe englobe celle de C (tout ce que l'on fait en C, on peut le faire en C++).

Indice TIOBE (*The Importance Of Being Earnest*)

- Mesure de la popularité des langages de programmation
- Basé sur le nombre de pages web retournées par les principaux moteurs de recherche via le tag du nom du langage



Aug 2023	Aug 2022	Change	Programming Language	Ratings	Change
1	1		Python	13.33%	-2.30%
2	2		C	11.41%	-3.35%
3	4	^	C++	10.63%	+0.49%
4	3	v	Java	10.33%	-2.14%
5	5		C#	7.04%	+1.64%

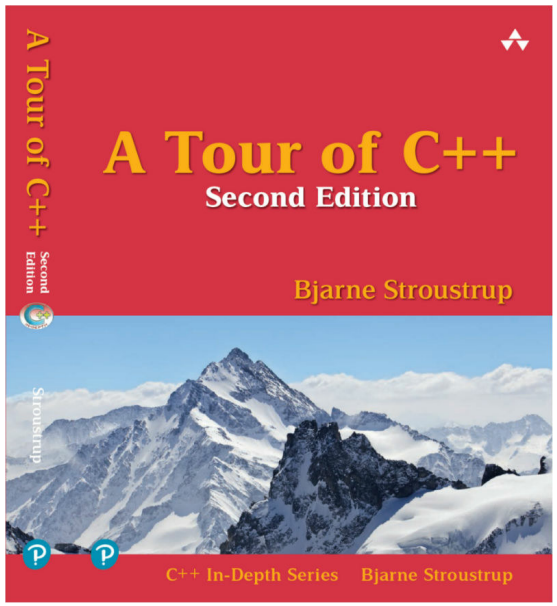
Domaines

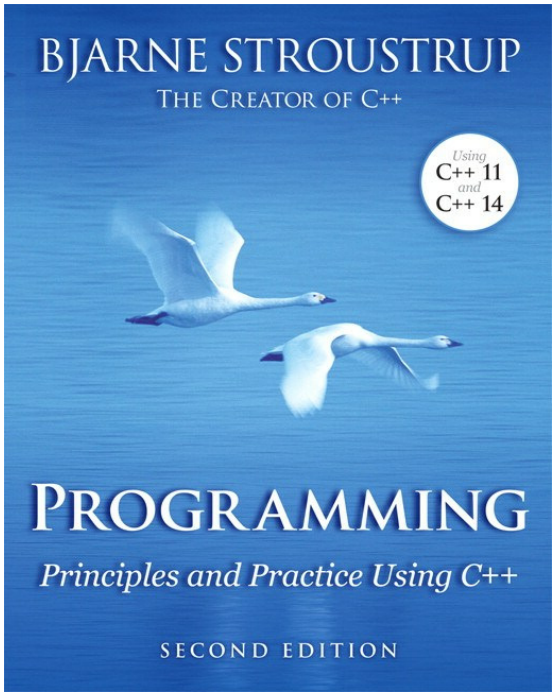
- Système d'exploitation
- Développement de jeux vidéos (Unity, Löve, ...) et réalité virtuelle ou augmentée (Unreal)
- IoT (*Internet of Things*)
- Moteur de base de données (MySQL, MongoDB, ...)
- Navigateur web (Chrome, Firefox, Safari et Opera)
- Bibliothèques de *machine learning* (TensorFlow, ...)
- Recherche scientifique, outils d'analyses financières, télécommunications
- ...

Introduction

Bibliographie

- Bjarne Stroustrup. *A tour of C++*. 2ème édition. Addison-Wesley, 2018 ;
- Bjarne Stroustrup. *Programming - Principles and Practice Using C++*. 2ème édition. Addison-Wesley, 2014 ;
- Bjarne Stroustrup. *Le langage C++*. 4ème édition. Addison-Wesley, 2003 ;





Sites web sur le langage

- le site de Bjarne Stroustrup : `https://www.stroustrup.com/`
- `https://en.cppreference.com/w/`
- `https://isocpp.org/`
- `https://www.cplusplus.com/`

Aide

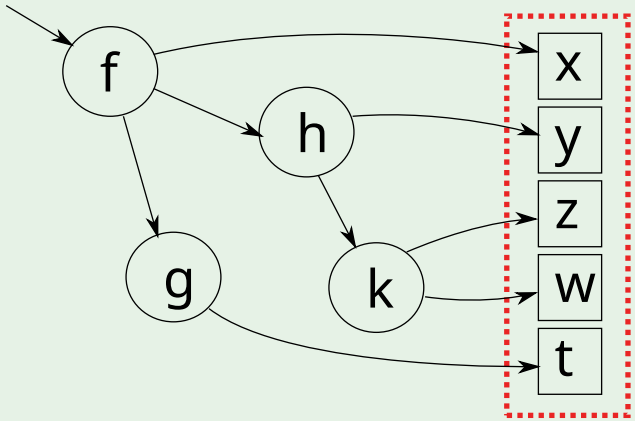
`https://stackoverflow.com/`

Plan

Les concepts

Programmation procédurale

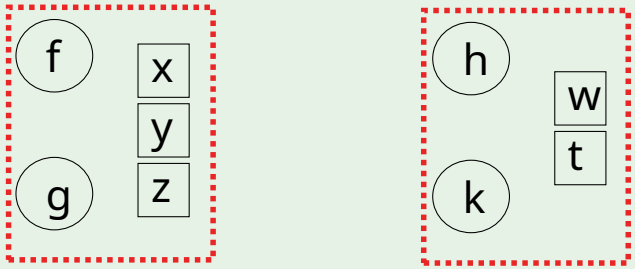
- un programme = un ensemble de **fonctions**, de types et de **variables**
- une fonction prends en entrée des variables (paramètres), réalise un traitement et retourne des valeurs
- un type est une définition de la nature des valeurs possibles d'une donnée (définition de la liste des valeurs possibles)
- une variable stocke des données de type simples ou complexes (tableau, structure, ...)



Les concepts

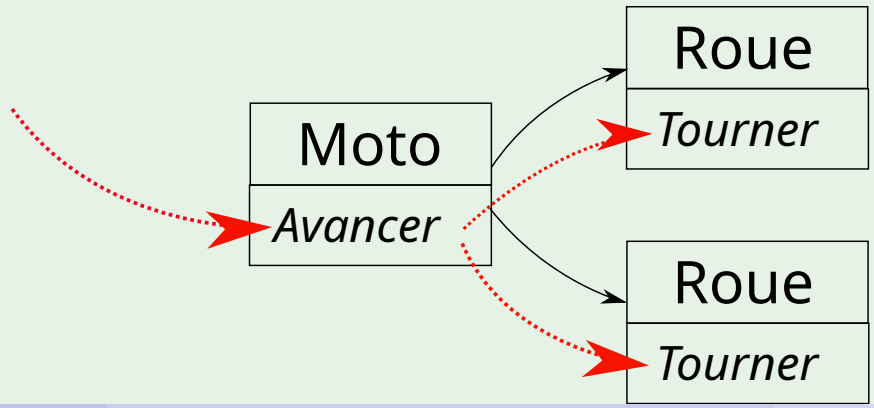
Programmation orientée objets

- un programme = un ensemble de **classes** et d' **objets**
- une classe est un type regroupant des données (ou propriétés) et des fonctions (ou méthodes)
- un objet est une instance d'une classe (ou une variable issue d'une classe)



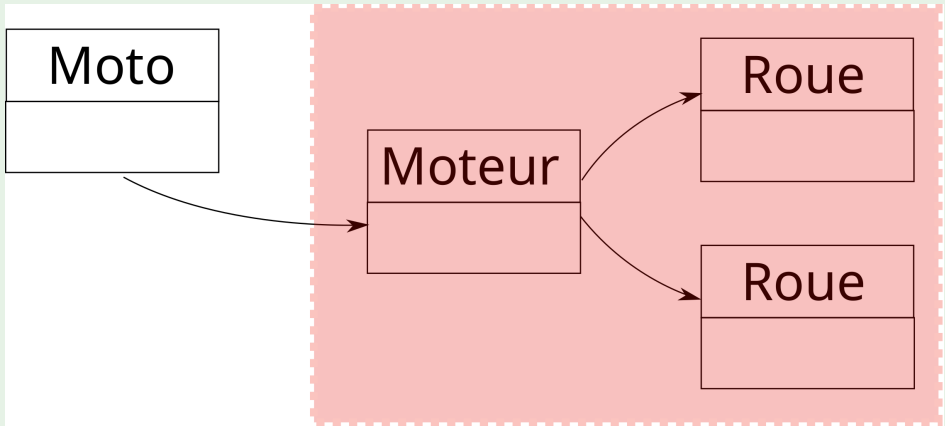
Relations entre objets

- l'objet protège ses données (intégrité) - il est le seul à pouvoir les manipuler
- les fonctions (ou méthodes) s'appliquent sur les données (ou propriétés) de l'objet
- un objet peut être "lié" à d'autres objets
- un objet peut demander aux objets qu'il connaît de faire quelque chose (invocation d'une méthode)



Objectifs

- cacher la complexité ou simplifier l'utilisation
- limiter l'impact des modifications
- factoriser des propriétés au sein de classe plus abstraites



Objectifs

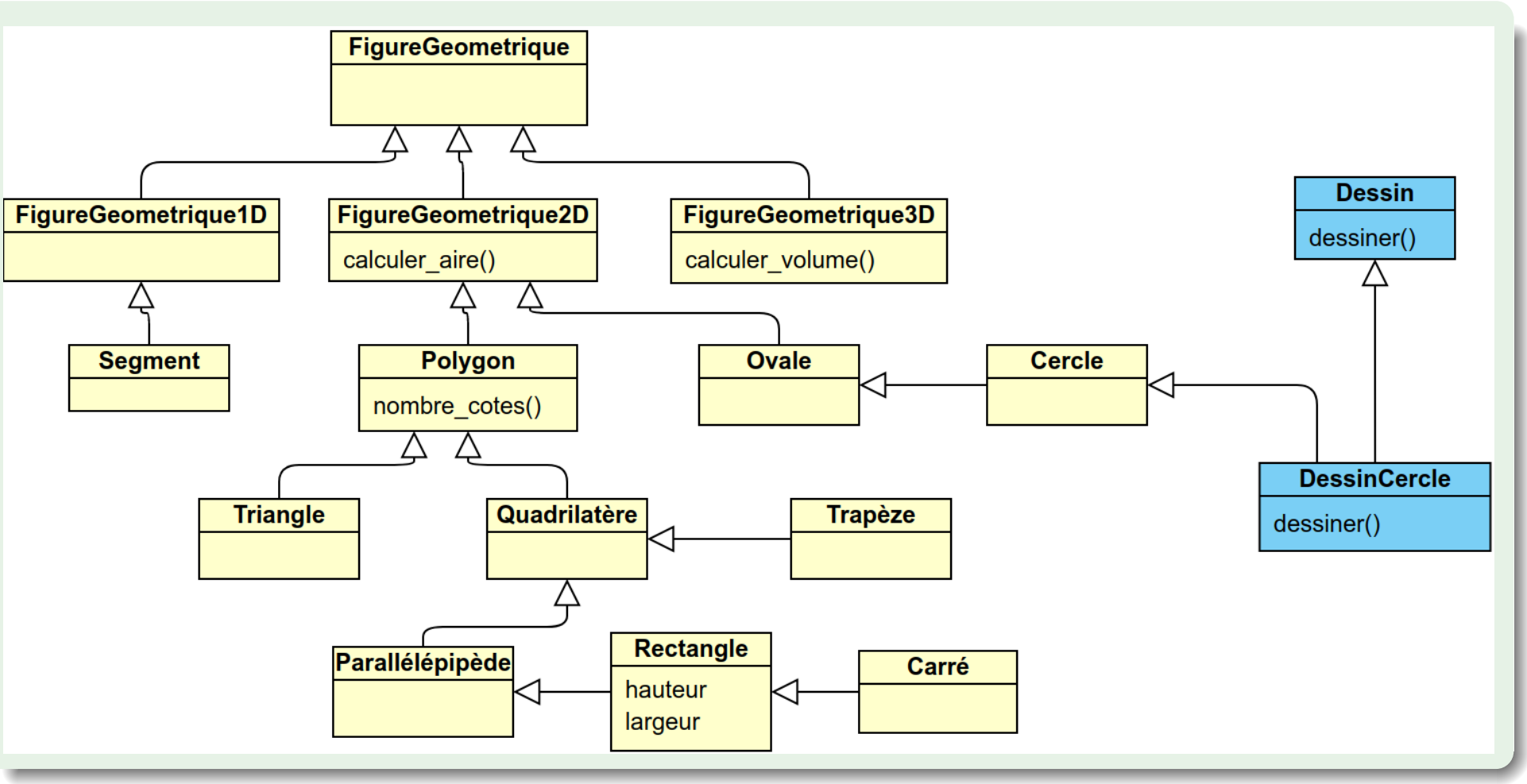
- généraliser ou spécialiser un concept (un type)
- relation entre deux classes : une sous-classe (classe fille) hérite des propriétés (données ou fonctions) d'une ou plusieurs classes mères
- cela forme une hiérachie de classes

Typage

- si une classe B hérite de A, alors "B est une sorte de A" ou "B est un A"
- en généralisant, toutes sous-classes de A est aussi de type A

Les concepts

Héritage

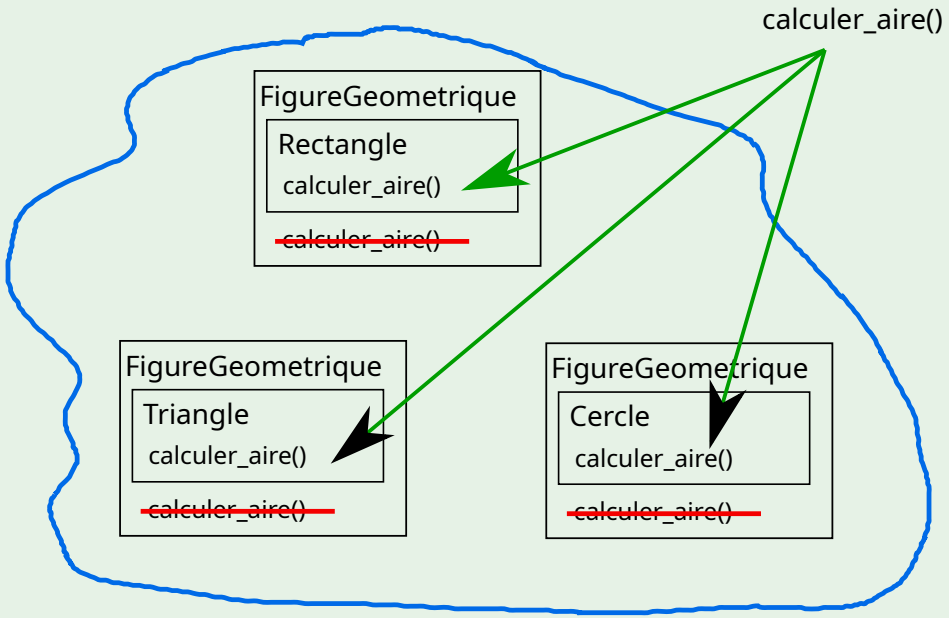


Objectifs

- définir des interfaces uniques (par exemple, un ensemble de méthodes) pour des types différents
- polymorphisme ad hoc : une même fonction qui possède le même nom mais pas les mêmes types de paramètres
- polymorphisme paramétrique : une classe ou une fonction dépendant d'un type générique
- polymorphisme de type sous-type : une classe fille va **redéfinir** une fonction définie dans la classe mère → attention à ne pas modifier la sémantique

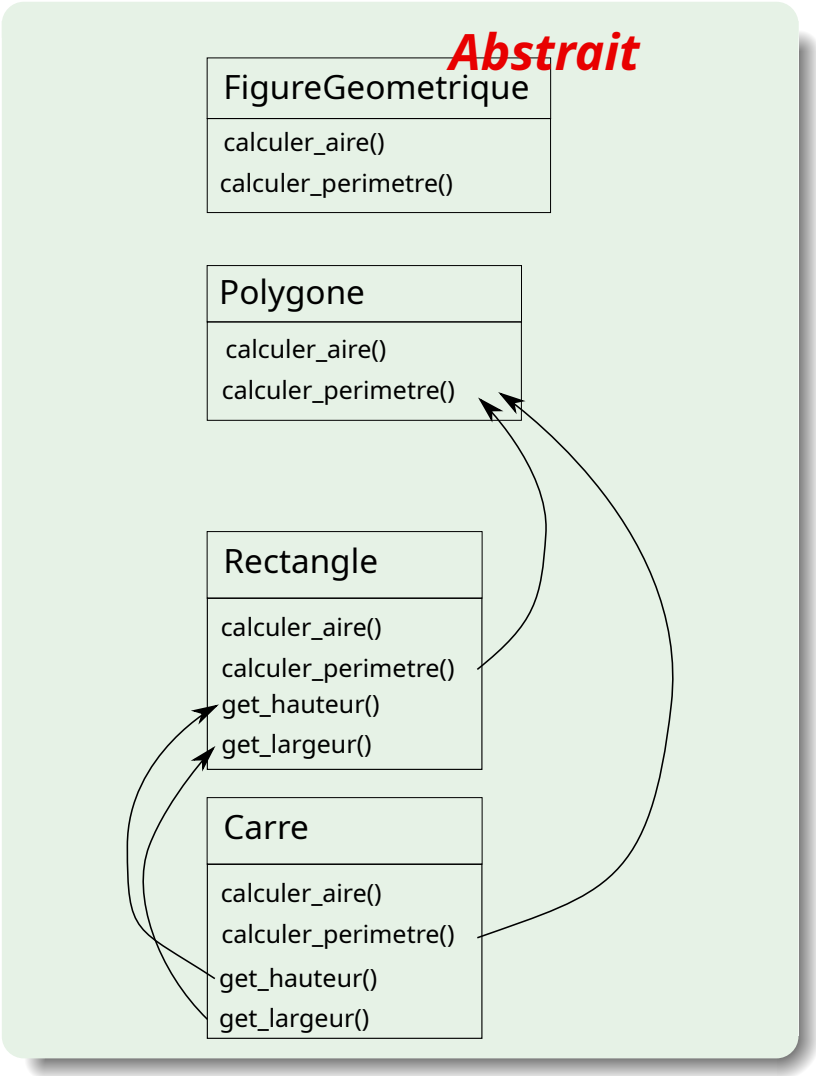
Principe

- possibilité de mettre dans un ensemble (ou collection) des objets de même type (par exemple, des objets de type FigureGeometrique)
- les objets peuvent être issus des sous-classes de FigureGeometrique
- **la fonction “calculer_aire()” dépendra du type réel de l’objet**



Mécanisme

- chaque classe possède une table référençant le code des fonctions
- si surcharge alors le code référencé est celui la classe
- sinon le code référencé est celui de la dernière classe ayant implémentée la fonction (en remontant la hiérarchie)



D'autres concepts

Responsabilité simple

- l'objet est responsable des traitements qu'il propose
- ces traitements doivent se limiter à l'objectif de l'objet (**pas faire plus que ce qu'il doit faire**)

Cohésion

- un objet doit former un tout cohérent
- conséquence : de petits objets, c'est mieux qu'un gros !

Couplage

- un objet peut communiquer avec les objets qu'il connait
- il faut minimiser et simplifier ces échanges
- sinon on parle de couplage fort → augmentation des problèmes lié aux modifications

Plan

Éléments du langage C++

Commentaires

Les commentaires, deux syntaxes possibles :

- /* et */ : tout ce qui se trouve entre les deux est commenté
- // : tout ce qui suit sur la même ligne est un commentaire

Exemple

```
/* ceci est un commentaire sur plusieurs lignes. ceci est un
   commentaire sur plusieurs lignes. ceci est un commentaire sur
   plusieurs lignes. ceci est un commentaire sur plusieurs
   lignes. */

// ceci est un commentaire sur une ligne
int x; // ceci est un commentaire
       // ceci est un commentaire
```

Grammaire générale :

- Une expression se termine par un point-virgule ;
- Un bloc contient des expressions et est contenu entre { et }

Exemple

```
{                       // ceci est un bloc
  int x = 1;

  x = x + 1;
  {                     // un deuxieme bloc
    x = x + 2;          // la variable x est toujours disponible
  }

  x = x + 1;            // x vaut 5

} // en sortie de bloc x est detruit
```

Les variables peuvent être déclarées n'importe où dans un bloc.

Exemple de portée

```
{                       // ceci est un bloc
                        // avec une variable locale x
  int x = 1;

  {                     // un deuxieme bloc
                        // avec une variable locale y
    int y;

  }  // en sortie de bloc, yest detruit

} // en sortie de bloc, x est detruit
```

Attention

La notion de variable globale existe mais est fortement déconseillé.

Initialisation

Il existe plusieurs façons d'initialiser une variable :

- à l'aide d'une affectation
- avec les accolades

```
int x = 1;
int y{1};
```

Tableau et accolade

Les éléments d'un tableau peuvent être aussi initialisés à la déclaration

```
int t[3]{1,2,3};

for (unsigned int i = 0; i < 3; ++i) {
  std::cout << t[i] << std::endl;
}
```

Éléments du langage C++

Les types simples (machine x86 64 bits)

Nom	Type	Taille
`char`	un entier	$[-128, 127]$
`short`	un entier	$[-2^{15}, 2^{15}-1]$
`int`	un entier	$[-2^{31}, 2^{31}-1]$
`long`	un entier	$[-2^{63}, 2^{63}-1]$
`unsigned char`	un entier sans signe	$[0, 255]$
`unsigned short`	un entier sans signe	$[0, 2^{16}-1]$
`unsigned int`	un entier sans signe	$[0, 2^{32}-1]$
`unsigned long`	un entier sans signe	$[0, 2^{64}-1]$
`float`	un réel	$[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$
`double`	un réel	$[-1.7 \times 10^{308}, 1.7 \times 10^{308}]$

Nouveaux types (C++11)

Pour pallier aux différentes représentations des types numériques, de nouveaux types sont apparus en C++11 : int8_t, int16_t, ...

Définition

1 = taille de char $\leq$ taille de short $\leq$ taille de int $\leq$ taille de long
taille de float $\leq$ taille de double

Attention

Une grande attention avec les réels (pour tout langage de programmation) est obligatoire :

```
double x = 0.1 + 0.1 + 0.1;
double y = 0.3;

assert(x == y); // erreur x est different de y !
```

Définition

- Une structure est un type composé définie par le programmeur
- Elle possède un ensemble d'attributs de types différents ou non
- Il est possible d'y ajouter des fonctions dont deux fonctions particulières (le constructeur et le destructeur)

```
struct Circle
{
  double _x_center;
  double _y_center;
  double _radius;

  Circle(double x_center, double y_center, double radius) :
    _x_center(x_center), _y_center(y_center), _radius(radius) { }

  double area() { return 3.14 * _radius * _radius; }
};
```

En C

Les constantes sont définies à l'aide du préprocesseur et de la commande #define.

```
#define PI 3.141529
```

En C++

Les constantes sont définies à l'aide du mot clé **const**.

```
const double PI = 3.141529;
```

Une énumération est un type simple défini par l'utilisateur sous forme d'un ensemble de valeurs :

```
enum MonType { VALEUR1, VALEUR2, VALEUR3, VALEUR4 };

MonType test = VALEURS1;
```

Remarque

- les énumérations sont traduites sous forme d'entiers par le compilateur C++
- la valeur du premier élément de l'ensemble est fixée à 0 (ici, `VALEUR1 == 0`)
- on peut fixer d'autres valeurs en donnant une valeur lors de la déclaration `{ VALEUR1 = 2, ... }`
- dans ce cas, les valeurs suivantes seront les entiers 3, 4 et 5

Une énumération est un type simple défini par l'utilisateur sous forme d'un ensemble de valeurs :

```
enum MonType { VALEUR1, VALEUR2, VALEUR3, VALEUR4 };

MonType test = VALEURS1;
```

Remarque

- les énumérations sont traduites sous forme d'entiers par le compilateur C++
- la valeur du premier élément de l'ensemble est fixée à 0 (ici, `VALEUR1 == 0`)
- on peut fixer d'autres valeurs en donnant une valeur lors de la déclaration `{ VALEUR1 = 2, ... }`
- dans ce cas, les valeurs suivantes seront les entiers 3, 4 et 5

Le test conditionnel :

- `if` : l'instruction de test
- `else` : si le test echoue
- `else` et `else if` : si le test échoue, une nouvelle instruction ou un nouveau test

Exemple

```
if (a < b) {
   ...
}

if (c < d) {
   ...
} else {
   ...
}
```

```
if (e < f) {
   ...
} else if (e < g) {
   ...
} else if (e < h) {
   ...
} else {
   ...
}
```

Les opérateurs de comparaisons :

Opérateur	Type
==	égalité
!=	différent
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal

Les combinaisons de conditions :

Opérateur	Type
and &&	et
or ||	ou
xor ^	ou exclusif
not !	négation

Exemples

```
if (min <= x and x <= max) { ...

if (x < 10 or y < 10) { ...

if ((min <= x and x <= max) or (min <= y and y <= max)) { ...
```

while

```
int i = 0;
while (i < 100) {
  ...
  i++;
}
```

for

```
for (int i = 0; i < 100; i++) {

  ...

}
```

do while

```
int i = 0;
do {
  ...
  i++;
} while (i < 100);
```

Les trois manières de faire des boucles en C++

→ point commun : le test doit être vrai pour continuer les boucles

Tester les valeurs par rapport à un ensemble de constantes (ne fonctionne que pour les types entiers et les énumérations)

Exemple

```
enum MonEnum { INIT, PHASE1, END };

MonEnum e;          // declaration d'une variable basee sur l'enumeration

switch (e) {        // commence les tests
case INIT:          // si e == INIT
  ...
  break;
case PHASE:         // si e == PHASE
  ...
  break;
case END:           // si e == END
  ...
  break;
};
```

Définition

Une fonction est une partie nommée d'un programme que l'on peut appeler d'autres parties du programme aussi souvent que nécessaire.

Exemples

```
// Declaration d'une fonction a deux arguments reels, x et y, et qui
// retourne a la fonction l'appelant la somme de ces deux reels
double somme(double x, double y) {
  return x + y;
}

// Une fonction qui affiche le couple (x, y) et la somme de ces
// deux chiffres
void show(double x, double y) {
  std::cout << "(x:" << x << ", y:" << y << ")";
  std::cout << "somme : " << somme(x, y);
}
```

Définition

Un espace de noms est un mot clé associé à un ensemble de définitions C++ (classes, fonctions, ...) afin d'éliminer les problèmes de définition multiples.

Exemples

```
namespace test {
  void toto()
  {
  }
}

namespace test2 {
  void toto()
  {
  }
}
```

Exemples

```
// Utilisation des namespaces
test::toto();
test2::toto();

using namespace test; // demande au compilateur d'utiliser test
                      // dans l'espace de nom global
toto();
```

Remarque

L'espace de noms de la bibliothèque standard C++ est `std`.

- Les entrée-sorties sont possibles grâce à deux flux :
  - cout : flux de sortie écran
  - cin : flux d'entrée clavier
- Les opérateurs << et >> permettent d'afficher sur la console et de lire au clavier ;
- Les retours chariots sont possibles ;

Exemples

```
#include <iostream>

std::cout << "Hello␣world␣!" << std::endl;

std::cout << "Hello␣world␣!" << "\n";
```

- Toute variable de type primitif est acceptée par ces fonctions (de même que les constantes et les chaînes de caractères constantes) ;

Comme C, la fonction main constitue le point de départ.

Exemples

```
int main(int argc, char** argv) {
  return 0;
}

int main() {
  return 0;
}
```

La compilation sous Linux est réalisée grâce à gcc ou clang++.

Compilation

```
# generation du fichier objet (main.o)
  $ gcc -c main.cpp
# generation de l'executable (main)
  $ gcc -o main main.o
```

Plan

En C, l'allocation dynamique se faisait à l'aide de deux fonctions : malloc et free.

En C

```
// Allocation d'un entier
int *x = (int*)malloc(sizeof(int));

// Allocation d'un tableau de caracteres
char *x = (char*)malloc(sizeof(char)*size);
free(x);
```

En C++, l'allocation dynamique est simplifiée grâce aux operateurs new et delete.

En C++

```
// Allocation d'un entier
int *x = new int;
delete x;

// Allocation d'un tableau de caracteres
char *x = new char[size];
delete[] x;
```

Gestion mémoire

Comment gérer au mieux la mémoire : garbage collector, optimisation de l'allocation et de la libération de petits objets, ...

Smart pointers - Définition

- smart pointers = "pointeur intelligent" ;
- le but principal est de déléguer la gestion mémoire à un pointeur intelligent ;
- création d'une sorte de garbage collector ; ne plus se préoccuper de la libération mémoire ;
- cette bibliothèque est un ensemble de classes `template` (classe paramétrable)

Définition

- le type unique_ptr prends en charge la gestion d'un objet alloué dynamique
- le pointeur est désalloué automatiquement si la variable de type unique_ptr n'existe plus

```
std::unique_ptr<int> x = std::make_unique<int>(10);
// equivalent a : std::unique_ptr<int> x(new int{10});
std::cout << *x << std::endl;
*x = 20;
std::cout << *x << std::endl;
```

- attention, le pointeur doit être géré par un seul unique_ptr
- il est possible de transférer la propriétaire du pointeur à un autre unique_ptr via la méthode release

Définition

- shared_ptr est une classe template qui encapsule un pointeur avec compteur de références ;
- à chaque affectation, construction par copie, ..., le compteur de réferences est incrémenté ;
- l'objet pointé est désalloué lorsque le nombre de références est nul ;
- le pointeur doit référencer un objet alloué dynamiquement (les tableaux ne sont pas pris en compte - utilisé shared_array dans ce cas).

shared_ptr

```
std::shared_ptr<int> x = std::make_shared<int>(10);
// equivalent a : std::shared_ptr<int> x(new int{10});

std::cout << *x << std::endl;
std::cout << x.use_count() << std::endl;

std::shared_ptr<int> y = x;
*y = 20;
std::cout << *x << std::endl;
std::cout << x.use_count() << std::endl;
std::cout << *y << std::endl;
```

- une variable dynamique x de type `int` est créee et initialisée à 10 ;
- le compteur de références est égal à 2 après la création de y ;
- c'est seulement lors de la destruction de x et y que la zone mémoire utilisée est désallouée.

Constructeurs

- un constructeur sans paramètre : le compteur de références est nul et ne référence rien ;

```
std::shared_ptr<int> p;
```

- un constructeur admettant un pointeur de Y (le type Y étant égal à T ou convertible à T) ; T* est le type du pointeur encapsulé

```
std::shared_ptr<int> p1 = new int{10};
std::shared_ptr<int> p2(new int{10});
```

Méthodes

- le constructeur par recopie (CopyConstructive) et l'opérateur d'affectation (Assignable) sont possibles ;
- reset() affecte le pointeur à null et met à zéro le compteur de références ; il existe des variantes de reset avec paramètres, dans ce cas, reset est équivant à swap ;
- swap échange le contenu (pointeur et compteur) des deux shared_ptr ;
- deux méthodes d'indirection (* et ->) afin de simplifier l'utilisation du pointeur caché ;
- trois méthodes sur le compteur de références :
  - operator bool : placer le pointeur dans un if
  - unique : le compteur est-il égal à un ?
  - use_count : valeur du compteur

Définition

- weak_ptr est une classe template et encapsule un pointeur qui référence une variable déjà pointée par des shared_ptr ;
- il est construit à partir d'un pointeur partagé ;
- pourquoi faire ?
- `shared_ptr` = propriétaire vs `weak_ptr` = non propriétaire
- si l'objet est détruit par le shared_ptr qui le référençait alors tout accès à la variable depuis le weak_ptr provoque une erreur de segmentation

Gestion mémoire et cycle de vie

weak_ptr

Afin d'accèder au pointeur contenu dans le `weak_ptr`

- définir localement un pointeur partagé en faisant appel à la méthode lock ;
- la fonction lock vérifie si le pointeur est toujours valide ;
- la fonction lock créé ensuite un pointeur partagé et par conséquent, si un reset est fait par la suite sur p alors r reste valide.

weak_ptr

```
std::shared_ptr<int> p(new int(5));
std::weak_ptr<int> q(p);

// un peu plus tard

if(std::shared_ptr<int> r = q.lock())
{
  // utilisation de *r
}
```

weak_ptr

```
std::weak_ptr<int> y;
{
  std::shared_ptr<int> x = std::make_shared<int>(10);
  y = x;
  std::shared_ptr<int> z_in = y.lock();
  std::cout << *z_in << std::endl;
}

std::shared_ptr<int> z_out = y.lock();
std::cout << *z_out << std::endl;
```

Statique vs dynamique

- Une variable est créee dans un bloc et disparaît à la fin du bloc
- Une variable est crée dynamiquement (allocation mémoire dans le tas - `heap`) ; si le lien n'est pas perdu (fuite mémoire), elle sera détruit par une opération `delete`

Cycle de vie dans un bloc

```
{
  A x; // creation de la variable x de type A

  ... // utilisation de la variable

} // libÃ©ration de la mÃ©moire occupÃ©e par la variable
```

Paramètre

- Un paramètre de type simple ou composé (`struct`) sera copié lors de l'appel (passage par valeur)
- La variable sera détruite à la fin de la fonction
- Si c'est un pointeur (ou une référence), pas de destruction

Cycle de vie d'un paramètre

```
int f(A x) // creation d'une copie de la variable
{
   ... // utilisation en lecture ou Ã©criture du paramÃštre

} // libÃ©ration de la mÃ©moire occupÃ©e par le paramÃštre

A y; // creation d'une variable de type A

f(y);
```

Gestion mémoire et cycle de vie

Référence

- Un moyen de se débarrasser des pointeurs, c'est la notion de référence ;
- Pour manipuler une variable déjà créée, on déclare une référence sur le type de la variable et on affecte la variable pour une variable du même type

Référence

```
Circle a;
Circle& c = a;
```

La variable c référence une instance de la structure Circle.

Gestion mémoire et cycle de vie

Référence

- Contrairement à un pointeur, l'accès aux méthodes et attributs se font simplement.

Référence

```
double d = c.getArea();
```

- Une référence doit **toujours** faire référence à une variable initialisée ;
- La notion de référence nulle n'existe pas ;
- Une référence sur une variable ne peut pas être modifié après une première affectation ;

Cycle de vie d'un paramètre - référence

```
int f(A& x) // creation d'une reference sur la variable
            // passe en paramÃštre
{
   ... // utlisation de la variable

}

A y; // creation de la variable

f(y);
```

Référence

La variable passée en paramètre peut être modifié dans la fonction via la référence

Par valeur

- Lorsqu'une struct est passée en paramètre, il y a **création par recopie** d'une nouvelle instance de la struct;
- Toute modification de cette dernière n'a aucun effet sur l'instance d'origine.

Par référence

- Au lieu de réaliser une copie de l'instance, seule une référence est passée ;
- En revanche, toute modification réalisée sur le paramètre à l'intérieur de la fonction affecte l'instance d'origine ;
- Avantage : pas de construction de nouvelle instance.

Retour de fonction

- Une fonction peut retourner une variable locale
- La variable “persiste” le temps d’être copié (ou transféré) dans la variable recevant le résultat

Cycle de vie d’une variable en retour d’une fonction

```
A f()
{
  A x;
   ... // utilisation de la variable
  return x;
}

A y = f(); // copie de la variable puis destruction de x
```

Par référence

Si une fonction retourne une référence alors la variable référencée et retournée peut être modifiée.

```
A& inc(const A& a) {
   a.x++;
   return a;
}

void g() {
   A a(10);

   inc(inc(a));
}
```

Par valeur

Retourner une instance implique la création temporaire d'instance.

```
struct A {
   int a;

   A(int a) { this->a = a; }
   A(const A& x) { a = x.a; }
   A inc() { A o(*this); o.a++; return o; }
};

void f() {
   A b(3);

   b = b.inc().inc();
}
```

La structure retournée par le premier appel à inc persiste.

Exemples de création selon le constructeur invoqué.

Constructeur par défaut

```
// une instance
A x;
// un pointeur sur une instance allouee dynamiquement
A* x = new A;
// un pointeur sur un tableau alloue dynamiquement d'instances
A* t = new A[10];
```

Le constructeur par défaut est important car c'est le seul qui permet l'allocation dynamique d'un tableau d'instances.

Constructeur à plusieurs paramètres

```
// une instance
A x(3);
// un pointeur sur une instance allouee dynamiquement
A* x = new A(3);
```

Struct sans constructeur

Le struct A possède un attribut.

```
A x{3};
```

Il est possible d'initialiser un tableau de structs en invoquant un autre constructeur que le constructeur par défaut.

Tableau d'objets

```
A t[3] = {A(1) , A(1,2) , A()};
```

Constructeur par recopie

```
// une instance
A y(x);
A y = x;
// un pointeur sur une instance allouÃ©e dynamiquement
A* y = new A(x);
```

Gestion mémoire et cycle de vie

Destruction

- La destruction des variables est prise en charge par le programmeur ;
- Pas de garbage collector sans utilisation de pointeur intelligent ;
- Toute variable allouée dynamiquement (utilisation de l'opérateur **new**) doit être désalloué lorsqu'il n'est plus utilisé ;
- Le mot-clé **delete** est utilisé par la libération de la mémoire utilisée par une instance allouée dynamiquement.
- Lorsqu'une variable de type struct est détruite par le compilateur (en cas de création d'instance) ou par l'utilisateur (en cas d'allocation dynamique), une méthode particulière est appelée : le `destructeur` ;

Gestion mémoire et cycle de vie

Destruction

- L'écriture d'un destructeur n'est pas obligatoire ;
- En revanche, il doit être impérativement développé si la structure possède des attributs alloués dynamiquement lors de la création d'une instance ;
- Le destructeur est une méthode dont le nom est celui de la structure précédé du caractère ~, sans type de retour et sans argument.

Exemple

```
struct A {
  A();   // Constructeur par defaut
  ~A();  // Destructeur
};
```

Plan

Définition

Une classe est une entité ou structure informatique dont le but est de simplifier, améliorer et de mieux structurer les développements informatiques.

En C++, une classe est définie par :

- le mot clé `class` (Ex. : `class Position`)
- d'une ou plusieurs fonctions constructeurs (Ex. : `Position()`)
- d'un destructeur (Ex. : `~Position()`)
- d'attributs de types simples, composés, de classes etc.
- de méthodes pour manipuler les attributs

Struct

- Une structure en C++ peut respeter les caractéristiques d'une classe (C++98)
- Une structure est un type regroupant un ensemble d'informations (attributs)
- La différence est que tout est public
- On peut aussi définir des constructeurs

Struct avec méthode

```
struct Test {
  int a;
  int b;

  int sum() { return a + b; }
};
```

Exemple d'une classe minimaliste

```
class Test { };
```

Pour cette classe vide, le langage C++ fournit des fonctions par défaut :

Classe générée par le C++

```
class Test {
  Test() { }                          // constructeur par defaut
  Test(const Test& test) { }          // constructeur par recopie
  Test(const Test&& test) { }         // constructeur de deplacement (C++11)
  ~Test() { }                         // destructeur

  Test& operator=(const Test& test) // operateur d'affectation
  { return *this; }
  Test& operator=(const Test&& test) // operateur d'affectation
                                     //  par deplacement (C++11)
  { return *this; }
};
```

Classe position

```
class Position {
  double x, y;                          // Definition des attributs
  int    numero;

  Position() :                          // constructeur par defaut
    x(0.0), y(0.0), numero(0)
  { }

  Position(double x, double y) :        // constructeur supplementaire
    x(x), y(y), numero(0)
  { }

  Position(const Position& pos) :       // constructeur par recopie,
    x(pos.x), y(pos.y), numero(pos.numero) // fourni par defaut
  { }
};
```

Classes

Paramètre par défaut

- Lors de l'appel, un paramètre de fonction se voit affecté d'une valeur (`Position(double x = 0, double y = 0)`);
- En C++, il est possible de définir une valeur par défaut au paramètre de type primitif.

Règles

- Lors de la déclaration d'une fonction, tout paramètre avec une valeur par défaut doit être suivi par des paramètres ayant tous une valeur par défaut ;
- Lors de l'appel, si le ième paramètre est omis alors il doit posséder une valeur par défaut et tous les paramètres suivants doivent être omis et possèdent une valeur par défaut.
- Les valeurs par défaut ne sont spécifiées que dans la déclaration des classes.

Une classe pour être utilisable, doit être instanciée (c-à-d, créée) :

Instantiation d'une classe

```
void creer_position() {
  Position p1;          // Creation d'un objet Position avec l'appel au
                        // constructeur par defaut (x = 0.0, y = 0.0)

  Position p2(1., 2.); // Creation d'un objet Position avec l'appel au
                        // constructeur a deux reels (x = 1.0, y = 2.0)

  Position p3(p1);      // Creation d'un objet Position avec l'appel du
                        // constructeur par recopie
}
```

Remarque

Les objets ci-dessus seront détruits à la sortie de la fonction, comme tout autre objet, type simple ou composé alloué dans un bloc.

Classes

Exemple complet

Une classe pour être utilisable, doit être instanciée (c-à-d, créée) :

Instantiation d'une classe

```
void creer_position() {
  Position p1;          // Creation d'un objet Position avec l'appel au
                        // constructeur par defaut (x = 0.0, y = 0.0)

  Position p2(1., 2.); // Creation d'un objet Position avec l'appel au
                        // constructeur a deux reels (x = 1.0, y = 2.0)

  Position p3(p1);      // Creation d'un objet Position avec l'appel du
                        // constructeur par recopie
}
```

Remarque

Les objets ci-dessus seront détruits à la sortie de la fonction, comme tout autre objet, type simple ou composé alloué dans un bloc.

Classes

Définition/déclaration

- En C++, la définition et la déclaration des classes peuvent être séparées (fortement conseillé pour optimiser la compilation);
- La définition est placée dans des fichiers d'entête (h, hpp ou hxx) ;
- La déclaration est placée dans des fichiers de type cpp ou cxx ;

Déclaration

```
#include <A.hpp>

A::A() { // le constructeur par defaut de A
  ...
}

void A::f() { // une methode de A
  ...
}
```

Double-inclusion

Une même classe peut être utilisée par plusieurs fichiers, il faut se protèger contre la multiple définition.

Exemple - Fichier A.hpp

```
#ifndef A_HPP
#define A_HPP

class A { ... };

#endif
```

pragma

- Utilisation du “pragma once”
- Lié au proprécesseur et non au langage

```
#pragma one

class A { ... };

#endif
```

Classes

Visibilité

Afin de structurer le code de programme objet de manière plus élégante, trois attributs de visibilité sont proposés :

1. public : les fonctions, les attributs sont accessibles par n'importe quel autre code.
2. protected : les fonctions, les attributs sont accessibles uniquement aux classes qui héritent de cette classe.
3. private : personne n'a accès aux éléments de la classe ; **c'est le mode par défaut**.

Règles

- Les attributs sont de visibilité *private*
- Réduire au maximum les accès à la visibilité *public*

Classes

Visibilité

Afin de structurer le code de programme objet de manière plus élégante, trois attributs de visibilité sont proposés :

1. public : les fonctions, les attributs sont accessibles par n'importe quel autre code.
2. protected : les fonctions, les attributs sont accessibles uniquement aux classes qui héritent de cette classe.
3. private : personne n'a accès aux éléments de la classe ; **c'est le mode par défaut**.

Règles

- Les attributs sont de visibilité *private*
- Réduire au maximum les accès à la visibilité *public*

Exemple de visilibité

```
class Position {
public:
   Position(int x, int y) :
     m_x(x), m_y(y)
   { }

   int getX();  // return m_x;
   int getY();  // return m_y;

   void move(int dx, int dy); // m_x += dx, m_y += dy

private:
   void setPosition(int x, int y); // m_x = x, m_y = y
   int m_x, m_y;
};
```

Ici, l'utilisateur de la classe est limité à la fonction `move` pour modifier les attributs de la classe

Contrairement aux trois autres modes de visibilité, `friend` donne un accès **total** aux éléments de la classe à une fonction globale ou à une autre classe.

Friend

La classe B et la fonction f ont accès à tous les éléments de la classe A.

```
class A {
private:
  int a;

public:
  A();

  friend class B;
  friend void f();
};
```

Le mot clé const

Contenu constant

- Variable, attribut etc. : ne peut modifier le contenu

Exemples

```
const int toto = 12;
toto = 13;                  // erreur de compilation

void addExtension(const std::string& filename)
{
  filename += ".txt";  // erreur de compilation
}

void show(const std::vector < int >& tab)
{
  std::vector < int >::const_iterator it;
  for (it = tab.begin(); it != tab.end(); it++) {
    std::cout << *it << "\n";
  }
}
```

Le mot clé const

Contenu constant

Le mot-clé const peut être utilisé avec les pointeurs. Deux syntaxes = deux significations bien différentes :

- le contenu de la zone mémoire pointée ne peut pas être modifiée :

Exemple

```
const int * x; // ou int const * x;
x = &y; // autorise
*x = 1; // illegal
```

- la valeur du pointeur ne peut pas être modifiée :

Exemple

```
int * const x;
*x = 1; // autorise
x = &y; // illegal
```

Le mot clé const

Contenu constant

- rien ne peut être modifiée :

Exemple

```
const int * const x; \\ ou int const * const x;
x = &y; // illegal
*x = 1; // illegal
```

Le mot clé const

Code constant

- Fonction membre d'une classe : ne peut modifier les membres de la classe

Exemple

```
class Position {
public:
  void setPosition(double x, double y) const {
    m_x = x;     // erreur de compilation, la fonction
    m_y = y;     // setPosition est constante
  }

private:
  double m_x;
  double m_y;
};
```

Classes

Le mot clé `this`

Le mot clé **this** réfère l'objet lui-même. C'est un pointeur constant.

Première utilisation

Les noms des paramètres d'une méthode sont les mêmes que les attributs de la classe, `this` permet de lever l'ambiguïté.

```
class A {
  int a;

  void f(int a) { this->a=a; }
};
```

Deuxième utilisation

Passer en paramètre la référence de l'objet lui-même.

```
class A {
  void f(...) {... g(*this); ...}
  void g(const A& a) {...}
};
```

Classes

Le mot clé `static`

- Les attributs statiques sont communs à toutes les instances de la classe ; on parle alors d'attribut de classe ;

```
class A {
  static int a;
};

int A::a = 0;
```

- Les méthodes statiques sont des méthodes de classe, il n'est pas nécessaire d'instancier une classe pour les invoquer ;

```
class A {
  static int f();
};

int A::f() { ... }
// appel
int a = A::f();
```

- Les méthodes statiques ne peuvent pas accéder à `this`.

Le concept d'héritage permet d'améliorer une classe existante en :

- surchargeant les méthodes existantes si la visibilité le permet ;
- permet d'agréger des comportements et les données de classes ;
- spécialisant un objet dans une tâche.

Exemple

```
class Position {
  ...
};

class PositionNommee : public Position {
  ...

  const std::string getName() const           // ajout d'une methode
  { return name; }          // retournant cette chaine de caracteres

private:
  std::string name;      // ajout d'un attribut chaine de caracteres
};
```

- Une classe peut hériter d'une seule ou plusieurs classes.

Exemple

```
class C : public A, public B {
  ...
};
```

- Lors de l'héritage, l'ensemble des attributs et des méthodes de la ou des classes mères est incorporé à la définition de la classe fille ;
- Aucun mot clé désigne directement la classe mère. En cas d'ambiguïté, il faut utiliser la portée (nom_class_mère::x).

Classes

Héritage et visibilité

- `class B : public A  ... ;`
  - un objet de type B (ou un pointeur sur ...) pourra toujours être vu comme un objet de type A (ou comme un pointeur sur ...) ;
  - accès depuis B aux membres public et protected de A.
- `class B : private A  ... ;`
  - un objet de type B (ou un pointeur sur ...) pourra être vu comme un objet de type A (ou comme un pointeur sur ...) depuis B ou les amis de B (seul B et ses amis savent que B hérite de A) ;
  - les membres public et protected de A ne sont pas accessibles dans les sous-classes de B.
- `class B : protected A  ... ;`
  - un objet de type B (ou un pointeur sur ...) pourra être vu comme un objet de type A depuis B, les amis de B et les sous-classes de B.

Le constructeur d'une sous-classe fait appel à un constructeur de sa ou ses classes mères.

Constructeur par défaut

Par défaut, C++ fait appel au constructeur par défaut.

Le constructeur de la classe mère est appelé avant celui de la sous-classe.

Destructeur

Les destructeurs sont invoqués dans l'ordre inverse.

Exemple

Le constructeur de la sous-classe doit fournir les arguments au constructeur de la ou des super-classes.

```
class C : public A, public B {
   C(int a, int b):A(a), B(b) { }
};
```

Le comportement du constructeur par recopie est différent selon que les constructeurs par recopie de la super-classe (A) et de la sous-classe (B) existent ou non.

Premier cas

B n'a pas de constructeur par recopie : il y a donc appel au constructeur par recopie par défaut de B. Cela provoque l'appel du constructeur par recopie de A. Si A n'a pas de constructeur par recopie, le constructeur par recopie par défaut est utilisé.

Deuxième cas

B a un constructeur par recopie : si celui-ci fait appel au constructeur par recopie de A dans son entête (B(const B& x):A() . . .), il sera appelé. Sinon le constructeur sans argument de A sera appelé.

Supposons que B hérite de A et considérons l'affectation entre objets de type B.

Premier cas

Si B surcharge l'opérateur =, on ne fera appel qu'à ce dernier.

Deuxième cas

Si B ne surcharge pas l'opérateur =, on fera appel à l'opérateur = de A (surchargé ou par défaut) pour les membres hérités de A et à l'affectation par défaut pour les autres.

Il est possible d'affecter un objet de la classe dérivée B à un objet de la classe de base A. Par exemple, l'énoncé: a = b; est valide.

Processus

- L'objet de la classe B est converti en objet de la classe A (casting) ;
- On fait ensuite appel à l'opérateur d'affectation (surchargée ou par défaut) de la classe A.

Illégal

L'énoncé inverse (b = a;) est illégal.

Classes

Héritage multiple

Si deux attributs ou méthodes appartenant aux deux super-classes (A et B) de C portent le même identifiant, il y a un problème d'ambiguïté.

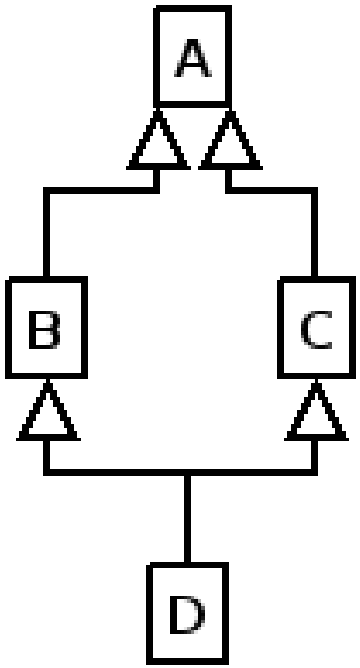
Exemple

```
class A {
  int i;
};
```

```
class B {
  int i;
};
```

```
class C : public A, public B {
  void f() {
    A::i = 5;
    A::i = 10;
  }
};
```

Si une classe hérité de manière multiple de la même classe



Info

La classe D hérite deux fois de la classe A. Dans la majorité des cas, n'a aucun intérêt.

Il est possible de limiter le nombre de présences d'une super-classe dans une sous-classe en la déclarant **virtual** dans l'héritage.

```
class A { ... };
class B : public virtual A { ... };
class C : public virtual A { ... };
class D : public B, public C { ... };
```

Polymorphisme

Définition

Rappel

Le polymorphisme est le troisième pilier de l'objet !

Définition

- une variable x de type pointeur (ou référence) sur A peut référencer une instance de type A ou une instance issue de toutes sous-classes de A ;
- l'appel à une méthode sur x doit conduire à l'appel à la bonne méthode **en cas de surcharge de la méthode invoquée**;
- polymorphisme = ligature dynamique (*Late binding*) ;
- en C++, les méthodes virtuelles.

Polymorphisme

Définition

Rappel

Le polymorphisme est le troisième pilier de l'objet !

Définition

- une variable *x* de type pointeur (ou référence) sur A peut référencer une instance de type A ou une instance issue de toutes sous-classes de A ;
- l'appel à une méthode sur *x* doit conduire à l'appel à la bonne méthode **en cas de surcharge de la méthode invoquée**;
- polymorphisme = ligature dynamique (*Late binding*) ;
- en C++, les méthodes virtuelles.

Exemple

```
class A { virtual void f() { ... } };

class B : public A { virtual void f() { ... } };

void g() {
  A *x = new B()
  x->f();
}
```

- Toute instance dispose d'un pointeur sur une table qui associe le nom des fonctions et l'adresse du code des méthodes (table des méthodes virtuelles) ;
- Il existe une table par classe ;
- Il est conseillé de déclarer les destructeurs virtuels.

Exemple

```
class A { virtual void f() { ... } };

class B : public A { virtual void f() { ... } };

void g() {
  A *x = new B()
  x->f();
}
```

- Toute instance dispose d'un pointeur sur une table qui associe le nom des fonctions et l'adresse du code des méthodes (table des méthodes virtuelles) ;
- Il existe une table par classe ;
- Il est conseillé de déclarer les destructeurs virtuels.

Les classes abstraites

Définition

- Une classe abstraite est une classe ayant au moins une méthode abstraite ;
- La ou les méthodes abstraites sont dites des fonctions virtuelles pures ;
- Une méthode abstraite ne possède pas de déclaration mais seulement d'une signature ;
- Une classe abstraite ne peut pas être instanciée ;
- Une sous-classe d'une classe abstraite ne redéfinissant pas toutes les méthodes abstraites est elle-même abstraite.

Exemple

```
virtual double perimeter() const =0;
```

Définition

- Il est possible de surcharger la plupart des fonctions et des opérateurs du langage C++ ;
- Il est possible de définir des méthodes dont le nom est un opérateur.

Exemple

```
class Complex {
private :
  double re,im;
public:
  Complex(double r =  0.0, double i = 0.0) : re(r), im(i) { }
  Complex operator+(const Complex& c) const;
};
```

Surcharge des opérateurs

Définition

Usage

```
void f() {
  Complex c1(1,1), c2(2,3);
  Complex c3 = c1 + c2;
}
```

- Tous les opérateurs ne sont pas surchargeables (. :: .* ?: sizeof # ##) ;
- Tous les autres opérateurs peuvent être surchargés suivant certaines conditions :
  - le nombre d'opérandes, la priorité et l'associativité ne peuvent être changés ;
  - on ne peut pas redéfinir de nouveaux opérateurs ;
  - on ne peut pas redéfinir les opérateurs sur les types primitifs ;

Surcharge des opérateurs

Définition

- Il est conseillé de ne pas modifier le sens d'un opérateur (operator+ pour une soustraction, par exemple) ;

Syntaxe

La syntaxe est la suivante : **type operator@(...);**

Surcharge des opérateurs

Affectation

- La surcharge d'opérateurs de type +=, -=, . . .nécessite une forme de retour différent ;
- Ces opérateurs offrent la particularité de pouvoir s'utiliser en cascade (comme tous les opérateurs d'affectation) ;
- Pour répondre à cette exigence, la fonction doit retourner une référence sur l'objet lui-même.

Exemple

```
Complex& Complex::operator+=(
                        const Complex& x) {
  re += x.re;
  im += x.im;
  return *this;
}
```

Usage

```
void f() {
  Complex c1(1,4);
  Complex c2(5,9);
  Complex c3(7,1);

  c1 += c2 += c3;
  c1 = (c2 += c3) + c2;
}
```

Surcharge des opérateurs

Indexation

- L'opérateur d'indexation [] suivi le même modèle que les opérateurs d'affectation ;
- Le retour doit s'effectuer par référence, afin de pouvoir modifier l'élément indexé ;
- Le type de l'argument n'est pas nécessairement un entier.

Exemple

```
char& String::operator[](int i) {
  if ((i < 0) or (i >= size))
    return buffer[0];
  else
    return buffer[i];
}
```

Usage

```
void f() {
  String a("abc");

  a[2] = 'd'; // a = "abd"
}
```

Surcharge des opérateurs

Affectation

Définition

Toute classe possède, par défaut, un opérateur d'affectation qui affecte attribut par attribut la valeur des attributs de l'instance source.

Usage

```
class A {
private:
  int m_a;

public:
  A(int p_a = 0) : m_a(p_a) {  }
  A& operator=(const A& a) { m_a = a.m_a; return *this; }
};
```

Surcharge des opérateurs

Affectation

Définition

Toute classe possède, par défaut, un opérateur d'affectation qui affecte attribut par attribut la valeur des attributs de l'instance source.

Usage

```
class A {
private:
  int m_a;

public:
  A(int p_a = 0) : m_a(p_a) {  }
  A& operator=(const A& a) { m_a = a.m_a; return *this; }
};
```

Surcharge des opérateurs

Affectation

Pointeur

Ne pas développer d'opérateur d'affectation pose problème pour les classes possédant des attributs de type pointeur que la classe alloue (dans les constructeurs) et désalloue (dans le destructeur).

Plan

Standard Template Library

- Une partie **très importante** du langage C++ : la STL (Standard Template Library)
- La STL propose un grand nombre d'éléments :
  - des structures de données, tableaux, listes chaînes, piles
  - des algorithmes de recherches, de comptage, de suppression, de tri, ...
  - des pointeurs intelligents, des générateurs aléatoires, des chronomètres, ...
- Nous allons nous focaliser sur trois types d'éléments :
  - les conteneurs : stockent les données
  - les itérateurs : parcourent les conteneurs
  - les algorithmes : travaillent sur les conteneurs à partir des itérateurs

Standard Template Library

- Une partie **très importante** du langage C++ : la STL (Standard Template Library)
- La STL propose un grand nombre d'éléments :
  - des structures de données, tableaux, listes chaînes, piles
  - des algorithmes de recherches, de comptage, de suppression, de tri, ...
  - des pointeurs intelligents, des générateurs aléatoires, des chronomètres, ...
- Nous allons nous focaliser sur trois types d'éléments :
  - les conteneurs : stockent les données
  - les itérateurs : parcourent les conteneurs
  - les algorithmes : travaillent sur les conteneurs à partir des itérateurs

Standard Template Library

- Une partie **très importante** du langage C++ : la STL (Standard Template Library)
- La STL propose un grand nombre d'éléments :
  - des structures de données, tableaux, listes chaînes, piles
  - des algorithmes de recherches, de comptage, de suppression, de tri, ...
  - des pointeurs intelligents, des générateurs aléatoires, des chronomètres, ...
- Nous allons nous focaliser sur trois types d'éléments :
  - les conteneurs : stockent les données
  - les itérateurs : parcourent les conteneurs
  - les algorithmes : travaillent sur les conteneurs à partir des itérateurs

Standard Template Library

- Une partie **très importante** du langage C++ : la STL (Standard Template Library)
- La STL propose un grand nombre d'éléments :
  - des structures de données, tableaux, listes chaînes, piles
  - des algorithmes de recherches, de comptage, de suppression, de tri, ...
  - des pointeurs intelligents, des générateurs aléatoires, des chronomètres, ...
- Nous allons nous focaliser sur trois types d'éléments :
  - les conteneurs : stockent les données
  - les itérateurs : parcourent les conteneurs
  - les algorithmes : travaillent sur les conteneurs grâce aux itérateurs

Standard Template Library

- Une partie **très importante** du langage C++ : la STL (Standard Template Library)
- La STL propose un grand nombre d'éléments :
  - des structures de données, tableaux, listes chaînes, piles
  - des algorithmes de recherches, de comptage, de suppression, de tri, ...
  - des pointeurs intelligents, des générateurs aléatoires, des chronomètres, ...
- Nous allons nous focaliser sur trois types d'éléments :
  - les **conteneurs** : stockent les données.
  - les **itérateurs** : parcourent les conteneurs.
  - les **algorithmes** : travaillent sur les conteneurs grâce aux itérateurs.

Standard Template Library

- Une partie **très importante** du langage C++ : la STL (Standard Template Library)
- La STL propose un grand nombre d'éléments :
  - des structures de données, tableaux, listes chaînes, piles
  - des algorithmes de recherches, de comptage, de suppression, de tri, ...
  - des pointeurs intelligents, des générateurs aléatoires, des chronomètres, ...
- Nous allons nous focaliser sur trois types d'éléments :
  - les **conteneurs** : stockent les données.
  - les **itérateurs** : parcourent les conteneurs.
  - les **algorithmes** : travaillent sur les conteneurs à partir des itérateurs.

Standard Template Library

- Une partie **très importante** du langage C++ : la STL (Standard Template Library)
- La STL propose un grand nombre d'éléments :
  - des structures de données, tableaux, listes chaînes, piles
  - des algorithmes de recherches, de comptage, de suppression, de tri, ...
  - des pointeurs intelligents, des générateurs aléatoires, des chronomètres, ...
- Nous allons nous focaliser sur trois types d'éléments :
  - les **conteneurs** : stockent les données.
  - les **itérateurs** : parcourent les conteneurs.
  - les **algorithmes** : travaillent sur les conteneurs à partir des itérateurs.

Standard Template Library

- Une partie **très importante** du langage C++ : la STL (Standard Template Library)
- La STL propose un grand nombre d'éléments :
  - des structures de données, tableaux, listes chaînes, piles
  - des algorithmes de recherches, de comptage, de suppression, de tri, ...
  - des pointeurs intelligents, des générateurs aléatoires, des chronomètres, ...
- Nous allons nous focaliser sur trois types d'éléments :
  - les **conteneurs** : stockent les données.
  - les **itérateurs** : parcourent les conteneurs.
  - les **algorithmes** : travaillent sur les conteneurs à partir des itérateurs.

Standard Template Library

- Une partie **très importante** du langage C++ : la STL (Standard Template Library)
- La STL propose un grand nombre d'éléments :
  - des structures de données, tableaux, listes chaînes, piles
  - des algorithmes de recherches, de comptage, de suppression, de tri, ...
  - des pointeurs intelligents, des générateurs aléatoires, des chronomètres, ...
- Nous allons nous focaliser sur trois types d'éléments :
  - les **conteneurs** : stockent les données.
  - les **itérateurs** : parcourent les conteneurs.
  - les **algorithmes** : travaillent sur les conteneurs à partir des itérateurs.

Les conteneurs

- Les conteneurs permettent de contenir des éléments : la généricité paramétrera le type des éléments à utiliser.
- Plusieurs types de conteneurs sont utilisables : les tableaux (vector et array), les files (queue et deque) les listes/piles (list et stack), les ensembles (set et multiset) et les dictionnaires (map et multimap).
- Il n'y a pas de **classe générique de base** (classe Object du java, par exemple). Cette option aurait pu être intéressante sur bien des aspects, mais il ne faut jamais oublier que l'un des buts fondamentaux de la librairie STL est d'être la plus rapide possible (et la liaison dynamique à un coût).

Les conteneurs

- Les conteneurs permettent de contenir des éléments : la généricité paramétrera le type des éléments à utiliser.
- Plusieurs types de conteneurs sont utilisables : les tableaux (vector et array), les files (queue et deque) les listes/piles (list et stack), les ensembles (set et multiset) et les dictionnaires (map et multimap).
- Il n'y a pas de **classe générique de base** (classe Object du java, par exemple). Cette option aurait pu être intéressante sur bien des aspects, mais il ne faut jamais oublier que l'un des buts fondamentaux de la librairie STL est d'être la plus rapide possible (et la liaison dynamique à un coût).

Les conteneurs

- Les conteneurs permettent de contenir des éléments : la généricité paramétrera le type des éléments à utiliser.
- Plusieurs types de conteneurs sont utilisables : les tableaux (vector et array), les files (queue et deque) les listes/piles (list et stack), les ensembles (set et multiset) et les dictionnaires (map et multimap).
- Il n'y a pas de **classe générique de base** (classe Object du java, par exemple). Cette option aurait pu être intéressante sur bien des aspects, mais il ne faut jamais oublier que l'un des buts fondamentaux de la librairie STL est d'être la plus rapide possible (et la liaison dynamique à un coût).

Les itérateurs

- Les itérateurs sont **fondamentaux** dans la STL !
- Ils en existent de plusieurs types :
  - Accès aléatoires : tableaux.
  - Accès linéaires : les listes, les maps, les tableaux
  - etc.
- Ils permettent de manipuler les conteneurs de manière transparente.
- Un itérateur peut-être vu comme un pointeur sur un élément d'un conteneur.
- Par définition, un itérateur permet de récupérer une donnée (pour la manipuler) et de passer à la suivante pour mettre en place l'itération.

Les itérateurs

- Les itérateurs sont **fondamentaux** dans la STL !
- Ils en existent de plusieurs types :
  - Accès aléatoires : tableaux.
  - Accès linéaires : les listes, les maps, les tableaux.
  - etc.
- Ils permettent de manipuler les conteneurs de manière transparente.
- Un itérateur peut-être vu comme un pointeur sur un élément d'un conteneur.
- Par définition, un itérateur permet de récupérer une donnée (pour la manipuler) et de passer à la suivante pour mettre en place l'itération.

Les itérateurs

- Les itérateurs sont **fondamentaux** dans la STL !
- Ils en existent de plusieurs types :
  - Accès aléatoires : tableaux.
  - Accès linéaires : les listes, les maps, les tableaux.
  - etc.
- Ils permettent de manipuler les conteneurs de manière transparente.
- Un itérateur peut-être vu comme un pointeur sur un élément d'un conteneur.
- Par définition, un itérateur permet de récupérer une donnée (pour la manipuler) et de passer à la suivante pour mettre en place l'itération.

Les itérateurs

- Les itérateurs sont **fondamentaux** dans la STL !
- Ils en existent de plusieurs types :
  - Accès aléatoires : tableaux.
  - Accès linéaires : les listes, les maps, les tableaux.
  - etc.
- Ils permettent de manipuler les conteneurs de manière transparente.
- Un itérateur peut-être vu comme un pointeur sur un élément d'un conteneur.
- Par définition, un itérateur permet de récupérer une donnée (pour la manipuler) et de passer à la suivante pour mettre en place l'itération.

Les itérateurs

- Les itérateurs sont **fondamentaux** dans la STL !
- Ils en existent de plusieurs types :
  - Accès aléatoires : tableaux.
  - Accès linéaires : les listes, les maps, les tableaux.
  - etc.
- Ils permettent de manipuler les conteneurs de manière transparente.
- Un itérateur peut-être vu comme un pointeur sur un élément d'un conteneur.
- Par définition, un itérateur permet de récupérer une donnée (pour la manipuler) et de passer à la suivante pour mettre en place l'itération.

Les itérateurs

- Les itérateurs sont **fondamentaux** dans la STL !
- Ils en existent de plusieurs types :
  - Accès aléatoires : tableaux.
  - Accès linéaires : les listes, les maps, les tableaux.
  - etc.
- Ils permettent de manipuler les conteneurs de manière transparente.
- Un itérateur peut-être vu comme un pointeur sur un élément d'un conteneur.
- Par définition, un itérateur permet de récupérer une donnée (pour la manipuler) et de passer à la suivante pour mettre en place l'itération.

Les itérateurs

- Les itérateurs sont **fondamentaux** dans la STL !
- Ils en existent de plusieurs types :
  - Accès aléatoires : tableaux.
  - Accès linéaires : les listes, les maps, les tableaux.
  - etc.
- Ils permettent de manipuler les conteneurs de manière transparente.
- Un itérateur peut-être vu comme un pointeur sur un élément d'un conteneur.
- Par définition, un itérateur permet de récupérer une donnée (pour la manipuler) et de passer à la suivante pour mettre en place l'itération.

Les itérateurs

- Les itérateurs sont **fondamentaux** dans la STL !
- Ils en existent de plusieurs types :
  - Accès aléatoires : tableaux.
  - Accès linéaires : les listes, les maps, les tableaux.
  - etc.
- Ils permettent de manipuler les conteneurs de manière transparente.
- Un itérateur peut-être vu comme un pointeur sur un élément d'un conteneur.
- Par définition, un itérateur permet de récupérer une donnée (pour la manipuler) et de passer à la suivante pour mettre en place l'itération.

Les itérateurs

Un exemple avec les listes (List)

```
// initialisation par defaut avec 10 entiers nuls
std::list < int > t(10);

// sans modifications : iterateur constant
for (std::list < int >::const_iterator it = t.begin();
it != t.end(); ++it) {
  std::cout << *it << "␣";
}

// avec modifications : iterateur non constant
for (std::list < int >::iterator it = t.begin(); it != t.end(); ++it
  *it = 1;
}
```

auto

Il est possible de remplacer la déclaration du type par le mot clé **auto** dans le cas non-const.

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - std::find et std::find_if : chercher quelque chose avec ou sans condition
    - std::fill : remplir un conteneur avec une donnée
    - std::generate : générer des données à partir d'une fonction
    - std::sort : trier les éléments
    - std::max/std::max_element : rechercher le maximum
    - std::count : compter le nombre d'éléments
    - std::swap, std::iter_swap, std::swap_ranges : échanger le contenu de variables

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - std::find et std::find_if : chercher quelque chose avec ou sans condition
    - std::fill : remplir un conteneur avec une donnée
    - std::generate : générer des données à partir d'une fonction
    - std::sort : trier les éléments
    - std::max/std::max_element : récupérer le maximum
    - std::count : compter le nombre d'éléments
    - std::swap, std::iter_swap : échanger deux éléments

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - std::find et std::find_if: chercher quelque chose avec ou sans condition
    - std::fill : remplir un conteneur avec une donnée
    - std::generate : générer des données à partir d'une fonction
    - std::sort : trier les éléments
    - std::min / std::max : rechercher le min/max
    - std::count : compter le nombre d'éléments
    - std::swap, std::rotate, std::reverse, ... : changer l'ordre des éléments

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - std::find et std::find_if: chercher quelque chose avec ou sans condition
    - std::fill : remplir un conteneur avec une donnée
    - std::generate : générer des données à partir d'une fonction
    - std::sort : trier les éléments
    - std::min / std::max : rechercher le min/max
    - std::count : compter le nombre d'éléments
    - std::swap, std::rotate, std::reverse, ... : changer l'ordre des éléments

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - ★ std::find et std::find_if: chercher quelque chose avec ou sans condition
    - ★ std::fill : remplir un conteneur avec une donnée
    - ★ std::generate : générer des données à partir d'une fonction
    - ★ std::sort : trier les éléments
    - ★ std::min / std::max : rechercher le min/max
    - ★ std::count : compter le nombre d'éléments
    - ★ std::swap, std::rotate, std::reverse, ... : changer l'ordre des éléments

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - ★ std::find et std::find_if: chercher quelque chose avec ou sans condition
    - ★ std::fill : remplir un conteneur avec une donnée
    - ★ std::generate : générer des données à partir d'une fonction
    - ★ std::sort : trier les éléments
    - ★ std::min / std::max : rechercher le min/max
    - ★ std::count : compter le nombre d'éléments
    - ★ std::swap, std::rotate, std::reverse, ... : changer l'ordre des éléments

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - ★ std::find et std::find_if: chercher quelque chose avec ou sans condition
    - ★ std::fill : remplir un conteneur avec une donnée
    - ★ std::generate : générer des données à partir d'une fonction
    - ★ std::sort : trier les éléments
    - ★ std::min / std::max : rechercher le min/max
    - ★ std::count : compter le nombre d'éléments
    - ★ std::swap, std::rotate, std::reverse, ... : changer l'ordre des éléments

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - ★ std::find et std::find_if: chercher quelque chose avec ou sans condition
    - ★ std::fill : remplir un conteneur avec une donnée
    - ★ std::generate : générer des données à partir d'une fonction
    - ★ std::sort : trier les éléments
    - ★ std::min / std::max : rechercher le min/max
    - ★ std::count : compter le nombre d'éléments
    - ★ std::swap, std::rotate, std::reverse, ... : changer l'ordre des éléments

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - ★ std::find et std::find_if: chercher quelque chose avec ou sans condition
    - ★ std::fill : remplir un conteneur avec une donnée
    - ★ std::generate : générer des données à partir d'une fonction
    - ★ std::sort : trier les éléments
    - ★ std::min / std::max : rechercher le min/max
    - ★ std::count : compter le nombre d'éléments
    - ★ std::swap, std::rotate, std::reverse, ... : changer l'ordre des éléments

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - ★ std::find et std::find_if: chercher quelque chose avec ou sans condition
    - ★ std::fill : remplir un conteneur avec une donnée
    - ★ std::generate : générer des données à partir d'une fonction
    - ★ std::sort : trier les éléments
    - ★ std::min / std::max : rechercher le min/max
    - ★ std::count : compter le nombre d'éléments
    - ★ std::swap, std::rotate, std::reverse, ... : changer l'ordre des éléments

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Algorithmes

- Les algorithmes :
  - Les algorithmes permettent de manipuler les données d'un conteneur de manière transparente.
  - Les algorithmes utilisent des itérateurs linéaire ou à accès aléatoire → tous les conteneurs peuvent appliquer les algorithmes.
  - Il en existe beaucoup :
    - ★ std::find et std::find_if: chercher quelque chose avec ou sans condition
    - ★ std::fill : remplir un conteneur avec une donnée
    - ★ std::generate : générer des données à partir d'une fonction
    - ★ std::sort : trier les éléments
    - ★ std::min / std::max : rechercher le min/max
    - ★ std::count : compter le nombre d'éléments
    - ★ std::swap, std::rotate, std::reverse, ... : changer l'ordre des éléments

Un exemple avec les listes (List)

```
std::fill(t.begin(), t.end(), 1);
std::cout << *std::min_element(t.begin(), t.end(), 1) << std::endl;
```

Tableau de caracatères : char

- Les chaînes de caractères de type C sont compatibles avec le langage C++. Certaines classes C++ utilisent des char[].

Example

Exemple

```
#include <fstream>

int main(int argc, char** argv)
{
  int i = 0;

  if (argc == 2) {
    std::ifstream ifs(argv[1]);
    ifs >> i;
  }
  std::cout << i << std::endl;
}
```

Tableau de caracatères : char

- Les chaînes de caractères de type C sont compatibles avec le langage C++. Certaines classes C++ utilisent des char[].

Example

Exemple

```
#include <fstream>

int main(int argc, char** argv)
{
  int i = 0;

  if (argc == 2) {
    std::ifstream ifs(argv[1]);
    ifs >> i;
  }
  std::cout << i << std::endl;
}
```

std::string

- La classe std::string encapsule un buffer de caractères.
- Elle peut contenir plusieurs caractères de fin de ligne « 0 ».
- Une API riche :
  - operator[], size() : l'accès à un char
  - c_str() et data() : la conversion de la chaine de caractères en tableau de caractères avec ou sans le caractère fin de ligne
  - append, operator+, operator+=, insert, replace : différentes opérations : les fonctions d'ajout, d'insertion, de suppression, d'affectation ...
  - find, rfind, find_first_of, find_last_of, find_first_not_of, find_last_not_of : les fonctions de recherche
- http://www.cplusplus.com/reference/string/basic_string/ ou https://en.cppreference.com/w/cpp/string/string

std::string

- La classe std::string encapsule un buffer de caractères.
- Elle peut contenir plusieurs caractères de fin de ligne « 0 ».
- Une API riche :
  - operator[](size_t) : l'accès à un char.
  - c_str() et data() : la conversion de la chaine de caractères en tableau de caractères avec ou sans le caractère fin de ligne.
  - append, operator+, operator+=, insert, replace, erase : les opérations : les fonctions d'ajout, d'insertion, de suppression, d'affectation ;
  - find, rfind, find_first_of, find_last_of, find_first_not_of, find_last_not_of : les fonctions de recherche.
- http://www.cplusplus.com/reference/string/basic_string/ ou https://en.cppreference.com/w/cpp/string/string

std::string

- La classe std::string encapsule un buffer de caractères.
- Elle peut contenir plusieurs caractères de fin de ligne « 0 ».
- Une API riche :
  - operator[](size_t) : l'accès à un char.
  - c_str() et data() : la conversion de la chaîne de caractères en tableau de caractères avec ou sans le caractère fin de ligne.
  - append, operator+, operator+=, insert, replace, erase, assign, operator= : les fonctions d'ajout, d'insertion, de suppression, d'affectation.
  - find, rfind, find_first_of, find_first_not_of, find_last_of : les fonctions de recherche.
- `http://www.cplusplus.com/reference/string/basic_string/` ou `https://en.cppreference.com/w/cpp/string/string`

std::string

- La classe std::string encapsule un buffer de caractères.
- Elle peut contenir plusieurs caractères de fin de ligne « 0 ».
- Une API riche :
  - operator[](size_t) : l'accès à un char.
  - c_str() et data() : la conversion de la chaîne de caractères en tableau de caractères avec ou sans le caractère fin de ligne.
  - append, operator+, operator+=, insert, replace, erase, assign, operator= : les fonctions d'ajout, d'insertion, de suppression, d'affectation.
  - find, rfind, find_first_of, find_first_not_of, find_last_of : les fonctions de recherche.
- `http://www.cplusplus.com/reference/string/basic_string/` ou `https://en.cppreference.com/w/cpp/string/string`

std::string

- La classe std::string encapsule un buffer de caractères.
- Elle peut contenir plusieurs caractères de fin de ligne « 0 ».
- Une API riche :
  - operator[](size_t) : l'accès à un char.
  - c_str() et data() : la conversion de la chaîne de caractères en tableau de caractères avec ou sans le caractère fin de ligne.
  - append, operator+, operator+=, insert, replace, erase, assign, operator= : les fonctions d'ajout, d'insertion, de suppression, d'affectation.
  - find, rfind, find_first_of, find_first_not_of, find_last_of : les fonctions de recherche.
- http://www.cplusplus.com/reference/string/basic_string/ ou https://en.cppreference.com/w/cpp/string/string

std::string

- La classe std::string encapsule un buffer de caractères.
- Elle peut contenir plusieurs caractères de fin de ligne « 0 ».
- Une API riche :
  - operator[](size_t) : l'accès à un char.
  - c_str() et data() : la conversion de la chaîne de caractères en tableau de caractères avec ou sans le caractère fin de ligne.
  - append, operator+, operator+=, insert, replace, erase, assign, operator= : les fonctions d'ajout, d'insertion, de suppression, d'affectation.
  - find, rfind, find_first_of, find_first_not_of, find_last_of : les fonctions de recherche.
- http://www.cplusplus.com/reference/string/basic_string/ ou https://en.cppreference.com/w/cpp/string/string

std::string

- La classe std::string encapsule un buffer de caractères.
- Elle peut contenir plusieurs caractères de fin de ligne « 0 ».
- Une API riche :
  - operator[](size_t) : l'accès à un char.
  - c_str() et data() : la conversion de la chaîne de caractères en tableau de caractères avec ou sans le caractère fin de ligne.
  - append, operator+, operator+=, insert, replace, erase, assign, operator= : les fonctions d'ajout, d'insertion, de suppression, d'affectation.
  - find, rfind, find_first_of, find_first_not_of, find_last_of : les fonctions de recherche.
- http://www.cplusplus.com/reference/string/basic_string/ ou https://en.cppreference.com/w/cpp/string/string

std::string

- La classe std::string encapsule un buffer de caractères.
- Elle peut contenir plusieurs caractères de fin de ligne « 0 ».
- Une API riche :
  - operator[](size_t) : l'accès à un char.
  - c_str() et data() : la conversion de la chaîne de caractères en tableau de caractères avec ou sans le caractère fin de ligne.
  - append, operator+, operator+=, insert, replace, erase, assign, operator= : les fonctions d'ajout, d'insertion, de suppression, d'affectation.
  - find, rfind, find_first_of, find_first_not_of, find_last_of : les fonctions de recherche.
- `http://www.cplusplus.com/reference/string/basic_string/` ou `https://en.cppreference.com/w/cpp/string/string`

Chaînes de caractères

Utilisations courantes

La classe `std::string` encapsule un tableau de char.

- Gestion complète de l'allocation, ré-allocation
- Nombreuses fonctions de parcours

Manipulation de chaînes

```
std::string str1 = "test";
std::string str2("test2");

std::string str3 = str1 + str2;
int pos = str2.find("2");

for (int i = 0; i < str3.size(); i++) {
  std::cout << str2[i] << "\n";
}
```

std::stringstream : les flux de chaînes

- std::string ne possède de fonctions pour ajouter des entiers, réels ou booléen directement.
- Les stringstream permettent une édition des chaînes de caractères par les flux :

Exemple

```
std::ostringstream st;
st << "la valeur de x est : " << 10 << " pixels,"
<< "alors que y vaut : " << 0.345 << " points,"
<< "x == 10 ? : " << std::boolalpha << (x == 10) << ".\n";

std::string chaine(st.str()); // transforme en std::string
std::cout << chaine;
printf("La chaine : %s", chaine.c_str());
```

std::stringstream : les flux de chaînes

- std::string ne possède de fonctions pour ajouter des entiers, réels ou booléen directement.
- Les stringstream permettent une édition des chaînes de caractères par les flux :

Exemple

```
std::ostringstream st;
st << "la␣valeur␣de␣x␣est␣:␣" << 10 << "␣pixels,"
<< "alors␣que␣y␣vaut␣:␣" << 0.345 << "␣points,"
<< "x␣==␣10␣?␣:␣" << std::boolalpha << (x == 10) << ".\n"

std::string chaine(st.str()); // transforme en std::string
std::cout << chaine;
printf("La␣chaine␣:␣%s", chaine.c_str());
```

std::stringstream : les flux de chaînes

- std::string ne possède de fonctions pour ajouter des entiers, réels ou booléen directement.
- Les stringstream permettent une édition des chaînes de caractères par les flux :

Exemple

```
std::ostringstream st;
st << "la␣valeur␣de␣x␣est␣:␣" << 10 << "␣pixels,"
<< "alors␣que␣y␣vaut␣:␣" << 0.345 << "␣points,"
<< "x␣==␣10␣?␣:␣" << std::boolalpha << (x == 10) << ".\n"

std::string chaine(st.str()); // transforme en std::string
std::cout << chaine;
printf("La␣chaine␣:␣%s", chaine.c_str());
```

std::vector / std::array

Les tableaux de type C existe en C++ :

- Les tableaux alloués dans la pile :

```
double tab1[1000];
```

- Alloués dans le tas, ne pas oublier la destruction :

```
double* tab2 = new double[1000];
delete[] tab2;
```

- Alloués dans le tas à deux dimensions :

```
double** tab3 = new double*[1000];
for (size_t i = 0; i < 1000; ++i)
  tab3[i] = new double[1000];

for (size_t i = 0; i < 1000; ++i)
  delete[] tab3[i];
delete[] tab3;
```

std::vector / std::array

Les tableaux de type C existe en C++ :

- Les tableaux alloués dans la pile :

```
double tab1[1000];
```

- Alloués dans le tas, ne pas oublier la destruction :

```
double* tab2 = new double[1000];
delete[] tab2;
```

- Alloués dans le tas à deux dimensions :

```
double** tab3 = new double*[1000];
for (size_t i = 0; i < 1000; ++i)
  tab3[i] = new double[1000];

for (size_t i = 0; i < 1000; ++i)
  delete[] tab3[i];
delete[] tab3;
```

std::vector / std::array

Les tableaux de type C existe en C++ :

- Les tableaux alloués dans la pile :

```
double tab1[1000];
```

- Alloués dans le tas, ne pas oublier la destruction :

```
double* tab2 = new double[1000];
delete[] tab2;
```

- Alloués dans le tas à deux dimensions :

```
double** tab3 = new double*[1000];
for (size_t i = 0; i < 1000; ++i)
  tab3[i] = new double[1000];

for (size_t i = 0; i < 1000; ++i)
  delete[] tab3[i];
delete[] tab3;
```

std::vector / std::array

- Gestion des tableaux C est problématique dans le sens où il faut faire très attention aux accès, à l'allocation, désallocation etc.
- Les classes std::vector et std::array (C++11) sont de très bons remplaçants :
  - Gestion automatique de la mémoire, allocation, réallocation et destruction.
  - Coût de $O(1)$ pour l'accès aléatoire et la suppression en fin de tableau
  - Coût de $O(n)$ pour l'ajout ou la suppression d'une case.
- Exemple :

std::vector / std::array

- Gestion des tableaux C est problématique dans le sens où il faut faire très attention aux accès, à l'allocation, désallocation etc.
- Les classes std::vector et std::array (C++11) sont de très bons remplaçants :
  - Gestion automatique de la mémoire, allocation, réallocation et destruction.
  - Coût de $O(1)$ pour l'accès aléatoire et la suppression en fin de tableau.
  - Coût de $O(n)$ pour l'ajout ou la suppression d'une case.
- Exemple :

std::vector / std::array

- Gestion des tableaux C est problématique dans le sens où il faut faire très attention aux accès, à l'allocation, désallocation etc.
- Les classes std::vector et std::array (C++11) sont de très bons remplaçants :
  - Gestion automatique de la mémoire, allocation, réallocation et destruction.
  - Coût de $O(1)$ pour l'accès aléatoire et la suppression en fin de tableau.
  - Coût de $O(n)$ pour l'ajout ou la suppression d'une case.
- Exemple :

std::vector / std::array

- Gestion des tableaux C est problématique dans le sens où il faut faire très attention aux accès, à l'allocation, désallocation etc.
- Les classes std::vector et std::array (C++11) sont de très bons remplaçants :
  - Gestion automatique de la mémoire, allocation, réallocation et destruction.
  - Coût de $O(1)$ pour l'accès aléatoire et la suppression en fin de tableau.
  - Coût de $O(n)$ pour l'ajout ou la suppression d'une case.
- Exemple :

std::vector / std::array

- Gestion des tableaux C est problématique dans le sens où il faut faire très attention aux accès, à l'allocation, désallocation etc.
- Les classes std::vector et std::array (C++11) sont de très bons remplaçants :
  - Gestion automatique de la mémoire, allocation, réallocation et destruction.
  - Coût de $O(1)$ pour l'accès aléatoire et la suppression en fin de tableau.
  - Coût de $O(n)$ pour l'ajout ou la suppression d'une case.
- Exemple :

Exemples

```
std::array < int, 10 > tab1;
tab1.fill(5);

std::vector < double > tab2(100);

std::vector < std::vector < double > > tab3(100);
for (size_t i = 0; i < 100; ++i) {
  tab3[i].resize(100);
}
```

std::vector / std::array

- Gestion des tableaux C est problématique dans le sens où il faut faire très attention aux accès, à l'allocation, désallocation etc.
- Les classes std::vector et std::array (C++11) sont de très bons remplaçants :
  - Gestion automatique de la mémoire, allocation, réallocation et destruction.
  - Coût de $O(1)$ pour l'accès aléatoire et la suppression en fin de tableau.
  - Coût de $O(n)$ pour l'ajout ou la suppression d'une case.
- Exemple :

Exemples

```
std::array < int, 10 > tab1;
tab1.fill(5);

std::vector < double > tab2(100);

std::vector < std::vector < double > > tab3(100);
for (size_t i = 0; i < 100; ++i) {
  tab3[i].resize(100);
}
```

std::vector / std::array

- Gestion des tableaux C est problématique dans le sens où il faut faire très attention aux accès, à l'allocation, désallocation etc.
- Les classes std::vector et std::array (C++11) sont de très bons remplaçants :
  - Gestion automatique de la mémoire, allocation, réallocation et destruction.
  - Coût de $O(1)$ pour l'accès aléatoire et la suppression en fin de tableau.
  - Coût de $O(n)$ pour l'ajout ou la suppression d'une case.
- Exemple :

Exemples

```
std::array < int, 10 > tab1;
tab1.fill(5);

std::vector < double > tab2(100);

std::vector < std::vector < double > > tab3(100);
for (size_t i = 0; i < 100; ++i) {
  tab3[i].resize(100);
}
```

std::vector et std::array

Les fonctions membres :

- operator[](int) et at(int) les opérateurs d'accès aléatoires aux éléments, le deuxième lève une exception si l'entier passé en paramètre sort du tableau

```
std::array < int, 5 > p;
std::vector < int > v(5);

std::cout << p[6] << std::endl;
std::cout << v[6] << std::endl;
std::cout << v.at(6) << std::endl; // levee d'exception
```

- size_t size() pour connaître la taille d'un tableau

std::vector et std::array

Les fonctions membres :

- operator[](int) et at(int) les opérateurs d'accès aléatoires aux éléments, le deuxième lève une exception si l'entier passé en paramètre sort du tableau

```
std::array < int, 5 > p;
std::vector < int > v(5);

std::cout << p[6] << std::endl;
std::cout << v[6] << std::endl;
std::cout << v.at(6) << std::endl; // levee d'exception
```

- size_t size() pour connaître la taille d'un tableau

std::vector - uniquement

- resize(size_t) et reserve(size_t) les fonctions pour, respectivement, modifier sa taille ou préallouer une zone mémoire
- push_back(T), pop_back(), les fonction d'ajout et de suppression d'élément en fin de tableau (coût $O(1)$).

Exemple

```
std::vector < double > v(1000, 1.0);

v.push_back(2.0);
```

std::vector - uniquement

- resize(size_t) et reserve(size_t) les fonctions pour, respectivement, modifier sa taille ou préallouer une zone mémoire
- push_back(T), pop_back(), les fonction d'ajout et de suppression d'élément en fin de tableau (coût $O(1)$).

Exemple

```
std::vector < double > v(1000, 1.0);

v.push_back(2.0);
```

std::vector et les itérateurs

Les itérateurs permettent de manipuler plus facilement les éléments des conteneurs de la STL :

Exemple

```
std::vector < std::string > t(200);

for (std::vector < std::string>::const_iterator it = t.begin();
     it != t.end(); ++it) {
       std::cout << *it << ‘‘ ’’;
     }

auto fd = std::find(t.begin(), t.end(), "debian");

if (fd != t.end()) {
  v.erase(fd);
}
```

auto

On peut remplacer la déclaration des itérateurs par le mot clé **auto**.

std::vector et les itérateurs

Depuis le C++11, les itérations peuvent se faire avec des `range-based for`.

Range-based for

- simplification des boucles `for`
- possible pour tout container disposant de la notion d'itérateur

Exemple

La variable `s` va prendre les valeurs successives de `t`

```
std::vector < std::string > t(200);

for (const std::string& s: t) {
  std::cout << s << ' ';
}
```

Les containers classiques

Les listes chaînées : `std::list`

Les listes chaînées, pour des ajouts et suppression rapide :

- Allocation dynamique de la mémoire

Allocation

```
// definition d'une liste chainees de reels
std::list < double > lst;

// creation d'une liste chainee 1.0, 2.0, 3.0
lst.push_back(2.0);
lst.push_front(1.0);
lst.push_back(3.0);
```

Complexité

Ajout, suppression, accès en début ou fin de liste : **O(1)**

Les containers classiques

Les listes chaînées : std::list

Accès aux données

```
// Definition d'une liste chainees d'objets
std::list < Object > lst;
[...]

// Affiche tous les objets de la liste chainee
std::list < Object >::iterator it;
for (it = lst.begin(); it != lst.end(); ++it) {
  std::cout << *it << "\n";
}

// Recherche un objet dans la liste egal Ã  Object("test")
// surcharge de l'operateur ==
it = std::find(lst.begin(), lst.end(), Object("test"));
if (it == lst.end()) {
  std::cout << "Objet non trouve\n";
} else {
  std::cout << "Objet trouve\n";
}
```

Une sorte de dictionnaire : à une clé on attache une valeurs

- la clé peut être de tous type s'il existe l'opérateur de comparaison
- la donnée est quelconque

Ajout de données

```
// Creation d'un dictionnaire dont le type de la cle est
// une chaine de caracteres et de donnees de type reelles.
std::map < std::string, double > dico;

dico["a"] = 1.0;
dico["toto"] = 0.125;

std::cout << dico["toto"] << "\n"; // affiche 1.0
std::cout << dico["titi"] << "\n"; // affiche 0.0, titi n'existe pas mais
```

Tester l'existence d'une clé

```
std::map < std::string, int > dico;
[...]
std::map < std::string, int >::iterator it = dico.find("cherche");
if (it ==  dico.end()) {
  std::cout << "La␣cle␣n'existe␣pas␣dans␣le␣dico.";
}
```

Parcours linéaire

```
std::map < std::string, int >::iterator it;
for (it = dico.begin(); it != dico.end(); it++) {
  std::cout << "cle.....:␣" << it->first
  << "valeur...:␣" << it->second
  << "\n";
}
```

Containers

Les pair

std::pair

- structure contenant deux éléments ou objets de types différents ou non
- premier élément référencé par l'attribut `first` et le deuxième par `second`
- utilisé par exemple lors d'un parcours d'une `map` ; le premier élément étant le clé et le second la valeur associée à la clé
- initialisation via les constructeurs ou la méthode `make_pair`

```
std::pair < std::string, double> p1("C++", -8.2);
std::pair < std::string, double> p2;

p2 = std::make_pair("Java", 12.9)
```

Containers

Les double listes chaînées : `std::deque`

Principe

- liste d'éléments accessibles par le début ou la fin de la liste
- ajout et suppression uniquement en début et fin de liste

```
std::deque <int> list;

list.push_back(10);
list.push_front(20);

list.pop_front();
list.pop_back();
```

Complexité

Ajout, suppression, accès en début ou fin de liste : **O(1)**

Containers

Encore des containers

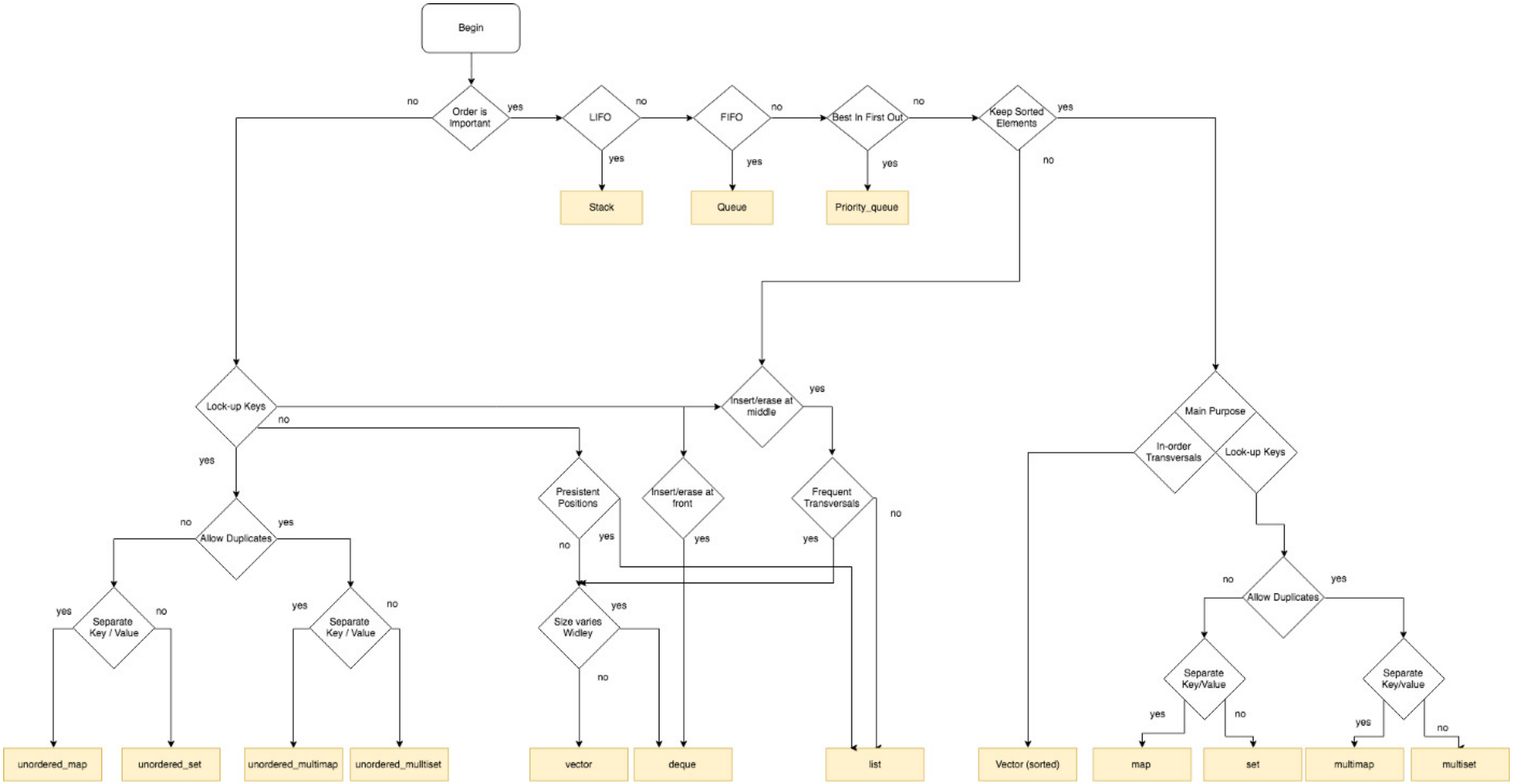
- `std::stack` : mécanisme d'une pile
- `std::queue` : liste de type FIFO
- `std::priority_queue` : liste de type FIFO **triée**
- `std::set` : liste ordonnée d'éléments **uniques**
- `std::multiset` : liste ordonnée d'éléments

Containers - complexité

Container	Insertion	Access	Erase	Find	Persistent Iterator
vector / string	Back: O(1) or O(n) Other: O(n)	O(1)	Back: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	No
deque	Back/Front: O(1) Other: O(n)	O(1)	Back/Front: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-

(c) https://medium.com/

Containers - choix



(c) https://medium.com/

Plan

UML vers C++

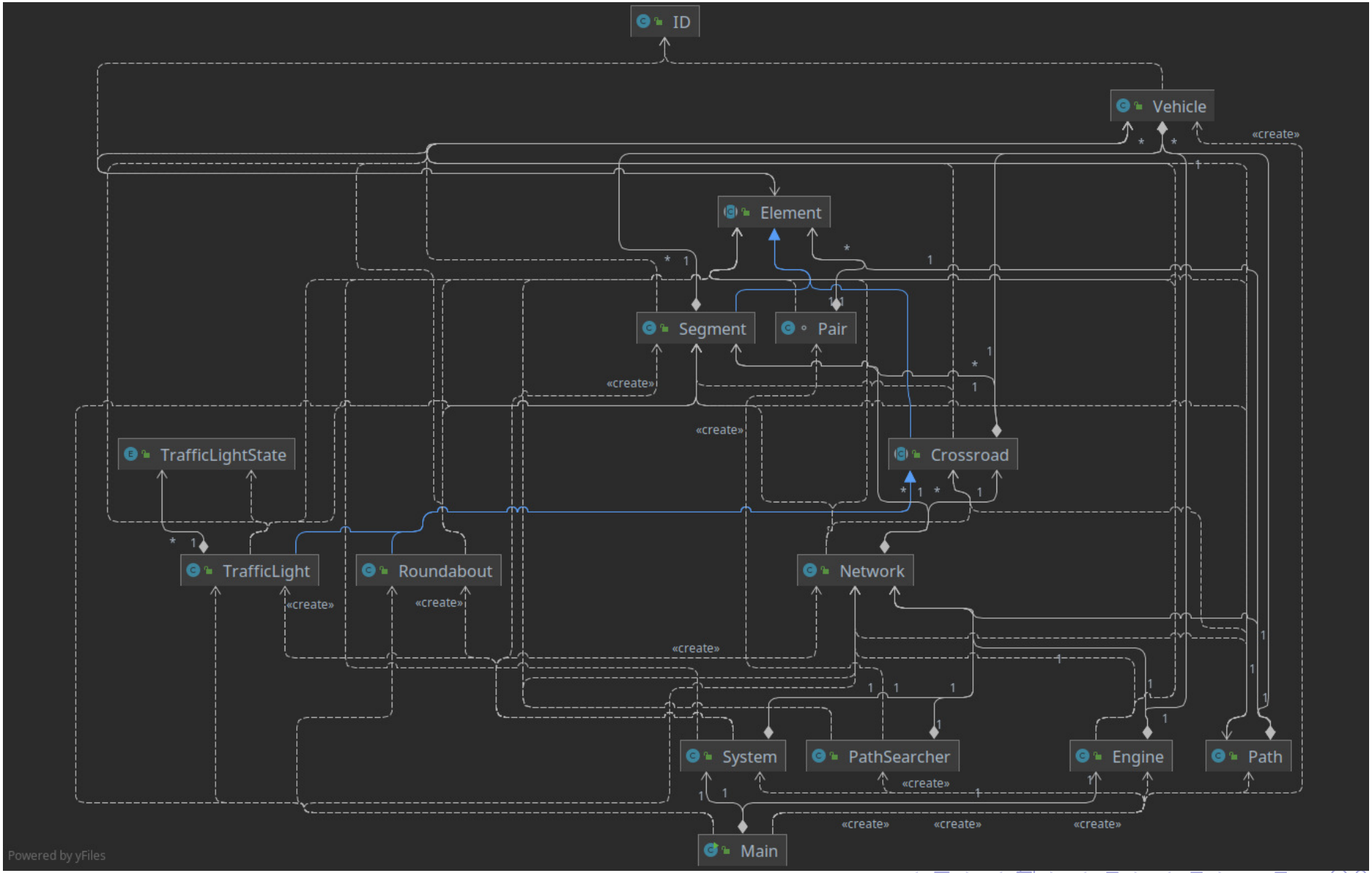
Génération de code

- A partir d'un diagramme de classes UML, il est possible de générer du code C++ (ou Java)
- Plusieurs règles de traduction peuvent être définies pour les différents éléments :
  - relation : association, agrégation et composition
  - héritage
  - ...
- Le mécanisme inverse est possible : le *retro-engineering*

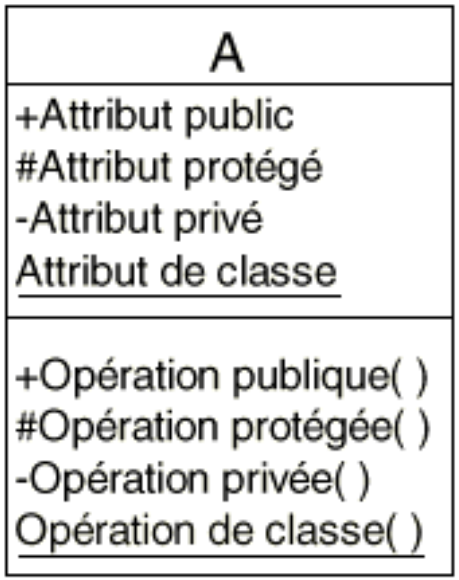
UML 2.5

`https://www.omg.org/spec/UML/2.5/PDF/`

UML vers C++

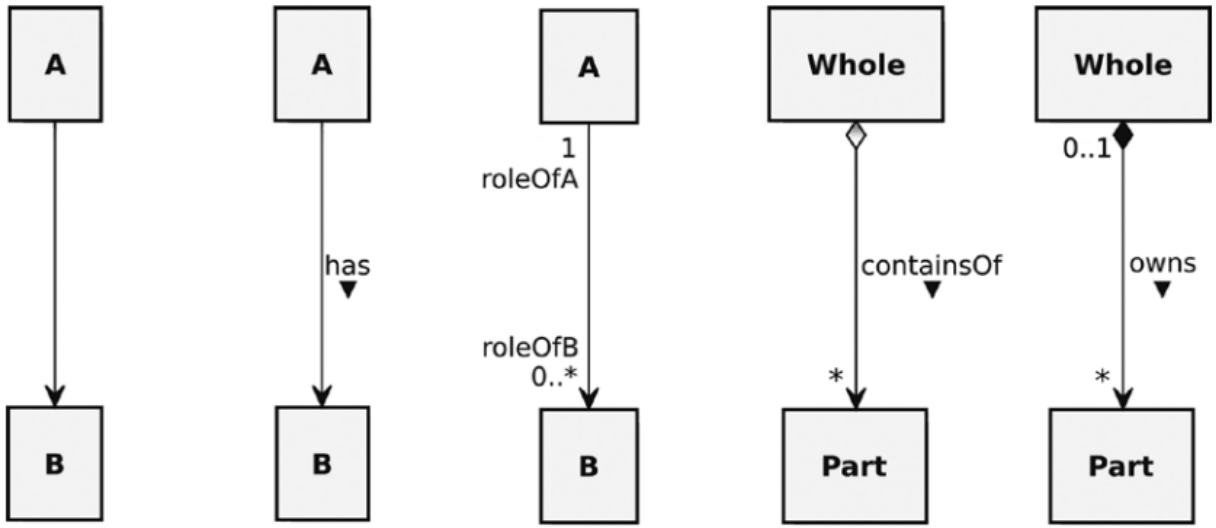

ID
Vehicle
Element
Segment
Pair
TrafficLightState
Crossroad
TrafficLight
Roundabout
Network
System
PathSearcher
Engine
Path
Main
«create»
Powered by yFiles

UML vers C++




```
class A {
  public :
  // + Attributs publics
  protected:
  // # Attributs proteges
  private:
  // - Attributs prives
  public:
  // Attributs de classe
  static type attr;

  public:
  // + Operations publiques
  protected:
  // # Operations protegees
  private:
  // - Operations privees
  public:
  // Operations de classe
  static type f(...);
};
```

UML vers C++



Association/Agrégation/Composition

- association : A est associé à B
- agrégation : A est composé de B mais B peut exister sans A (lien faible)
- composition : B est contenu dans A (lien fort) - si A est détruit, B l'est aussi

UML vers C++

Relation

Association A->B de type “1-1”

```
class B { ... };

class A {
  public:
    A(B& b):b(b) {...}
  private:
    B& b;
};
```

Si un rôle est spécifié sur B, alors il devient l’identifiant de l’attribut

Association A->B de type “1-1” et de type “has” ou Composition

```
class B { ... };

class A {
  public:
    A():b(...) {...}
  private:
    B b;
};
```

Il est possible de définir la classe B dans la classe A pour renforer le lien de composition.

Association A<->B de type “1-1” (pointeur intelligent)

```
class B;

class A {
  public:
    A(B* b = nullptr);
  private:
    B* b;

    friend class B;
};
```

```
class B {
  public :
    B(A* a = nullptr) : a(a) {
      if (this->a) a->b = this;
    }
  private :
    A* a;

    friend class A;
};

A::A(B* b) : b(b) {
  if (this->b) b->a = this;
}

void main() {
  A a;
  B b(&a);
}
```

Association A<->B de type “1-1”

```
class B;
class A {
 public: A() {}
 private:
  std::weak_ptr<B> b;
  void link(const std::weak_ptr<B>& b
  { this->b = b; }
  friend void make_association(
    const std::shared_ptr<A>& a,
    const std::shared_ptr<B>& b);
};

void make_association(
 const std::shared_ptr<A>& a,
 const std::shared_ptr<B>& b)
{
  a->link(b);
  b->link(a);
}
```

```
class B {
 public: B() {}
 private :
  std::weak_ptr<A> a;
  void link(const std::weak_ptr<A>& a
  { this->a = a; }
  friend void make_association(
    const std::shared_ptr<A>& a,
    const std::shared_ptr<B>& b);
};

int main () {
  auto a = std::make_shared<A>();
  auto b = std::make_shared<B>();

  make_association(a, b);
  return 0;
}
```

Cette traduction utilise des

UML vers C++

Relation

Association A<->B de type “0-1”

```
class B { ... };

class A {
  public:
    A(B* b = nullptr):b(b) {...}
  private:
    B* b;
};
```

Le destructeur ne prends pas en charge la destructeur.
Le pointeur pourrait être pris en charge par un `weak_ptr` si l’instance a été créé dynamiquement.

Agrégation A->B de type 0..1 ou 1

```
class B { ... };

class A {
  public:
    A(B* b = nullptr):b(b) {...}
    ~A() { delete b; }
  private:
    B* b;
};
```

On peut aussi remplacer le pointeur par un `std::unique_ptr`.

UML vers C++

Relation

Association A->B de type “0-*”

```
class B { ... };

class A {
  public:
    A() {...}
    A(std::initializer_list< B* > lb) :
      lb(lb) {...}
    void add(B* b) { vb.push_back(b); }
  private:
    std::vector< B* > vb;
};
```

Le pointeur sur B peut être remplacé par un `std::shared_ptr`.

UML vers C++

Relation

Composition A->B de type “0-*”

```
class A {
  public:
    class B { ... };

    A() : vb({B(...), B(...), ...})
    {...}

  private:
    std::vector< B > vb;
};
```

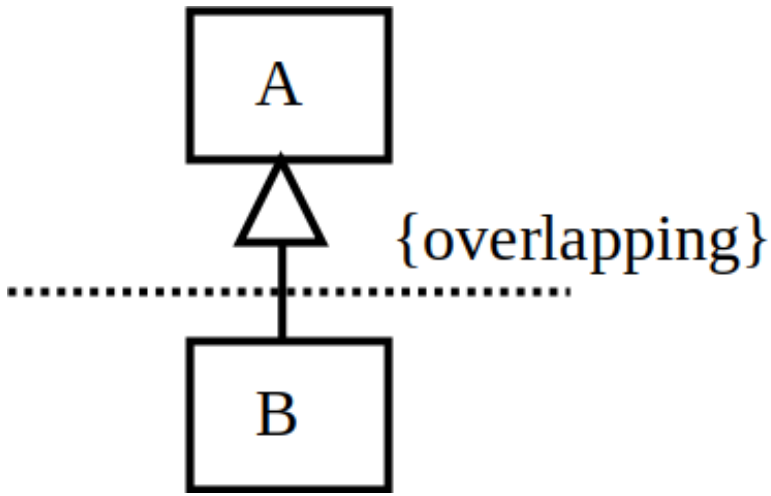
Si le nombre de B n’est pas constant alors les instances sont gérés par des pointeurs de type `std::unique_ptr` et leur instantiation doit être réalisés dans le constructeur.

UML vers C++

Héritage

```
class A { ... };

class B { ... };

class C : public A, public B { ... };
```

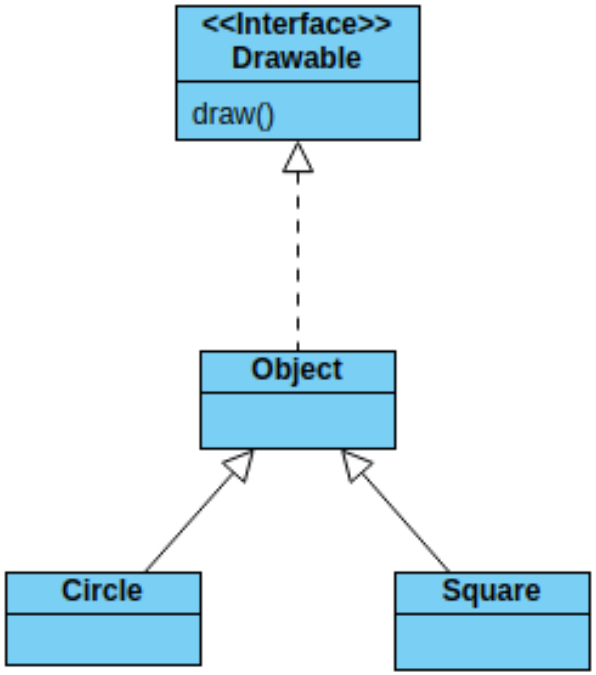
Possibilité d'ajouter une contrainte sur les liens d'héritage pour spécifier la non-duplication des attributs




```
class A { ... };
class B : virtual public A { ... };
```

UML vers C++

Interface et implémentation




```
class Drawable {
  public:
    virtual void draw() = 0;
};

class Object : public Drawable {...};
```

La traduction n'est pas parfaite car on créé un lien de type "est un" entre l'interface et la classe qui l'implémente.

Utilisation des stéréotypes

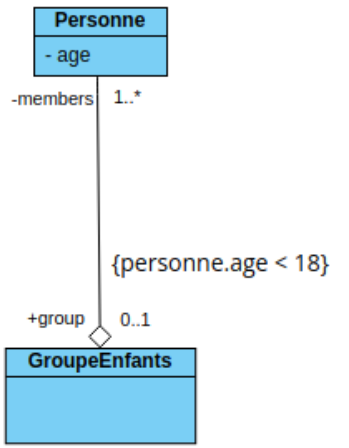
UML vers C++

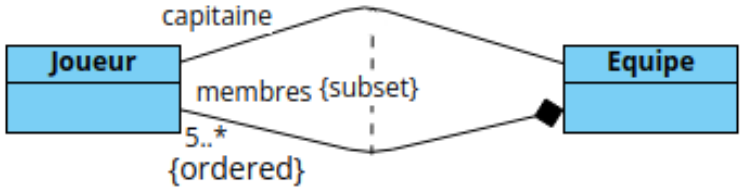
Encore plus !

Contraintes

- le langage OCL (*Object Constraint Language*) : {*personne.age* < 18}
- les contraintes : *ordered*, *subset*, *unique*, *frozen*, *or*

Visibilité des rôles





Permet de spécifier la visibilité d'un rôle et donc de l'attribut

ordered

Utilisation d'un set trié

```
class Joueur {
  public:
    bool operator<(Joueur const& j) const
    { return numero < j.numero; }
};

class Equipe {
  private:
    std::set < std::shared_ptr < Joueur > > membres;
};

int main()
{
    Equipe e{{std::make_shared<Joueur>(1)}};

    return 0;
}
```

Plan

Les fonctions lambda

Définition

- définir une fonction avec un environnement de capture (ou contexte)
- capture = ensemble des variables manipulables par la fonction
- on parle aussi de `closure` (une fermeture ou une clôture)

Utilisation

- utilisable avec les algorithmes de la STL : `std::for_each`, `std::find_if`, par exemple
- définir des objets fonction : `std::function`

```
std::vector<int> nums{3, 4, 2, 8, 15, 267};

std::for_each(nums.cbegin(), nums.cend(), [](const int& n) {
  std::cout << " " << n;
});
```

Les fonctions lambda

Avant C++11

Définition d'un `functor` : une classe possédant une surcharge de l'opérateur ()

```
class FoisDeux
{
  public:
    int operator()(int x) { return x * 2; }
};

A a;

std::cout << a(5) << std::endl;
```

Les fonctions lambda

Définition de l'environnement

- via les paramètres de la lambda (entre paranthèses)
- via la définition du contexte (entre crochets)

```
std::vector<int> v{0, 1, 2, 3, 4};

auto result = std::find_if(v.begin(), v.end(), [](int i){
  return (i > 0) and (i % 2 == 0);
});
```

Contexte

Ensemble de variables définie :

- par recopie : une copie de la variable est effectuée
- par référence : la variable est accessible

Les fonctions lambda

Capture par recopie

- recherche du nombre d'éléments divisibles par N
- la fonction `count_if` a besoin d'une fonction à un paramètre = la variable à tester
- N est une variable locale et doit être accessible à la lambda

```
std::vector<int> v{ 1, 2, 3, 4, 4, 3, 7, 8, 9, 10 };
const int N = 3;

int num = std::count_if(v.begin(), v.end(), [N](int i){
  return i % N == 0;
});
```

Les fonctions lambda

Capture par référence

- trie un ensemble d'entiers en ordre décroissant
- lambda à 2 paramètres : le couple de valeurs à tester
- incrémentation de la variable locale N à chaque comparaison
  → N est passée par référence

```
std::vector<int> v{1, 8, 3, 7, 2, 4};
int N = 0;

std::sort(v.begin(), v.end(), [&N](const auto& a, const auto& b) {
      ++N;
      return a > b;
});
```

Les fonctions lambda

Capture par défaut

- `[=]` : toutes les variables locales sont passées par valeur
- `[&]` : toutes les variables locales sont passées par référence

this ?

Si une lambda est définie dans une méthode d'une classe, alors `this` peut être capturé
→ les attributs de l'objet sont accessibles dans la lambda

- `[this]` : tous les attributs sont passés par référence
- `[*this]` : tous les attributs sont passés par valeur (C++17)

Les fonctions lambda

```
struct Foo {
 int m_x = 0;

 void func() {
  int x = 0;

  //Implicit capture 'this'
  [&]() { /*access m_x and x*/ }();

  //Redundant 'this'
  [&, this]() { /*access m_x and x*/ }();

  //Implicit capture 'this'
  [=]() { /*access m_x and x*/ }();
 }
};
```

Plan

Les exceptions

Définitions

Exception

Le programme idéal est un programme sans erreur !

- Le compilateur est le premier niveau de contrôle des erreurs dites de syntaxe, de type voire de sémantique ;
- Le deuxième niveau de contrôle fait intervenir la notion d' exception ;
- Lors du développement, on ne peut jamais savoir ce que l'utilisateur de la classe ou du programme va faire ;

Exemple

Par exemple, lors des spécifications, si le concepteur a imposé que le paramètre x d'une fonction f ne soit jamais nul. Si néanmoins l'utilisateur appelle cette fonction avec 0, que se passe-t-il ? Le programme se plante et s'arrête définitivement !!!

Les exceptions

Définitions

Les exceptions permettent de séparer le bloc d'instructions de la gestion des erreurs pouvant survenir dans un bloc.

Exemple

```
#include <exception>

try {
// Code pouvant lever des exceptions
}
catch (type ou type variable){
// Gestion de l'exception
}
```

Try/catch

Le mot clé **try** indique que le code du bloc peut soulever des exceptions et que si c'est le cas alors il y aura exécution d'un bloc marqué par le mot clé **catch**.

Les exceptions

Définitions

Les exceptions permettent de séparer le bloc d'instructions de la gestion des erreurs pouvant survenir dans un bloc.

Exemple

```
#include <exception>

try {
// Code pouvant lever des exceptions
}
catch (type ou type variable){
// Gestion de l'exception
}
```

Try/catch

Le mot clé **try** indique que le code du bloc peut soulever des exceptions et que si c'est le cas alors il y aura exécution d'un bloc marqué par le mot clé **catch**.

Try/catch

Un bloc catch prend en paramètre un type quelconque (classe, par exemple). On peut en définissant plusieurs blocs catch prenant différents types d'exception réalise un traitement différent selon l'exception soulevée.

Exemple

```
#include <exception>

// ...
try {
  char *ptr = new char[1000000000];  /
  // ... suite en cas de succÃšs de new (improbable ...)
}
catch ( bad_alloc ) {
  // en cas d'Ã©chec d'allocation mÃ©moire par new
  // une exception bad_alloc est lancÃ©e par new

  // traitement de l'erreur d'allocation
```

Try/catch

Un bloc catch prend en paramètre un type quelconque (classe, par exemple). On peut en définissant plusieurs blocs catch prenant différents types d'exception réalise un traitement différent selon l'exception soulevée.

Exemple

```
#include <exception>

// ...
try {
  char *ptr = new char[1000000000];  /
  // ... suite en cas de succÃšs de new (improbable ...)
}
catch ( bad_alloc ) {
  // en cas d'Ã©chec d'allocation mÃ©moire par new
  // une exception bad_alloc est lancÃ©e par new

  // traitement de l'erreur d'allocation
```

Les exceptions

Levée et propagation

- La levée d'une exception provoque une remontée dans l'appel des méthodes jusqu'à ce qu'un bloc catch acceptant cette exception soit trouvé ;
- Si aucun bloc catch n'est trouvé, une fonction spéciale terminate() est appelée et le programme s'arrête. Il est possible de changer cette fonction à l'aide de la fonction set_terminate();
- Un bloc catch(...) intercepte toutes les exceptions ;
- L'appel à une méthode pouvant lever une exception doit :
  - soit être contenu dans un bloc try/catch ;
  - soit être situé dans une méthode propageant (throw) cette classe d'exception. La méthode ne traite pas l'exception mais laisse la méthode appelante traite l'exception (Remarque : void f() throw() indique que la fonction f ne propage aucune exception).
- Par défaut, une méthode peut propage toutes les exceptions.

Propagation

- Si une exception est levée dans une fonction ou méthode ne propageant pas l'exception alors la fonction unexcepted() est appelée et le programme est arrêté ;
- On peut définir sa propre fonction unexpected() par un appel à la fonction set_unexpected() définie dans except.h comme :

```
typedef void (*unexpected_function)();
unexpected_function set_unexpected(unexpected_function t_fun
```

Création d'une exception

Définition d'une nouvelle exception par la déclaration d'une nouvelle classe.

```
#include <exception>
#include <iostream>

class Erreur { };

class Essai
{
public :
  ...
  void f1() { throw Erreur(); }
};

void main() {
  try {
    Essai e1;

    e1.f1();
  }
  catch (Erreur) {
    std::cout << "interception de Erreur" << std::endl;
  }
```

Plan

Le langage définit les entrées sorties sous forme d'objets et via l'utilisation des opérateurs `operator<<` et `operator>>` :

Sortie dans la console et dans un fichier

```
int i = 13;
double j = 3.1415;

// sortie dans la console
std::cout << "Sortie␣standard␣" << i << "␣" << j << std::endl;

// sortie dans un fichier nomme out.dat
std::ofstream fichier("out.dat");
fichier << "Sortie␣standard␣" << i << "␣" << j << std::endl;
```

Remarque

Ajouter un retour à la ligne est réalisé par un caractère `\n` ou via `std::endl`. Ce dernier vide également les caches (`flush`).

Le langage définit les entrées sorties sous forme d'objets et via l'utilisation des opérateurs `operator<<` et `operator>>` :

Sortie dans la console et dans un fichier

```
int i = 13;
double j = 3.1415;

// sortie dans la console
std::cout << "Sortie␣standard␣" << i << "␣" << j << std::endl;

// sortie dans un fichier nomme out.dat
std::ofstream fichier("out.dat");
fichier << "Sortie␣standard␣" << i << "␣" << j << std::endl;
```

Remarque

Ajouter un retour à la ligne est réalisé par un caractère `\n` ou via `std::endl`. Ce dernier vide également les caches (`flush`).

Entrée depuis la console ou depuis un fichier

```
double x, double y;

// recuperation depuis la console
std::cin >> x >> y;

// recuperation depuis un fichier
std::ifstream fichier("out.dat");
fichier >> x >> y;
```

Remarque

Pour lire une ligne entière d'un fichier on utilise `std::getline(std::ostream&, std::string)`

Flux

Les entrées / sorties

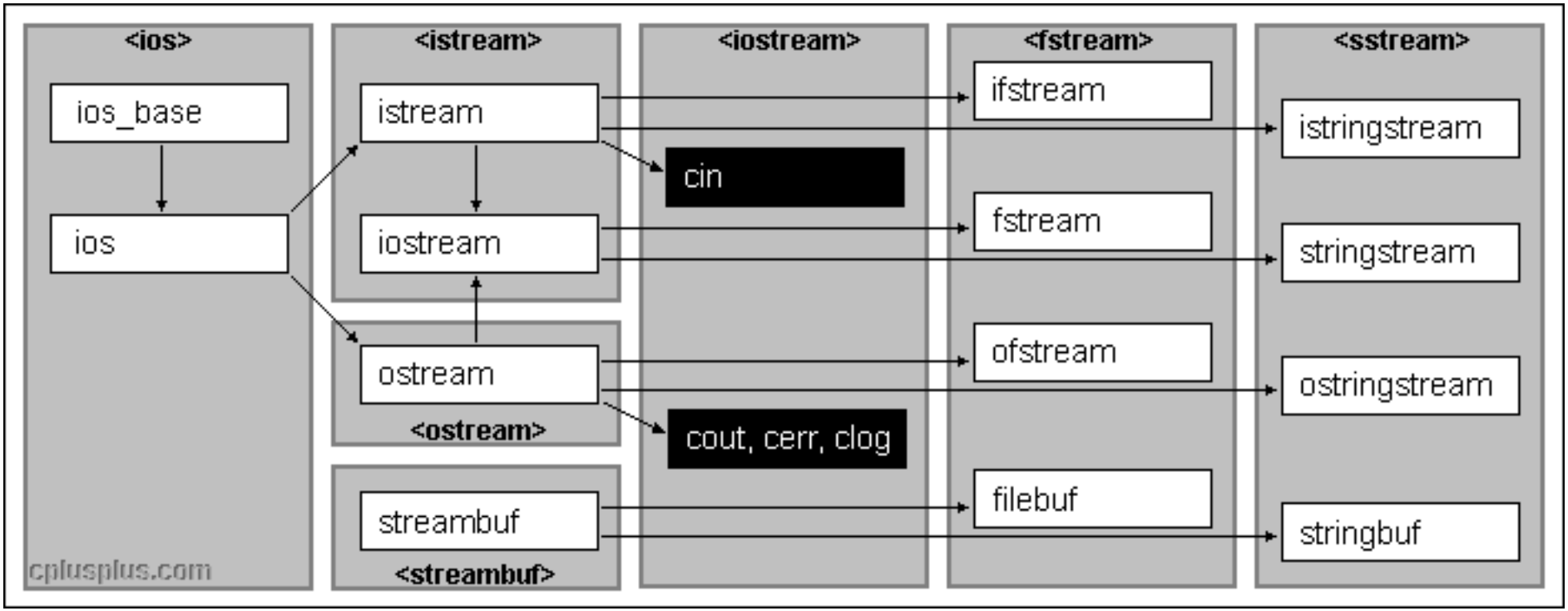


Figure – L'arbre d'héritage des flux (sources : `http://www.cplusplus.com`)

Flux

Les entrées / sorties

- L'écriture dans un `stream` s'effectue à l'aide de l'opérateur << :

```
ostream& operator<<(...);
```

- Le type de retour permet d'utiliser l'opérateur << en cascade.

```
((std::cout << "hello") << "world");
```

- La lecture dans un stream s'effectue à l'aide de l'opérateur >> :

```
istream& operator>>(...);
```

- Les opérateurs << et >> sont définis pour tous les types primitifs.

- Si vous désirez que les instances de vos classes puissent s'afficher à l'écran ou s'enregistrer dans un fichier, il faut surcharger l'opérateur <<
- Il ne peut être écrit qu'en dehors d'une classe stream (on ne peut pas modifier le code d'une classe des librairies standards !) et évidemment en dehors de vos classes (car le premier opérande est un stream !)
- Il faut donc définir l'opérateur << à l'extérieur de vos classes et en général, ami (`friend`) de vos classes afin d'avoir accès à leurs membres

Remarque

Les opérateurs << et >> sont définis sur tous les types primitifs ainsi que sur les chaînes de caractères.

Flux

Les entrées / sorties

```
class A {
private :
  int m_a;
public :
  A(int p_a):m_a(p_a) { }

  friend ostream& operator<<(ostream&, const A&);
};

ostream& operator<<(ostream& o, const A& a)
{
  o << a.m_a;
  return o;
}
```

Fichier - écriture

- par défaut, un `ofstream` est ouvert en mode “nouveau” (`out`) et en mode texte
- s’il existe déjà, le contenu sera écrasé
- d’autres modes :
  - app : mode ajout
  - binary : le fichier est considéré comme un ensemble d’octets

Fichier - lecture

- par défaut, un `ifstream` est ouvert au début et en mode texte
- d’autres modes :
  - ate : pointeur de fichier à la fin du fichier
  - binary : le fichier est considéré comme un ensemble d’octets

Fichier - écriture

- par défaut, un `ofstream` est ouvert en mode “nouveau” (`out`) et en mode texte
- s’il existe déjà, le contenu sera écrasé
- d’autres modes :
  - app : mode ajout
  - binary : le fichier est considéré comme un ensemble d’octets

Fichier - lecture

- par défaut, un `ifstream` est ouvert au début et en mode texte
- d’autres modes :
  - ate : pointeur de fichier à la fin du fichier
  - binary : le fichier est considéré comme un ensemble d’octets

Définition

- fonctions qui modifient le comportement d'un flux d'entrée/sortie
- un exemple : afficher un entier en base 16 (hexadécimale)

```
std::cout << std::setbase(16) << 100 << std::endl;;
```

- `std::endl` est aussi un `manipulator` (sans paramètre)

`Manipulator` avec paramètres

- setw(val) : définir la taille du champ
- setfill(c) : définir le caratère de remplissage
- setprecision(val) : définir la précision des nombres flottants
- setiosflags(flag) : définir des flags de formatage

```
cout << setiosflags(ios::showbase | ios::uppercase);
```

- resetiosflags(m) : supprimer un ou plusieurs flags de formatage

Flux

`file system` - C++17

Problématique

- rendre portable la gestion du système de fichiers
- exprimer les chemins et la navigation
- accéder aux types et aux permissions des fichiers

- la classe `path` représente un chemin
- des méthodes pour décomposer le chemin (root_name, root_directory, root_path, relative_path, parent_path, filename, stem et extension) ou pour tester
- des fonctions du namespace `std::filesystem` : exists, copy, remove, rename, ...

Référence

`en.cppreference.com/w/cpp/filesystem`

Plan

La généricité

Définitions

Définition 1

La généricité a pour but de paramétrer les classes ou les fonctions de manière à ne pas à avoir à réécrire une classe alors que seul le type d'un attribut change, par exemple

Définition 2

Les templates sont des classes ou des fonctions dépendants d'un certain nombre de paramètres (type ou valeur)

Macro

Les templates sont des sortes de macros avec un contrôle de type réalisé à la compilation

La généricité

Définitions

Définition 1

La généricité a pour but de paramétrer les classes ou les fonctions de manière à ne pas à avoir à réécrire une classe alors que seul le type d'un attribut change, par exemple

Définition 2

Les templates sont des classes ou des fonctions dépendants d'un certain nombre de paramètres (type ou valeur)

Macro

Les templates sont des sortes de macros avec un contrôle de type réalisé à la compilation

La généricité

Définitions

Définition 1

La généricité a pour but de paramétrer les classes ou les fonctions de manière à ne pas à avoir à réécrire une classe alors que seul le type d'un attribut change, par exemple

Définition 2

Les templates sont des classes ou des fonctions dépendants d'un certain nombre de paramètres (type ou valeur)

Macro

Les templates sont des sortes de macros avec un contrôle de type réalisé à la compilation

Définition 3

Une définition d'une classe template s'effectue à l'aide du mot-clé `template` et d'une liste de paramètres.

```
template <typename T> class A { ... };
```

- La classe A peut utiliser T pour remplacer le type d'une donnée membre ou le type d'un argument d'une fonction membre ou le type de la valeur retournée par une fonction membre ;
- Une fois définie, la classe template A pourra être instanciée avec un type particulier ;
- La classe A ne peut pas être utilisée sans être instanciée.

`A<int> x; ou A<B> y;`

Définition 3

Une définition d'une classe template s'effectue à l'aide du mot-clé `template` et d'une liste de paramètres.

```
template <typename T> class A { ... };
```

- La classe A peut utiliser T pour remplacer le type d'une donnée membre ou le type d'un argument d'une fonction membre ou le type de la valeur retournée par une fonction membre ;
- Une fois définie, la classe template A pourra être instanciée avec un type particulier ;
- La classe A ne peut pas être utilisée sans être instanciée.

```
A<int> x; ou A<B> y;
```

La généricité

Paramètres multiples

- Un template peut dépendre de plusieurs paramètres ;

```
template <typename T, typename U> class A { ... };
```

- Certains de ces paramètres peuvent être valués (`int` ou `long`);

```
template <typename T, int size> class A { ... };
const int n = 50;
A < int, 10 > x; // instantiation par une valeur
A < int, n > y; // instantiation par une variable const
```

- A chaque instanciation, le code généré est dupliqué autant de fois que les paramètres du template sont différents.

Paramètre par défaut

Un paramètre d'un template peut possèder un type ou une valeur par défaut.

```
template <typename T, typename S = size_t > class Vector {
...
};
```

hpp/cpp

- les templates doivent être définis dans les fichiers `header`
- possibilité de déporter le corps des méthodes ou fonctions dans un autre fichier

```
// file A.hpp
template <typename T>
class A {
  public:
  void f();
};
#include "A.tpp"
```

```
// file A.tpp
template <typename T>
void A<T>::f()
{ ... }
```

Template ou polymorphisme ?

- polymorphisme :
  - héritage
  - redéfinition de méthodes
  - liaison dynamique
- template :
  - production de types ou fonctions à la compilation
  - liaison statique

Exemple

```
template <typename T>
void f(const T &t)
{ t.g(); }
```

La classe T doit disposer de la méthode g. Si g est une méthode virtuelle alors le polymorphisme sera utilisé.

```
template <typename T>
void f(const T &t)
{ g(t); }
```

La fonction g doit exister et dont accepter un paramètre de type T.

Idée

Laisser le compilateur déterminer le type des paramètres du template

Exemple

```
// specification complete
std::pair < int, double > p = {1, 5.2};

// utilisation du mot cle auto
auto p = std::make_pair(1, 5.2);

// en C++17
std::pair p = {1, 5.2};
```

Exemple

```
template < typename T > class Vector {
  public:
  Vector(int);
  Vector(std::initializer_list<T>);
  //...
};

// Vector <int>
Vector v1{1,2,3};
Vector v2 = v1;

// pointeur sur un Vector<int>
auto p = new Vector{1,2,3};

// le type des elements n'est pas deductible
Vector<int> v3(1);
```

Exemple

Via le constructor avec un "initializer_list<T>"

```
Vector<std::string> vs1{"Hello", "world"};

// Vector de "const char*"
Vector vs2{"Hello", "world"};

// Vector avec des strings
Vector vs3{"Hello"s, "world"s};

// error : les types ne sont pas homogenes
Vector vs4{"Hello"s, "world"};
```

La généricité

Déduction

```
template < typename T >
class Vector
{
public:
    Vector(std::initializer_list<T>)
    { }

    template<typename Iter>
    Vector(Iter, Iter)
    { }

    Iterator<T> begin() const
    { return Iterator<T>{}; }
};
```

```
template < typename T >
struct Iterator
{   // alias
    using value_type = T;

    Iterator<T> operator+(int)
    { return Iterator<T>{}; }
};

Vector v1{1,2,3,4,5};
Vector v2(v1.begin(), v1.begin()+2);
```

Deduction guide

```
template<typename Iter>
Vector(Iter, Iter) -> Vector<typename Iter::value_type>;
```

Les concepts - C++20

Constat

Lorsque l'on spécifie un paramètre, le compilateur va vérifier si le type est compatible avec son utilisation.

```
template<typename Seq, typename Num>
Num sum(Seq s, Num v)
{
  for (const auto& x: s) {
    v += x;
  }
  return v;
}
```

- Seq doit impérativement possèder les méthodes `begin` et `end`
- Num doit offrir l'opérateur += avec les éléments de s

Utilisation des concepts

`typename` est remplacé par des concepts

```
template<Sequence Seq, Number Num>
  requires Arithmetic<Value_type<Seq>, Num>
Num sum(Seq s, Num v)
{
  for (const auto& x: s) {
    v += x;
  }
  return v;
}
```

- Seq doit respecter les contraintes définies par le concept Sequence
- le type des valeurs contenues dans Seq doit être mathématiquement compatible avec Num

Définition de concepts

Possibilité de définir de nouveaux concepts

```
template <typename T>
concept Equality_comparable =
  requires (T a, T b) {
    { a == b } -> bool;
    { a != b } -> bool;
  };
```

- le nouveau concept vérifie si le type T possède les opérateurs == et != et qu'une valeur booléenne est générée

Plan

Numerics

Bibliothèque de calcul

- les fonctions mathématiques classiques (`cmath`) : fabs, sqrt, sin, ...
- depuis C++17, les fonctions mathématiques spéciales (`cmath`) : beta, bessel, certaines intégrales, ...
- depuis C++20, les constantes numériques comme pi, e, sqrt2_v, ...

Des algorithmes

- pgcd (`gcd`), ppcm (`lcm`), interpolation linéaire (`lerp`), ...
- la générateur de nombres aléatoires avec des algortihmes tels que Mersenne Twister
- les rationnels (à la compilation)
- les bits (`bit`)

Algorithme

Implémentation d'une somme

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int sum = std::accumulate(v.begin(), v.end(), 0);

int product = std::accumulate(v.begin(), v.end(), 1,
                              std::multiplies<int>());
```

L'opérateur appliqué lors du cumul peut être changé

```
template<typename InputIt, typename T, typename BinaryOperation>
constexpr T accumulate(InputIt first, InputIt last, T init,
                       BinaryOperation op)
{
    for (; first != last; ++first) {
        init = op(std::move(init), *first);
    }
    return init;
}
```

Alternatives

Problématique

Pouvoir spécifier plusieurs types de retour pour une fonction.

Première solution : variant

```
std::variant<std::string, int> get_message(std::istream& s)
{
  // read stream
  if (...) return std::string(...);
  if (...) return number; // int
}

auto m = get_message(s);

if (std::holds_alternative<std::string>(m))
{
  std::string t = m.get<std::string>();
} else { ... }
```

Alternatives

Deux autres possibilités

- optional : la fonction get_message retourne soit un string soit rien

```
std::optional<std::string> get_message(std::istream& s)
{ ... return std::string(...); ... return {}; }

if (auto m = get_message(s)) { ... } else { ... }
```

- any : on ne spécifie pas les types de retour possibles

```
std::any get_message(std::istream& s)
{ ... }

auto m = get_message(s);
try {
  string& t = std::any_cast<std::string>(m);
  ...
  } catch (std::bad_any_access) { ... }
```

Time - C++11-14-20

Le temps - chrono

- mesurer, convertir et exprimer

Mesurer

```
auto start = std::chrono::steady_clock::now();
// to do something
auto end = std::chrono::steady_clock::now();
std::chrono::duration<double> elapsed_seconds = end - start;
std::cout << "elapsed time: " << elapsed_seconds.count() << "s\n";
```

UTC - C++20

```
std::chrono::time_point<std::chrono::utc_clock> epoch;

std::cout << std::format("The time of the Unix epoch was {0:%F}T{0:%R%z}."
                         epoch);
```

```
The time of the Unix epoch was 1970-01-01T00:00+0000.
```

Exprimer - C++14

Surcharge de l'opérateur ""xxx(...)

constexpr std::chrono::milliseconds operator "" ms(unsigned long long ms);

```
using namespace std::chrono_literals;

auto d1 = 250ms;
std::chrono::milliseconds d2 = 1s;
std::cout << "250ms = " << d1.count() << " milliseconds\n"
          << "1s = " << d2.count() << " milliseconds\n";
```

```
250ms = 250 milliseconds
1s = 1000 milliseconds
```

Calendar - C++20

Manipulation de dates

```
std::cout << std::boolalpha;

std::chrono::weekday wd {0}; // Sunday is 0 or 7
--wd;
std::cout << (wd == std::chrono::Saturday) << ' ';
++wd;
std::cout << (wd == std::chrono::Sunday) << '\n';
```

Encore plus fort

```
constexpr auto ym {year(2021)/8};
constexpr auto md {9/day(15)};
constexpr auto mdl {October/last};
constexpr auto mw {11/Monday[3]};
constexpr auto mwdl {December/Sunday[last]};
```

Concurrency - C++11

Définition

Exécution de plusieurs tâches simultanément

- exécution parallèle
- accès concurrent et partage de données
- synchronisation

std::thread

Création de threads

```
void f(int k)
{
    for (int i = 0; i < 5; ++i) {
      std::cout << "Thread " << k
                << " executing\n";
      std::this_thread::sleep_for(
        std::chrono::milliseconds(10));
    }
}
```

```
int main()
{
  std::thread t1(f, 1);
  std::thread t2(f, 2);

  t1.join();
  t2.join();
  return 0;
}
```

Accès concurrent

- être sûr que plusieurs processus ne manipulent pas une donnée simultanément
- utilisation de `mutex` et de verrous
  - `scoped_lock` : verrou sur plusieurs mutex dans un bloc
  - `shared_lock` : verrou partagé par plusieurs threads
  - `unique_lock` : verrou à accès unique

Writer - Reader

```
std::shared_mutex write;

// One write, no reads.
void write_fun()
{
  std::lock_guard<std::shared_mutex> lock(write);
  // do write
}

// Multiple reads, no write
void read_fun()
{
  std::shared_lock<std::shared_mutex> lock(write);
  // do read
}
```

Condition variable

```
class Message { ... };
queue<Message> q;
condtion_variable c;
mutex m;

void consumer()
{
  while (true) {
    unique_lock lck(m);
    c.wait(lck,[]() {
      return not q.empty();
    });
    auto a = q.front();
    q.pop();
    lck.unlock();
    // ... use message
  }
}
```

```
void producer()
{
  while (true) {
    Message a(...);
    scoped_lock lck(m);
    q.push(a);
    c.notify_one();
  }
}
```

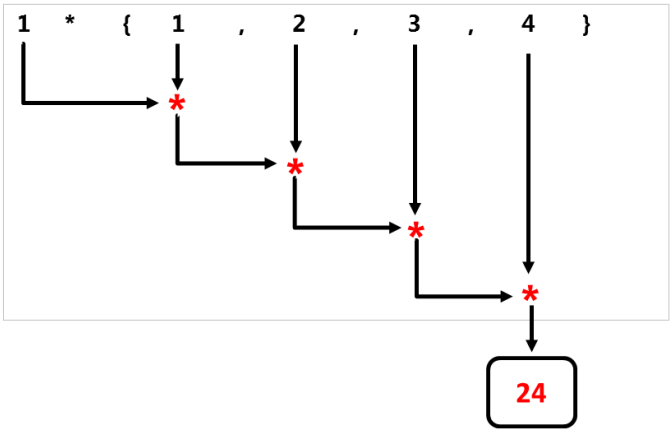
promise/future

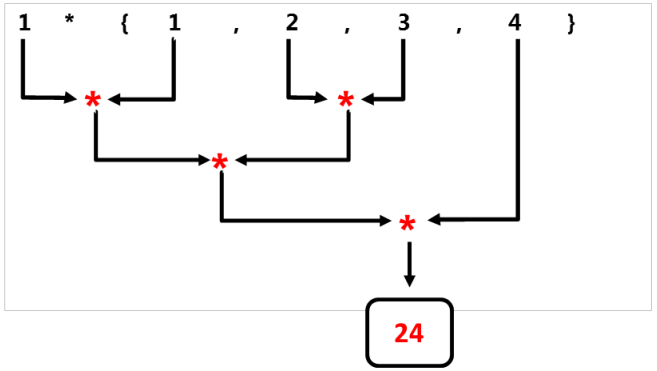
- définition d'une promesse d'un résultat futur fournit par un thread
- le thread "set" le résultat dès qu'il est disponible
- le processus principal atteint le résultat (`get`)

```
void product(std::promise<int>&& p, int a, int b) {
  // ....
  p.set_value(a * b);
}

int main()
{
  std::promise<int> p;
  std::future<int> r = p.get_future();
  std::thread t(product, std::move(p), 10, 20);

  std::cout << "20␣*␣10␣=␣" << r.get() << std::endl;

  t.join();
}
```

Algorithme parallèle - C++17

- la plupart des algorithmes de la STL sont exécutables en parallèle
  → utilisation des registres xmm du CPU (un registre = 4 ints)
- l'algorithme `reduce`





Reduce

```
// sequentiel : 705 ms
std::accumulate(v.cbegin(), v.cend(), 0.0)};
// parallele : 54 ms
std::reduce(std::execution::par, v.cbegin(), v.cend())};
```

Licence

Cours C++

eric.ramat@univ-littoral.fr