# LO02

## Principes et pratique de la Programmation Orientée Objet


Antoine Jouglet antoine.jouglet@utc.fr

## Avant-propos

Ce poly est composé d'un ensemble de diapositives dont certaines ont été vues en cours alors que d'autres apportent quelques précisions sur certains points afin de compléter le discours. Les diapositives sont volontairement denses afin d'apporter toutes les informations nécessaires à l'étude.

Les diapositives comportent peu d'exemples. Cependant ceux-ci sont réunis sous la forme d'exercices corrigés dans le poly qui complète ce cours. Lors de l'étude il est alors **fortement recommandé** de faire les exercices correspondants à chaque chapitre. Les corrections de ces exercices comportent souvent de nombreux détails qui permettent de mieux assimiler le cours.

### Remarque

Il est très probable qu'il subsiste des fautes d'orthographe et des erreurs. Vous pouvez aider à améliorer ce document en détectant les erreurs et les envoyant à `antoine.jouglet@utc.fr`. Toute proposition de thème d'exercice est aussi la bienvenue.

Merci pour votre aide !

# SOMMAIRE


| | |
|---|---|
| **Éléments de base de programmation en C++** | |
| C++ introduction | 7 |
| Éléments et structure d'un programme en C++ | 8 |
| Les données en C++ | 11 |
| Les types du C++ | 14 |
| Adressage indirect | 18 |
| Allocation dynamique en C++ | 21 |
| Fonctions : transmissions d'arguments et de valeurs de retour, arguments par défaut, surcharge de fonctions, fonctions inline, fonctions constexpr | 22 |
| Espaces de noms | 25 |
| **Concepts de la programmation orientée objet** | |
| Notions d'objet et de classe | 29 |
| L'approche orientée objet | 30 |
| Classes (implémentation, représentation) | 32 |
| Le principe d'encapsulation | 37 |
| Naissance d'un objet (instanciation) | 40 |
| Mort d'un objet | 42 |
| Les membres statiques | 43 |
| Surcharge des opérateurs en C++ | 44 |
| Objets et conversions | 46 |
| Détection des erreurs et gestion des exceptions | 47 |
| Liens entre objets, associations entre classes | 51 |
| Copies et affectations entre objets | 54 |
| Les pointeurs intelligents (C++11) | 56 |
| Hiérarchies de classes | 58 |
| Héritage | 61 |
| Héritage et redéfinition | 63 |
| Héritage et redéfinition du constructeur de recopie et d'affectation | 64 |
| Héritage multiple | 65 |
| Le principe de substitution | 66 |
| Le polymorphisme | 67 |
| Les classes abstraites | 69 |
| L'héritage private et protected | 71 |
| Transtypage | 72 |
| Identification dynamique de type | 73 |
| Interface et comportement : problématiques | 74 |
| Héritage : spécificateurs de redéfinition (C++11) | 76 |
| L'héritage virtuel | 77 |

| | |
|---|---|
| La programmation générique | 78 |
| Définir des nouveaux patrons en C++ | 80 |
| Patrons de fonctions | 82 |
| Patrons de classes | 84 |
| Le mot clé typename | 86 |
| **Patrons de conception** | |
| Les design patterns | 89 |
| Design pattern - Template Method | 92 |
| Design pattern - Strategy | 94 |
| Design pattern - Adapter | 96 |
| Design pattern - Composite | 98 |
| Design pattern - Factory method | 100 |
| Design pattern - Singleton | 103 |
| Design pattern - Iterator | 105 |
| **Bibliothèques** | |
| La bibliothèque standard de C++ | 109 |
| La classe string | 111 |
| Quelques éléments sur les flux | 112 |
| Les exceptions standards | 114 |
| STL - généralités | 115 |
| STL - Les itérateurs | 117 |
| STL - Les conteneurs séquentiels | 119 |
| STL - Les conteneurs associatifs | 123 |
| STL - Les algorithmes | 126 |
| Bibliographie | 129 |
| Lexique (français, anglais, chinois) | 131 |

# Éléments de base de programmation en C++

1

# C++
# Introduction



## Origines de la POO

2

- Simula (Simple universal language) :
  - a introduit le paradigme orienté objet en programmation, en 1960.
  - Il est considéré comme le premier langage à objet et le prédécesseur de langages ultérieurs.
- C'est réellement par et avec Smalltalk (1972) que la programmation par objets débute et que sont posés les concepts de base de l'orienté objet : objet, messages, encapsulation, héritage, polymorphisme, etc.
- Années 80 : effervescence des langages à objets : Objective C (1980), C++ (1983), Eiffel (1984), Common Lisp Object System.
- Années 90 : extension de la programmation par objets dans les différents secteurs du développement logiciel.



## Présentation du C++ [wikipedia]

3

- Apparu en1983 (dernière révision en 2011, m.a.j. en 2014 et 2017)
- Auteur : Bjarne Stroustrup (laboratoire Bell d'AT&T )
- Dialectes : ANSI C++ 1998, ANSI C++ 2003, C++11, C++14, C++17, C++20
- Influencé par C, Simula, Ada 83, ALGOL 68, CLU, ML
- A influencé Ada 95, C#, Java, PHP, D, X++
- On peut considérer que C++ « est du C » avec un ajout de fonctionnalités (bien que certains programmes syntaxiquement corrects en C ne le sont pas en C++).
- C++ est un langage de programmation permettant la programmation sous plusieurs paradigmes :
  - programmation impérative et procédurale,
  - programmation orientée objet.
- Années 1990 : est devenu l'un des langages de programmation les plus populaires dans l'industrie informatique.



## Programmation Orientée Objet

4

- C++ utilise les concepts de la programmation orientée objet et permet notamment :
  - la classification,
  - l'encapsulation,
  - la composition de classes,
  - l'association de classes,
  - l'héritage, qui permet le polymorphisme,
  - l'abstraction,
  - la programmation générique.



## « Hello world »

5

```
#include <iostream>
#include <string>
using namespace std;
int main() {
  cout.operator<<("Bonjour!\n");
  cout<<"What is your name?\n";
  string name;
  cin>>name;
  cout<<"Hello "<<name<<"!\n";
  return 0;
}
```

1

# Éléments et structure d'un programme en C++



## Type

2

- C++ est un langage typé : un élément manipulé dans un programme dispose d'un type indiquant les caractéristiques de l'élément et la façon de le manipuler (opérations que l'on peut effectuer sur ou avec l'élément).
- Avantages:
  - permet au compilateur de vérifier la validité des opérations qu'on appliquera à ou avec un élément;
  - contraint le programmeur à se restreindre aux seules opérations autorisées;
  - aide à la détection des erreurs.
- Un descripteur est un mot clé (`virtual`, `extern`, `static`,…) spécifiant la classe de stockage ou la portée d'une fonction ou d'une variable. Un descripteur ne fait pas partie du type.



## Déclarations et définitions

3

- Un élément peut être une variable, une fonction ou un type.
- On appelle déclaration le fait d'informer le compilateur qu'un élément existe et de lui associer un nom.
- On appelle définition le fait de demander au compilateur de créer un élément et de lui associer un nom. Une définition est aussi, de fait, une déclaration.
- Lorsqu'il s'agit de la définition d'une variable ou d'une fonction, de l'espace mémoire est réservé pour stocker l'élément défini.



## Définition et déclaration d'une variable

4

```
Type identificateur;
Type id1, id2, id3, …;
Type identificateur=expression;
Type id1, id2=v2, id3, id4, id5=e5;
Type identificateur(expression);
Type identificateur{expression}; // C++11
Type identificateur={expression}; // C++11
```

- Une variable ne doit être définie qu'une seule fois.
- Si on utilise le mot clé **`extern`** devant le type, il s'agit alors d'une simple déclaration de variable : celle-ci doit être définie dans une autre unité de traduction.
- Pour être utilisée, une variable doit avoir été préalablement déclarée.



## Initialisation des variables

5

- La définition d'une variable ne suffit pas (en général) à l'initialiser. Les variables non initialisées ont une valeur indéfinie.
- Initialiser les variables avec une valeur par défaut est donc une bonne habitude à prendre.
- On peut utiliser des expressions (qui seront évaluées) pour initialiser les variables.

```
Type identificateur=expression;
Type identificateur(expression);
Type identificateur{expression}; // C++11
Type identificateur={expression}; // C++11
```

- Les deux dernières formes d'initialisation avec {}, interdisent les conversions implicites de type dégradantes (ex : `double` vers `int`).



## Initialisation des variables

6

- Les différentes formes d'initialisation peuvent être combinées sur une même ligne :

```
Type id1, id2=e1, id3{e2}, id4, id5=e3;
```

- Les initialisations avec {} peuvent ne pas prendre d'expression, c'est alors l'initialisateur par défaut du type de la variable qui est utilisé :

```
int i{}; // int i=int(); int i=0;
int j={}; // int j=int(); int j=0;
```

- Cette dernière forme peut être aussi utilisée pour l'affectation d'une variable :

```
int i=4;
i={}; // affectation : i=0;
```

## Les mots clé `auto` et `decltype`

7

- Depuis C++11, le mot clé **`auto`** peut être utilisé au moment de la définition d'une variable initialisée pour laisser le compilateur en déterminer automatiquement le type :

```
auto identificateur=valeur_initiale;
auto identificateur{valeur_initiale};
```

- Cela permet d'alléger l'écriture et parfois d'obtenir un code plus sûr [Meyers, 2014] (notamment en évitant d'oublier une initialisation qui devient obligatoire).
- Attention, dans le cas de l'instruction « `auto id={val_init};` », `id` est du type `initializer_list`.
- On peut aussi utiliser le mot clé **`decltype`** pour utiliser le type d'une expression existante pour définir une autre variable :

```
auto var=36; // var est de type int
decltype(var) var2; // var2 est aussi de type int
```



## Définition d'une fonction

8

```
type_retour nom_fonction(type_1 arg_1, …,
                         type_n arg_n){
  /* instructions */
}
```

- Une définition de fonction est la description à l'aide d'instructions, de ce que fait la fonction.

Incompatibilités avec le C :

- La 2ème forme de définition du C n'est plus possible :

```
type_retour nom_fonction(arg_1,…,arg_n)
  type_1 arg_1; … type_n arg_n; {/*instructions*/}
```

- Le type de retour d'une fonction qui ne renvoie pas de valeur est toujours `void` (il ne peut pas être omis). De plus `void` ne doit plus être utilisé pour indiquer une fonction sans argument.
- Depuis C++14, on peut utiliser le mot clé `auto` comme type de retour d'une fonction. Le type sera déduit en fonction du type de l'expression retournée.



## Déclaration d'une fonction : prototype

9

```
type_retour nom_fonction(type_1 arg_1,…,type_n arg_n);
```

- Une déclaration de fonction est une simple information (nom, type de retour, type des arguments) fournie au compilateur.
- Pour être utilisée, une fonction doit avoir été préalablement déclarée.
- La portée du prototype est limitée à la partie du fichier source à la suite du prototype si elle est en dehors de tout bloc.
- On peut mettre un prototype dans un bloc : portée limitée au bloc.
- Les noms des variables dans les prototypes sont facultatifs et ne jouent aucun rôle à part informatif.
- Il n'y a pas de contrôle de cohérence des noms entre les différents prototypes et la définition de la fonction.
- Contrairement au C, on ne peut omettre ni le type de retour, ni les types des arguments.



## La fonction main

10

- Lorsqu'un programme est chargé, son exécution commence par l'appel d'une fonction spéciale du programme qui s'appelle « main ».
- Cette fonction marque le début du programme. La fonction main est appelée par le système d'exploitation : elle ne peut pas être récursive.

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

- La fonction `main` doit renvoyer une valeur de type `int` qui représente un code d'erreur d'exécution du programme.
- Elle peut aussi recevoir des paramètres du système d'exploitation.



## Structure d'un programme en C++

11

- Un programme en C++ est constitué d'un ou plusieurs fichiers source.
- Chacun de ces fichiers contient du code source : une suite de déclarations et de définitions de variables, de types, de fonctions, de classes.
- Leur extension est généralement « .cpp » (ou « .cc » ou « .cxx »).
- Dans un programme, exactement un fichier source contient la fonction globale d'entête « `int main()` » (avec en option des paramètres) qui sert de point d'entrée du programme.



## Fichiers d'entête

12

- Les déclarations de variables globales et de fonctions, ainsi que les déclarations/définitions de type, sont en général regroupées dans des fichiers séparés appelés des fichiers d'entête (header files). Leur extension est généralement « .h » (ou « .hh », « .hpp », « .hxx »).
- La directive `#include` du préprocesseur permet de développer le contenu d'un fichier d'entête dans un autre fichier.
- Il est préférable d'utiliser les directives du préprocesseur `#ifndef`, `#define`, `#endif` pour s'assurer qu'un fichier A ne soit développé qu'une fois dans un fichier B, même si A est inclus plusieurs fois dans B par transitivité (par ex B inclut C qui lui-même inclut A).

## Inclusion conditionnelle de fichier

13

**A.h**

```
#ifndef _A_H
#define _A_H
/* texte habituel du fichier */
#endif
```

**C.h**

```
#include "A.h"
//…
```

**B.h**

```
#include "A.h"
#include "C.h"
//…
```

A.h ne sera inclus qu'une seule fois car la constante _A_H aura déjà été définie lors de la 1ère inclusion. Lors de la deuxième inclusion, tout le texte situé entre « #ifndef _A_H » et « #endif » n'est pas inséré par le pré-compilateur



## Compilation d'un programme en C++

14

- Chaque fichier source est présenté séparément au compilateur. Il est alors prétraité par le préprocesseur (traitement des macros, inclusion de fichiers avec `#include`). Le prétraitement produit une unité de traduction.
- Pour chaque unité de traduction, le compilateur produit un fichier objet avec l'extension « .obj » (sur windows) ou « .o » (sur Unix et Mac OS X). Un fichier objet est un fichier binaire qui contient du code machine.
- Une fois que toutes les unités de traduction sont compilées, les fichiers objet sont combinés ensemble par l'éditeur de liens (linker).
- L'éditeur de liens concatène les fichiers objet et résout les adresses mémoire des fonctions et des autres symboles référencés dans les unités de compilation.



## Fichiers d'entête et compilation

15

- Les fichiers d'entête ne sont pas des fichiers source et ne donnent pas lieu à la création de fichier objet.
- Ils ne devraient contenir que des déclarations de variables et de fonctions ainsi que des définitions de type qui permettront aux différentes unités de compilation de communiquer entre elles.
- Il est inapproprié de mettre une définition de fonction ou de variable dans un fichier d'entête : différentes inclusions du fichier d'entête contenant cette définition dans plusieurs fichiers source mènent à une erreur lors de l'édition des liens à cause de la présence de plusieurs définitions du même symbole.
- Un type (structure, classe, modèle,…) peut être défini dans plusieurs unités de traduction (une fois au plus par unité) mais les définitions doivent être strictement identiques.
- Une fonction ou une variable déclarée, mais non définie, mène à une erreur de symbole non résolu, si elle est utilisée.



## Bibliothèques

16

- En pratique, l'éditeur de liens doit utiliser des symboles définis dans des bibliothèques.

Il y a 2 types de librairies :

- Librairies statiques : les éléments liés sont directement intégrés dans l'exécutable (comme les fichiers objets).
- Librairies dynamiques (librairies partagées, DLLs) sont localisées dans un endroit standard de la machine et sont automatiquement chargées au démarrage de l'application (liées dynamiquement).

1

# Les données en C++



## Notion de donnée

2

- Une donnée (on utilise aussi le terme « objet » au sens bas niveau) est une zone mémoire réservée d'une certaine taille que l'on peut manipuler.
- La taille d'une donnée est le nombre d'unités mémoires utilisées par cette donnée. Une unité mémoire est le bloc de bits le plus petit qui peut être réservé (alloué). En C++, le nombre de bits utilisés pour une unité mémoire est égale au nombre de bits pour coder une donnée de type `char` : 8 bits (généralement) ou plus.
- Une donnée est caractérisée par un type et une adresse.
- L'adresse d'une donnée est un nombre entier qui désigne la localisation de la première unité mémoire de la donnée.
- Le type permet d'interpréter (utiliser) la donnée et définit sa taille (le nombre d'unités mémoire prises par la donnée).
- Une donnée est accessible au travers du ou des identificateurs qui la « représente » directement ou indirectement.



## Variable / Identificateur

3

- Une variable est une donnée munie d'un identificateur au travers duquel on peut accéder directement à la donnée.
- Une variable est caractérisée par sa classe de mémorisation qui détermine sa visibilité (portée) et la durée de vie de la donnée associée à l'identificateur.



## Portée d'une variable

4

- Une variable n'est visible **qu'à partir de la ligne où elle a été déclarée**.
- Ensuite, la portée peut être
  - locale à un bloc
  - limitée à une unité de compilation
  - illimitée



## Variables locales

5

- Une variable déclarée à l'intérieur d'un bloc est visible dans tout ce bloc, et dans tous les blocs créés à l'intérieur de ce bloc.
- Elle n'est pas visible en dehors de ce bloc.
- Une variable définie à l'intérieur d'un bloc est dite **locale à ce bloc**.
- Un paramètre d'une fonction est une variable locale à cette fonction.



## Masquage de variables

6

- Une déclaration d'un identificateur dans un bloc peut en masquer une autre située dans le bloc conteneur.
- Il est possible de faire référence à un nom global masqué en utilisant « **::** ».
- Il n'existe aucun moyen de faire référence à un nom local masqué.

## Classes de mémorisation d'une variable

7

- variables automatiques (auto) : variables internes à un bloc, elles sont détruites lorsque l'on quitte le bloc.
  - variables de registres (register) : variables automatiques mémorisées (si possible) dans les registres rapides de la machine.
- variables statiques (static) : variables internes à un bloc, mais conservant leurs valeurs jusqu'au moment où l'on est de retour dans ce bloc.
- Les variables sont automatiques par défaut (si la classe de mémorisation n'est pas indiquée).
- variables externes à tout bloc (dites globales) : ces variables existent et conservent leur valeur pendant l'exécution du programme complet. Elles peuvent être utilisées pour communiquer entre deux fonctions, même dans le cas de fonctions compilées séparément.



## Données, mémoire et durée de vie

8

- Le C++ utilise 3 segments mémoire pour stocker les données.
  - Le segment statique (static), sur lequel les variables globales et statiques du programme sont chargées avant l'exécution du programme. Leur durée de vie est celle du programme.
  - La pile (stack) sur laquelle sont empilés les cadres des appels de fonction. Les variables locales non statiques des fonctions sont allouées et désallouées automatiquement sur ce segment (variables automatiques). Leur vie commence à leur définition et dure jusqu'à la fin du bloc auquel elle appartiennent.
  - Le tas (heap) sur lequel sont allouées les données dynamiques. Ces données sont allouées et désallouées explicitement par le programmeur. Leur durée de vie est donc contrôlée finement par le programmeur.



## Durée de vie d'une variable automatique

9

- Sauf indication, un objet défini dans une fonction est alloué au moment de sa définition et désalloué lorsque son nom sort de la portée courante.
- Il s'agit d'un objet automatique.
- S'il existe un initialisateur, la variable locale est initialisée lors de sa définition.
- L'allocation, l'éventuelle initialisation et la désallocation d'une telle variable se produisent lors de chaque appel de la fonction. Chacun de ces appels gère sa propre variable.



## Variables définies en dehors d'une fonction

10

- Une variable déclarée en dehors de toute fonction est visible dans tout le fichier où elle a été déclarée.
- Quand elle est définie en dehors de tout `namespace`, une telle variable est dite **globale**.
- Une variable définie sans initialisateur dans la portée globale ou d'un `namespace` est initialisée par défaut (au contraire des variables locales et des données créées sur le tas).
- Si une telle variable est précédée du mot clé `extern`, il s'agit d'une simple déclaration et non d'une définition. Cette déclaration ne doit pas avoir d'initialisateur. Une variable avec une exacte correspondance de type doit être définie dans une autre unité de traduction.

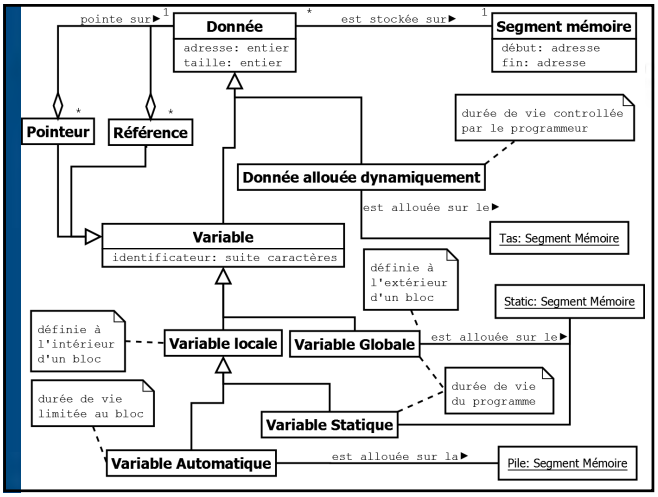


## Durée de vie d'une variable statique

11

- Si une variable locale est déclarée `static`, un unique objet, alloué sur le segment statique, sera représenté cette variable pour l'ensemble des appels de la fonction.
- Exactement une initialisation (même s'il n'existe pas d'initialisateur) aura lieu à la première exécution de la définition.
- Les objets déclarés dans une portée globale ou de namespace, ainsi que les variables statiques déclarées dans les fonctions ou les classes, sont créés et initialisés une seule fois et leur existence s'achève à la fin du programme.





## La taille des données

13

- La taille des données C++ s'exprime sous la forme de multiples de la taille d'une donnée de type `char`. Cette dernière est donc par définition de 1.
- `sizeof(T)` renvoie la taille d'une donnée de type `T`.
- `sizeof(e)` renvoie la taille de la valeur issue de l'expression `e`.



## Variables constantes : const

14

- Le mot clé `const` utilisé lors de la déclaration déclare une variable comme constante.
- Une constante ne peut pas être affectée, mais il est obligatoire de l'initialiser :

```
const double p=3.14*n; // où n est connue dans la portée
```

- La déclaration d'une variable avec `const` garantit que sa valeur ne sera pas changée dans les limites de sa portée.
- `const` fait partie du type de la variable : au lieu de spécifier la façon dont la constante doit être allouée, il restreint les possibilités d'utilisation d'un objet.
- Lorsque l'on utilise `const` avec une variable globale, sa portée est limitée au fichier source (on ne peut pas les utiliser dans un autre fichier source en utilisant `extern`) et ceci afin de pouvoir les utiliser à la place des `#define`.



## Notion d'expression

15

- Une expression est une instruction qui a une valeur et un type. Elle peut prendre une des formes suivantes :
  - `nomVariable // une variable est une expression`
  - `une littérale // une constante`
  - `operator expression /* un opérateur unaire suivi d'une expression */`
  - `expr1 operator expr2 /* 2 expressions séparées par un opérateur binaire */`
  - `expr1 ? expr2 : expr3 /* 3 expressions séparées par l'opérateur ternaire ?: */`
  - `nomFonction(arg1,...,argk); // un appel de fonction où arg1, …, argk sont des expressions`
- Une expression est évaluée afin d'en calculer la valeur.
- Toute expression C++ est soit une lvalue, soit une rvalue.



## lvalue

16

- Une lvalue est une expression qui permet d'accéder à une donnée.
  - Ce peut être un identificateur (accesseur direct) de la donnée elle-même dans le cas d'une variable.
  - Ce peut être une expression plus complexe (accesseur indirect) qui désigne la donnée au moyen d'un déréférencement d'une adresse : si `E` est une expression de type pointeur, `*E` est une lvalue qui permet d'accéder à la donnée pointée par `E`.
- Puisqu'une lvalue désigne une donnée en mémoire, elle peut toujours être précédée de l'opérateur `&` pour obtenir son adresse.
- Une lvalue fait référence à un objet qui persiste au-delà d'une expression.
- La signification de lvalue était à l'origine "quelque chose que l'on peut placer à gauche de l'opérateur d'affectation (`operator=`) (ce qui n'est plus vrai maintenant).
- Toutes les variables, y compris les variables non modifiables (**const**), sont des lvalues.



## rvalue

17

- Une rvalue est une valeur temporaire qui n'est pas persistante au-delà de l'expression dont elle est issue.
- Une rvalue ne peut apparaître qu'à droite de l'opérateur d'affectation `operator=`.
- On ne peut lui appliquer l'opérateur `&`.

```
int x=4, y=6; // x et y sont des lvalues
x=x*y; // x*y n'est pas une lvalue
x*y=8; // erreur
&(x*y); // erreur
```

1

# Les types du C++



## Les types fondamentaux

2

- `void` : utilisé pour spécifier le fait qu'il n'y a pas de type (ne peut être utilisé que comme sous ensemble d'un type plus compliqué (fonctions, pointeurs sur des données non typées, …);
- `bool` : les booléens ;
- `char` : les caractères ;
- `wchar_t` (nombre d'octets variable), `char16_t` (UTF-16) , `char32_t` (UTF-32) : les caractères longs ;
- `int` : les entiers ;
- `long int`, `long long int` : les entiers longs (`int` facultatif) ;
- `short int` : les entiers courts (`short` facultatif);
- le modificateur de type `unsigned` peut être appliqué aux types de base `char`, `short`, `int`, `long` et `long long` : utilisé lorsque des valeurs sont toujours positives. Lorsque le type de base est omis d'une déclaration, `int` est utilisé par défaut.
- `float` : les réels ;
- `double`, `long double` : réels en double ou quadruple précision.



## Les entiers

3

- Les types entiers ordinaires sont toujours **signés** par défaut.
- Les `int` précédés du mot clé `signed` sont des synonymes plus explicites. Par défaut une littérale entière est de type `int`.
- On peut cependant forcer une valeur littérale à être non signée ou à être un entier `long` ou `long long`, en la faisant suivre par le suffixe `u`, `l` ou `L` :
  - `34 //int -> au moins 16 bits`
  - `34u //unsigned int -> au moins 16 bits`
  - `34l //long int -> au moins 32 bits`
  - `34ul //unsigned long int -> au moins 32 bits`
  - `34L //long long int -> au moins 64 bits`
  - `34uL //unsigned long long int -> au moins 64 bits`
- Une littérale entière commençant par `0x` est un nombre hexadécimal.
- Une littérale entière commençant par `0` suivi d'un chiffre est un nombre octal.



## Les caractères

4

- Un caractère est un type entier (signé ou non signé suivant les implémentations).
- Une littérale caractère est une constante entière à laquelle on a associé un caractère selon une norme (ASCII) : la constante porte le nom du caractère encadré d'apostrophes :
  `'A', 'B', '*', '1', '@',…`
- En préférant une littérale caractère à la notation décimale, on obtient des programmes offrant une meilleure portabilité. En préfixant une littérale, nous pouvons modifier son type :

```
decltype('A') == char; decltype(L'A') == wchar_t;
decltype(u'A') == char16_t; decltype(U'A') == char32_t;
```

- Une littérale peut être stockée dans une donnée de type caractère :

```
char x1='A'; wchar_t x2=L'A';
char x3=u8'A'; //codé en UTF-8
char16_t x4=u'A'; //codé en UTF-16
char32_t x4=U'A'; //codé en UTF-32
```



## Les flottants

5

- Types flottants : `float`, `double` et `long double`.
- Une littérale virgule flottante est par défaut du type `double`.
- On peut cependant modifier le type d'une constante en ajoutant le suffixe `L` (pour long double) ou `f` (pour float) après la constante :
  - `3.14159L // long double`
  - `4.32e19f // float`



## Le type `bool`

6

- le langage C++ a introduit par rapport au langage C un nouveau type de donnée appelé `bool`. Ce type de variable accepte deux états :
  - `true` (vrai) : correspondant à la valeur 1
  - `false` (faux) : correspondant à la valeur 0

## La taille des données

7

- Une donnée de type `char` est constituée d'au moins 8 bits.
- Relations entre les tailles des types fondamentaux :
  - `1 == char <= short <= int <= long <= long long`
  - `1 <= bool <= long`
  - `char <= wchar_t <= long`
  - `float <= double <= long double`
  - `N == signed N == unsigned N`
- La bibliothèque `<limits>` connaît ces aspects pour une implémentation donnée et permet d'obtenir un certain nombre d'indications sur chaque type :

```
std::cout<<"max="<<numeric_limits<int>::max() <<"\n"
         <<"min="<<numeric_limits<int>::min() <<"\n";
```



## `size_t`

8

- Le type `size_t` est un alias d'un type entier non signé fondamental utilisé pour représenter la taille des objets.
- La valeur retournée par `sizeof()` est donc de type `size_t`.
- Ce type est largement utilisé par les bibliothèques standards pour représenter des tailles de conteneur ou pour indexer des tableaux.
- Sa taille dépend de l'implémentation. Le nombre de bits utilisés est toujours au moins de 16.



## Types construits par le programmeur

9

- Les types énumérés
- Les pointeurs et les références
- Les tableaux
- Les structures et les classes
- Synonymes de type



## Les types énumérés non délimités (enum)

10

- Une énumération est un type dans lequel il est possible de stocker un ensemble de valeurs spécifiées par le programmeur. De telles valeurs sont appelées énumérateurs : `enum { bleu, vert, jaune};`
- Une énumération a en général un identificateur :
  `enum Couleur {bleu, vert, jaune};`
- Chaque énumération est de type distinct. Le type d'un énumérateur est l'énumération auquel il appartient. Un énumérateur est convertible implicitement en `int`.
- Une variable de type énumération peut stocker des valeurs correspondant aux énumérateurs de cette énumération :
  `Couleur c1=bleu, c2=jaune;`
- Les identificateurs des énumérateurs sont dans la portée dans laquelle ils ont été déclarés. Aucun autre élément ne peut donc avoir cet identificateur dans la même portée.
- Une enum non délimitée est définie mais ne peut pas être déclarée.



## Les types énumérés délimités (`enum class`)

11

- C++11 propose l'énumération délimité dans laquelle les identificateurs des énumérateurs ont la portée de l'énumération :

```
enum class Couleur {bleu, vert, jaune}; // définition
Couleur c1=Couleur::bleu, c2=Couleur::jaune;
```

- Cela évite notamment de polluer l'espace de noms.
- Les énumérateurs sont plus fortement typés : les énumérateurs ne peuvent plus être implicitement en un autre type.
- Néanmoins, le type sous-jacent est toujours le type `int` par défaut. On peut spécifier un autre type en cas de besoin :

```
enum class Couleur : char {bleu, vert, jaune};
```

- Une énumération délimitée peut être déclarée de manière anticipée :

```
enum class Couleur; // déclaration
```



## Tableaux

12

- Un tableau est un agrégat d'éléments de même type.
- Pour un type `T`, `T[taille]` est le type tableau de `taille` éléments de type `T`.
- Les éléments sont indexés de `0` à `taille-1`.
- `taille` doit être une expression constante (une expression qui peut être évaluée à la compilation).
- Les tableaux à plusieurs dimensions sont représentés par des tableaux de tableaux.
- Il n'est pas possible de transmettre un tableau par valeur en argument ou en retour d'une fonction contrairement aux autres types.
- Un tableau peut être initialisé par une liste de valeurs entre accolades. La taille du tableau possédant une liste d'initialisation est calculée en fonction des éléments de cette dernière si la taille n'est pas précisée.

## Tableaux array (C++11)

13

- Le type paramétré `array<T,n>` permet de remplacer les tableaux natifs du C de type `T` et de taille `n`, en ajoutant notant des éléments de sécurisation.
- Tout comme les tableaux natifs, il possède une taille définie par une expression constante calculée à la compilation.
- Il peut être initialisé par une liste d'initialisation. Les éléments pour lesquels on ne fournie pas d'initialisateurs sont initialisés avec la valeur par défaut du type `T`.

```
array<int,10> tab1; /* tableau de 10 int initialisés avec 0. */
array<int,5> tab2{1,2,3}; /* tableau 5 int, les 3 premiers int sont initialisés avec 1,2,3, les 2 autres avec 0.*/
```

- La méthode `data()` renvoie un pointeur de type `T*` permettant de le traiter comme un tableau de type `T`.



## Boucles basées sur les intervalles (C++11)

14

- Les boucles « range-based `for` » permettent de parcourir facilement chacun des éléments d'une collection.
- Elles sont notamment utilisables avec les tableaux de type C, les `array`, et tout conteneur disposant des méthodes `begin()` et `end()` pour un parcours avec itérateurs.
- Les éléments du tableaux peuvent parcourus par valeur ou par référence (`const` ou non) selon les besoins.

```
array<int,10> tab1;
int tab2[]={1,2,3};
for(const int& e:tab1) std::cout<<e;
for(const int& e:tab2) std::cout<<e;
for(int& e:tab1) e=1;
for(int& e:tab2) e=0;
for(int e:tab1) std::cout<<e;
for(int e:tab2) std::cout<<e;
for(int e: {0,1,2,3,4}) std::cout<<e;
```



## Chaines de caractères

15

- Une chaine de caractères est un tableau de `char`.
- Par convention, la littérale `'\0'`, dont la valeur est `0`, est utilisée pour marquer la fin d'une chaine : tous les caractères du tableau situés après cette littérale sont ignorés (par les fonctions manipulant ces chaines).
- Les chaines de caractères suivent les mêmes conventions de déclaration, de définition et d'initialisations que les tableaux d'autres types :

```
char str[10]={'a','r','b','r','e','\0'};
```

- Une méthode pratique d'initialisation de chaîne consiste à utiliser une littérale chaîne de caractères :

```
char str[10]="arbre";
```



## Littérales chaines de caractères

16

- Une littérale chaîne est une séquence de caractères encadrée de guillemets (").
- Une littérale chaîne contient un caractère de plus qu'il n'y paraît : `'\0'` : `sizeof("hello")` renvoie 6.
- Le type d'une littérale chaîne est « tableau du nombre approprié de caractères `const` » : `"hello"` est donc du type `const char[6]`.
- La chaîne vide est représentée par une paire de guillemets adjacents `""` et est du type `const char[1]`.
- Dans les définitions précédentes du C et du C++, le type d'une littérale chaîne était `char*`. Il est donc possible d'attribuer une littérale chaîne à un type `char*` (permet de garantir la validité de millions de lignes de code). Il s'agit toutefois d'une erreur de vouloir modifier une littérale chaîne par l'intermédiaire d'un tel pointeur.

```
char* p= "hello";
p[4]='a'; // erreur : résultat indéfini
```



## Littérales chaines de caractères

17

- Le caractère constant des littérales chaîne permet aux implémentations d'optimiser de manière significative la façon dont les littérales de chaîne sont stockées et accédées.
- Si on veut modifier une chaîne, il faut en copier les caractères dans un tableau.

```
char p[]= "hello"; p[4]='a'; //ok
```

- Une littérale chaîne est allouée de façon statique. Il est donc correct d'en renvoyer une à partir d'une fonction.

```
const char* coucou(){
   return "bonjour !!!";
}
```

- La mémoire dans laquelle est stockée `"bonjour !!!"` n'est pas libérée après un appel de `coucou()`.
- Le fait que 2 littérales chaînes identiques soient allouées en tant que littérale chaîne unique ou non est défini par l'implémentation. Ainsi, elles peuvent ou non avoir la même adresse.
- Une chaîne ne peut pas contenir de réels retours à la ligne.
- Une chaîne possédant le préfixe `L` comme `L"hello"` est une chaîne de caractères larges. Son type est `const wchar_t[]`.



## Littérales chaines de caractères longs

18

```
"hello"; //tableau de char
L"hello"; //tableau de wchar_t
u8"hello"; //tableau de char, codés en UTF-8
u"hello"; //tableau de char16_t, codés en UTF-16
U"hello"; //tableau de char32_t, codés en UTF-32
```

Toutes ces littérales terminent par le caractère `0`.

## Structures

19

- Une structure est un agrégat d'éléments de types arbitraires.
- Déclaration d'un type structure :

```
struct id_struct {
  type1 id1;
  type2 id2;
  …
};
```

- Une structure `struct` est une forme simple de `class`.



## Structures

20

- Définition d'une variable structure :

```
id_struct obj;
```

Contrairement au C, la notation `struct` devant `id_struct` n'est plus requise.

- La notation employée pour l'initialisation des tableaux peut être aussi employée pour les structures (`{...}`) :

```
id_struct obj={val1, val2, …};
```

- Il est possible d'utiliser le nom d'une structure avant que ce dernier soit défini. Pour ce faire, il est indispensable de ne pas avoir à connaître le nom d'un membre ou la taille de la structure.



## Accès à un champ d'une structure

21

- Pour accéder à un champ d'une variable structure, on utilise l'opérateur « `.` » :

```
id_struct obj;
obj.id1=valx;
std::cout<<obj.id2<<"\n";
```

- Pour accéder à un champs d'une variable structure à partir d'un pointeur vers cette structure (voir « adressage indirect »), on utilise l'opérateur « `->` » :

```
id_struct* pt=&obj;
pt->id1=valx;
// equivalent à (*pt).id1=valx;
```



## Affectation et initialisation entre structures

22

- L'opérateur d'affectation est automatiquement défini pour un type structuré défini par l'utilisateur.
- Cet opérateur copie exactement les valeurs contenues dans les différents champs de la structure source vers la structure définition.
- De la même manière, on peut initialiser une structure avec une autre structure.



## Synonymes de type avec `typedef`

23

- Le mot clé `typedef` permet de créer des nouveaux types en utilisant et combinant des types existants :

```
typedef int entier;
typedef int* ptr_entier;
typedef int& ref_entier;
typedef int vecteur[3]; /* vecteur est un tableau de
                           3 int */
typedef personne individu; /* où personne est une
                              structure */
typedef individu* ptr_individu;
typedef individu couple[2]; /* couple est un tableau
                       de 2 invidus/personnes */
```

- Notons qu'il est impossible de faire des synonymes de patron avec `typedef`.



## Synonymes de type avec `using`

24

- Depuis C++11, il est préconisé d'utiliser des déclarations d'alias avec `using` plutôt que d'utiliser `typedef`.

```
using entier=int;
using ptr_entier=int*;
using ref_entier=int&;
using vecteur=int[3];
```

- En plus d'être plus lisibles en produisant un code généralement plus simple, ils permettent aussi l'alias de template :

```
template<class T> using vect=array<T,3>;
```

1

# Adressage indirect



## Donnée, identificateur, mémoire

2

- Une donnée (objet au sens bas niveau) est une région contiguë de mémoire. Une variable est une donnée munie d'un identificateur au travers duquel on peut accéder directement à la donnée.
- Une donnée a au plus un identificateur (nom de la variable la désignant) correspondant à une adresse mémoire.
- Or en programmation, il faut pouvoir manipuler une même donnée dans des contextes différents qui s'ignorent et où son identificateur n'est pas forcément visible (par ex. dans une autre fonction que celle dans laquelle elle est définie).
- Il faut pouvoir aussi accéder aux données sans identificateur (allouées dynamiquement).



## Adressage indirect

3

- Une donnée peut être désignée par plusieurs référents qui correspondent aussi à cette même adresse mémoire grâce au mécanisme d'adressage indirect.
- Adressage indirect : possibilité pour une donnée d'être accédée par l'intermédiaire d'une variable référente, contenant une adresse désignant l'emplacement mémoire de cette donnée.
- Plusieurs de ces variables référentes peuvent alors désigner la même donnée.
- Permet d'offrir plusieurs voies d'accès « faciles » à une même donnée en mémoire possédant ou non un identificateur.



## pointeurs et références

4

- En C et C++, « un pointeur » est une variable qui peut être utilisée comme référent d'une autre variable.
- Mais le C++ possède aussi un autre type qui facilite et sécurise l'adressage indirect : le type référence.
- Les pointeurs et les références stockent l'adresse d'une donnée. Les principales différences entre pointeurs et références sont :
  - la syntaxe (plus facile pour les références) ;
  - une référence doit être initialisée, ne peut pas être nulle et ne peut pas être réassignée (au contraire des pointeurs) ;
  - une référence est déréférencée automatiquement.



## Les pointeurs

5

- Pour un type T, `T*` est le type "pointeur de T". Une variable de ce type contient une adresse désignant une zone mémoire pouvant contenir un objet de type `T`.
- L'opération `&x` renvoie l'adresse d'une donnée désignée par l'expression `x` qui peut alors être stockée dans un pointeur.
- Un pointeur peut changer de valeur par réaffectation.
- L'opérateur d'indirection (ou de déréférencement) `*` sur un pointeur fait référence à l'objet vers lequel est dirigé le pointeur.
- Le pointeur nul est représenté par la littérale `nullptr` depuis C++11 (c'était la littérale 0 en C++98).
- Un pointeur peut être implicitement converti en un type `bool` : un pointeur non nul est converti en `true`, un pointeur nul est converti en `false`.



## Les pointeurs « const » et les pointeurs constants

6

- En préfixant la déclaration d'un pointeur avec `const`, c'est l'objet pointé qui n'est pas modifiable par l'intermédiaire du pointeur : `const T* pt;`
- Pour déclarer un pointeur constant, (i.e. pour faire en sorte que ce soit le pointeur qui soit non modifiable), on utilise la déclaration `*const` plutôt que `*` :

```
T* const pt;
```

- Attention à `const*` :

  « `T const* p;` » équivaut à « `const T* p;` »

## Les pointeurs « const » et les pointeurs constants

7

- On peut attribuer l'adresse d'une variable non constante à un pointeur `const`.
- En revanche, l'adresse d'une constante ne peut être attribuée à un pointeur sur lequel aucune restriction ne s'applique, car cette opération autoriserait la modification de la donnée.
- Il est possible de supprimer explicitement les restrictions sur un pointeur de constante à l'aide d'une conversion de type explicite (à utiliser avec précaution…).



## lvalue références

8

- Une référence fournit un autre mode d'adressage indirect plus sécurisé et plus facile à utiliser.
- La notation « X&» se lit « lvalue référence sur une donnée de type X ».
- Pour garantir la validité d'une référence (c'est à dire l'associer à une donnée), on doit l'initialiser à sa définition.
- La valeur d'une référence ne peut pas être modifiée après initialisation. C'est-à-dire qu'une référence est liée à la même zone mémoire (a priori une donnée) tout au long de sa vie.
- L'initialisateur d'un type `T&` doit être une `lvalue` de type `T`.
- Il est possible de définir des références sur pointeur.
- Il n'est pas possible de définir des pointeurs sur des références, ni définir des tableaux de références.



## Facilités d'écriture

9

- Une fois initialisée, une référence s'utilise directement comme si c'était la donnée référencée (de la même façon qu'un éventuel identificateur de cette donnée si elle correspond à une variable).
- Une référence n'est donc pas déréférençable explicitement …c'est le compilateur qui s'en charge implicitement car il n'y a pas d'ambiguïté.
- On utilise donc l'opérateur « . » plutôt que « -> » lorsque qu'une référence pointe sur une structure.
- Si x est une référence `&x` renvoie l'adresse de l'objet référencé et non l'adresse de la référence.



## Les références const

10

- En préfixant la déclaration d'une référence avec `const`, l'objet référencé n'est plus modifiable par l'intermédiaire de la référence : `const T& pt=une_lvalue;`
- Une référence est non modifiable par nature, elle est donc toujours constante.
- «`T const& ref=v`» équivaut à «`const T& ref=v`»
- L'initialisateur d'une variable de type `const T&` n'a pas besoin d'être une `lvalue` ou même d'appartenir au type `T` :
  - une conversion de type implicite en `T` est tout d'abord appliquée, si nécessaire;
  - la valeur obtenue est ensuite placée dans une variable temporaire de type `T`;
  - cette variable temporaire est alors utilisée comme valeur de l'initialisateur;
  - cette variable temporaire persiste jusqu'à la fin de la portée de la référence.



## Pointeurs vs. références

11

- Les références ont une syntaxe plus simple (déréférencement implicite) et leur utilisation est plus sécurisée (une référence est toujours initialisée avec l'adresse d'une *lvalue* normalement valide et ne peut pas changer de valeur).
- Les pointeurs sont moins sécurisés (pointeurs non initialisés, *etc*) mais offre une utilisation plus souple, indispensable dans beaucoup de cas (ils peuvent être réassignés n'importe quand et ils peuvent être nuls).
- Pour cette raison, les pointeurs prévalent souvent. Les références sont le plus souvent utilisées pour transmettre des arguments aux fonctions avec une écriture simplifiée.



## Pointeurs vs. références

12

- Les pointeurs et les références sont représentées de la même manière en mémoire et peuvent très souvent se substituer entre eux dans leur utilisation.
- Quand utiliser l'un ou l'autre ?

13

# Compléments sur les pointeurs

- Arithmétique des pointeurs
- Le type `void*`
- Tableaux et pointeurs



## Arithmétique des pointeurs

14

- Soustraction de pointeurs :
  - définie uniquement lorsque les deux pointeurs sont dirigés sur des éléments du même tableau (bien que le langage n'offre aucun moyen rapide de garantir cet état de fait).
  - lorsque l'on soustrait un pointeur à un autre, le résultat correspond au nombre d'éléments du tableau situés entre les deux pointeurs, c'est-à-dire un entier.
- Il est possible de soustraire ou d'ajouter un entier à un pointeur. Dans les deux cas le résultat est un pointeur.
- L'addition de pointeurs n'a aucun sens et n'est pas autorisé.



## Le type `void*`

15

- Un pointeur de tout type d'objet peut être attribué à une variable de type `void*`.
- Une variable de type `void*` peut servir à affecter une autre variable de type `void*`.
- Un `void*` peut être **explicitement** converti en un autre type.
- En C++, seule la conversion implicite de `void*` vers `int*` est autorisée. Dans les autres cas, il faut faire un cast.
- Attention : l'indirection `*` ne s'applique pas au type `void*`.
- On ne peut pas non plus incrémenter un `void*`.
- Il est généralement dangereux d'utiliser un pointeur converti en un type différent de celui de l'objet pointé.



## Tableaux et pointeurs

16

- L'identificateur d'une variable tableau est une constante qui est convertible **implicitement** dans une valeur de type pointeur égale à l'adresse du premier élément du tableau.
- Le type de cette valeur est alors "pointeur sur le type des éléments du tableau".

```
char txt[]="Message"; // variable tableau
char* msg=txt; // variable pointeur sur char
```

- L'identificateur du tableau `txt` est une constante de type `char[8]` (convertible implicitement en `char*`) alors que `msg` est une variable de type pointeur (`char*`). On peut écrire `msg=txt;` mais pas `txt=msg;`



## Tableaux et pointeurs

17

- La conversion implicite d'une valeur de type tableau vers une valeur de type pointeur est très utilisée dans les appels de fonction manipulant des tableaux.
- A la suite de la conversion, la taille du tableau évidemment est perdue pour la fonction appelée.
- Les tableaux sont étroitement liés aux pointeurs parce que, de manière interne, l'accès aux éléments des tableaux se fait par manipulation de leur adresse de base, de la taille des éléments et de leurs indices.
- L'adresse du n-ième élément d'un tableau est calculée avec la formule : `Ad_n=Ad_Base+n*sizeof(T)` où `Adresse_Base` est l'adresse de base du tableau. Cette adresse de base est à la fois l'adresse du tableau et l'adresse de son premier élément.



## Tableaux et pointeurs

18

- Afin de pouvoir utiliser l'arithmétique des pointeurs pour manipuler les éléments des tableaux, le C++ effectue la conversion implicite tableau vers pointeur d'élément.
- Si `x` est un identificateur de tableau ou de pointeur, les expressions `x[n]` et `*(x+n)` sont équivalentes.
- Les conversions implicites sont une facilité introduite par le compilateur.
- Mais en réalité, les tableaux ne sont pas des pointeurs.
- Ce sont des variables comme les autres, à ceci près :
  - leur type est convertible implicitement en pointeur sur le type de leurs éléments;
  - le passage par valeur d'un tableau n'est pas possible.

1

# Allocation et désallocation dynamiques en C++



## Allocation dynamique/désallocation en C++

2

- A la place des fonctions `malloc` et `free` du C, nouveaux opérateurs spécifiques : `new`, `delete`, `new[]` et `delete[]`.
- Les deux **opérateurs** `new` et `new[]` permettent d'allouer de la mémoire.
- Les deux **opérateurs** `delete` et `delete[]` permettent de restituer la mémoire utilisée.
- Le plus petit objet pouvant être alloué indépendamment, et pointé à l'aide d'un type de pointeur est un `char`.
- Lorsqu'il n'y a pas assez de mémoire disponible, les opérateurs `new` et `new[]` appellent un gestionnaire d'erreur : une exception `std::bad_alloc`.



## `new` et `delete`

3

- Syntaxe de `new` : le mot-clé new est suivi du type de la donnée à allouer. L'opérateur renvoie une valeur de type pointeur `T*` sur cette donnée.

```
T* id = new T;
```

Équivalent à `T* id=(T*)malloc(sizeof(T))` pour les types primitifs.

- Syntaxe de `delete` : faire suivre le mot-clé du pointeur sur la donnée à libérer.

```
delete id;
```

Équivalent à `free(id);` pour les types primitifs.



## `new[]` et `delete[]`

4

- Les opérateurs `new[]` et `delete[]` sont utilisés pour allouer et restituer la mémoire pour les types tableaux.
- Syntaxe :

```
T* id=new T[taille];
delete[] id;
```

- L'opérateur `new[]` alloue la mémoire et crée les objets dans l'ordre croissant des adresses.
- Inversement, l'opérateur `delete[]` détruit les objets du tableau dans l'ordre décroissant des adresses avant de libérer la mémoire.



## Attention aux mélanges…

5

- `new[] et delete[]` ne sont pas les mêmes opérateurs que `new` et `delete`. Utiliser l'opérateur `delete[]` avec les pointeurs renvoyés par l'opérateur `new[]` et l'opérateur `delete` avec les pointeurs renvoyés par `new`.
- Ne pas mélanger les mécanismes d'allocation mémoire du C et du C++.
- Il faut préférer les opérateurs C++ d'allocation et de désallocation de la mémoire aux fonctions `malloc` et `free` du C.
- Ces opérateurs ont de plus l'avantage de permettre un meilleur contrôle des types de données et d'éviter un transtypage.



## Variables de types primitifs *v.s.* instances de classes

6

- La manière dont les objets sont construits par les opérateurs `new` et `new[]` dépend de leur nature.
  - Données d'un type de base du langage ou structures simples : aucune initialisation particulière n'est faite. La valeur est donc indéfinie.
  - Instances de classes : le constructeur de ces classes sera automatiquement appelé lors de leur initialisation.
- Lors de la désallocation avec `delete` ou `delete[]` d'instances de classes, le destructeur de ces classes est appelé automatiquement.

1

# Fonctions

- Transmission d'arguments et d'une valeur de retour
- Arguments par défaut
- Surcharge (polymorphisme ad'hoc)
- Variable / fonction `inline`
- Variable / fonction `constexpr`



## Mode de transmission : le passage par valeur

2

- Les arguments d'une fonction sont toujours transmis « par valeur ».
- Cela signifie qu'un paramètre de fonction est une variable locale à la fonction qui sera initialisée avec le résultat de l'évaluation de l'expression passée en argument, c'est-à-dire avec une valeur.
- Il est de même pour le mode de transmission d'une valeur de retour d'une fonction : c'est une valeur résultat de l'évaluation de l'expression transmise à l'instruction `return`.
- Notons que c'est l'unique mode de transmission possible en C/C++.



## Passage par valeur et adressage indirect

3

- L'utilisation du passage par valeur ne permet pas de modifier des données dont la portée s'étend au contexte courant (données définies dans une autre fonction, un autre bloc, etc.).
- Pour cela, il faut passer par l'adressage indirect (pointeurs et références).
- On parle alors de passage par adresse ou par référence.
- Notons que ces deux modes de transmission sont aussi des passages par valeurs. Ces valeurs sont des adresses.



## Transmission par adresse

4

- Le passage par adresse consiste à transmettre une expression dont le résultat correspondant à son évaluation est une valeur de type pointeur correspondant à l'adresse d'un objet.
- Cette adresse peut alors être utilisée dans la fonction pour agir sur l'objet pointé (en utilisant un déréférencement explicite avec l'opérateur `*`).
- Le type du paramètre de la fonction est donc un « `T*` ».
- Si le paramètre est un pointeur `const`, (i.e. de type `const T*` ou `T const *`), la fonction n'est pas autorisée à modifier l'objet pointé (seule la lecture est autorisée).



## Transmission par référence

5

- Le passage par référence consiste à transmettre une expression dont le résultat correspondant à son évaluation est une `lvalue` correspondant à un objet.
- Cette référence peut alors être utilisée dans la fonction pour agir sur l'objet pointé (en utilisant implicitement un déréférencement pris en charge par le compilateur).
- Le type du paramètre de la fonction est donc un « `T&` ». Impose donc à un argument effectif d'être une `lvalue` de type `T`.
- Si le paramètre est une référence `const`, (i.e. de type `const T&` ou `T const &`), la fonction n'est pas autorisée à modifier l'objet pointé (seule la lecture est autorisée).



## Retour de valeur par adresse ou par référence

6

- Les types des valeurs transmises par l'instruction `return` en retour d'une fonction peuvent également être des types pointeur ou référence.
- Les adresses stockées dans ces pointeurs ou références peuvent pointer sur des objets alloués dynamiquement dans la fonction ou sur des objets existants dans d'autres fonctions.
- Il faut faire attention à ne pas renvoyer d'adresse sur des objets alloués automatiquement dans la fonction puisqu'ils sont désalloués automatiquement juste après le retour de valeur de la fonction.
- Lorsqu'une fonction renvoie une référence, il devient possible d'utiliser son appel comme une `lvalue`.

## Arguments par défaut

7

- On peut donner des valeurs par défaut aux arguments des fonctions. L'utilisateur peut alors ne pas donner de valeur pour ces variables.
- Les valeurs par défaut sont fixées dans la déclaration de la fonction et non dans sa définition.
- Lorsqu'une déclaration prévoit des valeurs par défaut, les arguments concernés doivent obligatoirement être les derniers de la liste.
- Les valeurs par défaut ne sont pas forcément des expressions constantes. Elles ne peuvent toutefois pas faire intervenir de variables locales à la fonction.



## Surcharge de fonction : « *polymorphisme ad hoc* »

8

- Il est possible d'écrire des fonctions ayant des noms identiques mais effectuant des actions différentes.
- On parle de **surdéfinition** (**overloading**, **surcharge**) lorsqu'un même symbole possède plusieurs significations différentes.
- Les fonctions de même nom doivent différer par leurs listes de paramètre. Les types des paramètres ne peuvent pas différer uniquement que par des `const` (provoque une ambiguïté) sauf si ce sont des pointeurs ou des références.
- Les fonctions surchargées ne peuvent différer uniquement par leur type de retour.



## Surcharge de fonction : mécanisme

9

- Lors de l'appel, le choix de la fonction est fait par le compilateur suivant le nombre et le type d'arguments de la fonction.
- Principes :
  - Le compilateur cherche une correspondance exacte entre paramètres réels et paramètres formels.
  - Si la mise en correspondance exacte échoue, le compilateur tente à nouveau une mise en correspondance en effectuant des conversions autorisées.



## Appels de fonction

10

- A chaque appel d'une fonction, il y a mise en place des instructions nécessaires pour établir la liaison entre le programme et la fonction :
  - sauvegarde de "l'état courant"
  - recopie des valeurs des arguments
  - branchement avec conservation de l'adresse de retour
  - recopie de la valeur de retour
  - restauration de l'état courant
  - retour dans le programme



## Fonctions et variables « en ligne » : `inline`

11

- Une fonction ou une variable en ligne se définit de manière ordinaire en faisant précéder sa définition de la spécification `inline`.
- La signification de ce mot a changé en C++17.
- La notion de fonction `inline` existe depuis les débuts du C++.
- La notion de variable `inline` n'existe que depuis C++17.



## Fonctions `inline` : signification jusque C++17

12

- À chaque appel d'une fonction `inline`, le compilateur incorpore les instructions de la fonction à la place de l'appel.
- Permet un gain de temps. Cependant, les instructions sont générées pour chaque appel : cela consomme une quantité de mémoire du programme proportionnelle au nombre d'appels.
- Le compromis est difficile à trouver. Cependant, une règle simple est de réserver les fonctions `inline` aux fonctions courtes.
- Quand on utilise `inline`, c'est le compilateur qui fait l'incorporation des instructions (en langage machine). Pour les macros, c'est le préprocesseur.
- Une fonction en ligne doit être définie dans le même fichier source que celui où on l'utilise.
- Elle ne peut être compilée séparément. Pour qu'une même fonction en ligne puisse être partagée par différents programmes, il faudra la placer dans un fichier entête.
- Le mot clé `inline` est considéré comme une « requête » que le compilateur peut ignorer (par exemple, si on utilise quelque part l'adresse de la fonction, elle ne peut plus être `inline`).

## Fonctions et variable `inline` : signification à partir de C++17

13

- Le mot clé `inline` n'est plus une « requête » pour indiquer la préférence d'une substitution d'un appel de fonction. En effet, le compilateur peut substituer un appel de fonction même si elle n'est pas `inline` et peut toujours ne pas substituer l'appel d'une méthode `inline`. Les règles d'optimisation de substitution d'un appel deviennent indépendantes du mot clé.
- Le mot clé est utilisé pour permettre la multiple définition dans le programme d'une fonction ou d'une variable non statiques mais avec des contraintes :
  - Exactement une définition de la fonction/variable inline doit être disponible dans chaque unité de traduction où on l'utilise.
  - Toutes les définitions dans les différentes unités de traduction doivent être identiques et elles doivent être inline dans chacune des unités de traduction.
  - Chaque définition à la même adresse dans toutes les unité de traduction.



## Variables `constexpr` (C++11)

14

- Le mot clé `constexpr` utilisé lors de la déclaration déclare une variable constante dont la valeur peut être calculée à la compilation.
- Toute variable définie avec `constexpr` est une constante. L'inverse n'est pas toujours vraie :

```
void f(int n){
        constexpr auto x=10; // ok
        constexpr auto y=10*n; /* erreur x ne peut pas être
                calculée au moment de la compilation */
        const auto y=10*n; // ok
}
```

- Ce mot clé est essentiellement utilisé dans des contextes exigeant des constantes connues à la compilation (par ex. pour définir un tableau d'une taille donnée),
- Une telle donnée peut être placée dans une zone de mémoire en lecture seule (important dans les systèmes embarqués) [Meyers, 2014]



## Fonctions `constexpr` [Meyers, 2014]

15

- Quand on utilise `constexpr` devant le prototype d'une fonction, on indique au compilateur que cette fonction peut produire des constantes connues à la compilation lorsqu'elle est utilisée avec des constantes connues à la compilation :
  - Si toutes les valeurs des arguments transmis à la fonction sont connues à la compilation, le résultat sera déterminé pendant la compilation.
  - Si au moins un argument transmis à la fonction n'est pas connue au moment de la compilation, elle se comporte comme une fonction normale.
- Certains calculs traditionnellement effectuées à l'exécution peuvent maintenant être réalisés à la compilation (l'exécution du programme sera plus rapide).
- Depuis C++11, une fonction/variable `constexpr` est implicitement `inline`.



## Fonctions `constexpr` [Meyers, 2014]

16

```
constexpr int puissance(int x, int n) {
     /*…*/
}

int f(const int y){
     constexpr auto x=5;
     array<int,puissance(y,2)> t1; // erreur
     array<int,puissance(x,2)> t2; // ok
}
```

- En C++11, les fonctions `constexpr` ne peuvent contenir qu'une seule instruction exécutable (un `return`). Depuis C++14, elles peuvent contenir plusieurs lignes de code.
- Les fonctions `constexpr` ont pour obligations de prendre et de retourner des littéraux, i.e. des types dont il est possible de déterminer la valeur au moment de la compilation. Attention, en C++11 `void` n'est pas considéré comme un littéral (en C++14 oui).
- Il est recommandé d'utiliser `constexpr` dès que possible. Mais cela nécessite d'accepter sur le long terme les contraintes que cela impose.

1

# Espaces de noms (namespaces)



## Espaces de noms

2

- Les espaces de nommage sont des zones de déclaration qui permettent de délimiter la recherche des noms des identificateurs par le compilateur.
- buts :
  - regrouper logiquement des identificateurs
  - éviter de définir des objets dans la portée globale.
  - éviter des conflits de noms entre plusieurs parties d'un même projet.



## Espaces de noms

3

```
namespace identificateur {
  /* définitions et déclarations */
}
```

- Permet de définir des ensembles disjoints d'identificateurs.
- Chaque ensemble est repéré par son nom utilisé pour qualifier les symboles concernés.
- Il est possible d'utiliser le même identificateur pour désigner 2 choses différentes si elles appartiennent à deux espaces de noms différents.



## Résolution de portée

4

- Pour se référer à des identificateurs définis dans un espace de noms à l'extérieur de l'espace de noms, on utilise l'opérateur de résolution de portée `::` .
- La qualification est inutile à l'intérieur de l'espace de nommage lui-même.

```
int i=1;    // i est global.
namespace A
{
    int i=2; // i de l'espace de nommage A.
    int j=i; // Utilise A::i.
}
int main(void)
{
    i=1;     // Utilise ::i.
    A::i=3;  // Utilise A::i.
    return 0;
}
```



## Directive Using

5

- Pour éviter l'utilisation systématique de l'opérateur de résolution de portée pour se référer à des identificateurs définis dans cet espace de noms, on utilisera une directive using :

```
using namespace une_bibli;
/* à partir d'ici, les identificateurs de
  une_bibli sont connus */
```

- On peut lever les ambiguïtés qui apparaissent lorsqu'on utilise plusieurs espaces de noms comportant des identificateurs identiquesen utilisant l'opérateur de résolution de portée `::`.



## Déclarations, namespace, portées

6

- Une instruction `using` ne peut apparaître que si l'espace de noms auquel elle fait référence a déjà été déclaré.
- Les portées locales ordinaires, les portées globales et les classes sont des espaces de noms.
- Un espace de noms représente une portée. Les règles de portées habituelles s'y appliquent donc.
- Les identificateurs déclarés ou définis à l'intérieur d'un même espace de nommage ne doivent pas entrer en conflit.

## Ajouter des éléments dans un namespace

7

- Un espace de noms est ouvert : il est possible de lui ajouter des noms à partir de plusieurs déclarations de cet espace de noms.
- Ces différentes ouvertures (et fermetures) du même namespace peuvent avoir lieu dans des fichiers différents.

**fichierA.h**

```
namespace EN {
   void f();
}
```

**fichierB.h**

```
namespace EN {
   void g();
   void h();
}
```



## L'espace de noms `std`

8

- Tous les identificateurs des fichiers en-tête standards sont définis dans l'espace de noms `std`.
- l'instruction:

```
using namespace std;
```

est souvent utilisée…

- Toutes les bibliothèques standards du C sont disponibles dans l'espace de nommage `std`.
- Les fichiers d'entêtes correspondants sont les mêmes qu'avant mais précédés d'un c et sans l'extension `.h`.

ex : `stdio.h` devient `cstdio`



## Compléments sur les namespaces (utilisation avancée)

9



## Déclaration Using

10

- On peut aussi utiliser une déclaration using pour faire un choix permanent :

```
namepace A{
  int i; // Déclare A::i
}
void f(void){
    using A::i; // A::i peut être utilisé sous le nom i
    i=1;        // équivalent à A::i=1
}
```

- Les `using`-déclarations permettent en fait de déclarer des alias des identificateurs.
- Ces alias doivent être considérés exactement comme des déclarations normales et sont donc donc soumis aux mêmes contraintes.



## Alias de namespaces

11

- On peut faire un alias (synonyme) d'un namespace :

```
namespace B=A; //B est synonyme de A.
```



## Imbriquer des namespaces

12

- Les définitions d'espace de noms peuvent s'imbriquer. Cette déclaration doit avoir lieu au niveau déclaratif le plus externe de l'espace de nommage qui contient le sous-espace de nommage.

```
namespace Conteneur{
    int i; // Conteneur::i.
    namespace Contenu
    {
        int j; // Conteneur::Contenu::j.
    }
}
```

- Il n'est pas possible d'effectuer une déclaration de namespace au sein d'une classe ou d'une fonction.

# Concepts et programmation orientés objet

1

# Notions d'objet et de classe



## « Objet ? »

2

> Objet [Le Petit Robert]

[concret] Toute chose qui affecte les sens. Chose solide ayant unité et indépendance et répondant à une certaine destination.

[abstrait] Tout ce qui se présente à la pensée, qui est occasion ou matière pour l'activité de l'esprit. Ce qui est donné par l'expérience, qui existe indépendamment de l'esprit par opposition au sujet qui pense.

[informatique] Mode de représentation structuré permettant de décrire un élément par ses caractéristiques, ses propriétés et par ses relations avec les autres objets.



## Perception des objets

3

- Notre environnement est complexe.
- Cependant, nous pouvons distinguer des « objets » que nous percevons grâce à leurs attributs (couleur, taille, âge, matériau, mobilité,…) et grâce aux interactions que ces objets exercent entre eux.
- Activité de modélisation (abstraction) : nous sommes attentifs à un sous-ensemble de la caractéristique d'un objet.



## L'objet [Muller et Gaertner, 2003]

4

- Les objets informatiques définissent une représentation abstraite des entités d'un monde réel ou virtuel, dans le but de les piloter ou de les simuler.
- Ils encapsulent une partie de la connaissance du monde dans lequel ils évoluent.
- L'objet est une unité atomique formée de l'union d'un état et d'un comportement. Il présente alors les 3 caractéristiques :
  - son état : valeurs instantanées de tous ses attributs
  - son comportement qui regroupe ses compétences et décrit les actions et les réactions de cet objet.
  - son identité qui caractérise son existence propre en le distinguant des autres objets de façon non ambiguë, et cela, indépendamment de son état.



## Attributs et Opérations

5

- Un attribut est une information qui qualifie un objet et qui peut prendre une valeur dans un domaine de définition donné
  - Les attributs ne sont pas les objets : ils servent à caractériser les objets.
  - La nature des attributs est telle qu'ils se retrouvent attribut d'un nombre important d'objets.
  - L'entité (l'objet) est perçu car il est au croisement de différents attributs. Les valeurs assignées aux différents attributs permettent de personnaliser un objet par rapport aux autres.
- Chaque atome de comportement est appelé opération.
  - Les opérations d'un objet sont déclenchées suite à une stimulation externe, représentée sous la forme d'un message envoyé par un autre objet.
- L'état et le comportement sont liés : le comportement à un instant donné dépend de l'état, et peut modifier l'état.



## Les classes

6

- Pour réduire la complexité du monde, l'humain a appris à regrouper les éléments qui se ressemblent et à distinguer des structures générales (en éliminant des détails inutiles) : identification des caractéristiques communes à un ensemble d'éléments.
- Les objets qui ont les mêmes attributs et les même opérations en catégories que nous appelons des classes.
- La classe décrit le domaine de définition d'un ensemble d'objets.
- La description générale des caractéristiques (attributs et opérations) est contenue dans la classe et chaque objet de la classe à ses propres valeurs pour chacun des attributs.
- [Muller et Gaertner, 2003] Une classe donne une description abstraite d'un ensemble d'objets qui partagent des caractéristiques communes. Ces caractéristiques constituent la propriété caractéristique de l'ensembles des instances.

1

# L'approche orientée objet



## L'Approche objet

2

- Basée sur l'utilisation d'objets qui représentent (ce sont des abstractions) des concepts, des idées ou toute entité du monde réel.
- A pour but une modélisation d'un environnement.
- Considère un système comme un ensemble organisés d'éléments (les objets) qui se définissent les uns par rapport aux autres.
- Méthode de décomposition basée sur ce que le système est et fait (et pas seulement sur ce que le système fait).



## Les différents niveaux de l'approche objet

3

- L'approche orientée objet prend en compte le cycle de vie complet d'un logiciel :
  - L'analyse orientée objet : spécification d'un problème en utilisant une formalisation en terme d'objets.
  - La conception orientée objet : proposition d'une solution spécifiée en terme d'objets.
  - L'implémentation (et maintenance) orientée objet : codage d'une solution en programmant avec des objets :
    - Les langages orientés objet permettent de décrire et manipuler des classes (des modèles) et leur instances.
    - La programmation orientée objet est un paradigme de programmation informatique qui consiste en la définition et l'assemblage de modules logiciels qui sont des objets.



## Avantages de l'approche OO

4

- Stabilité (continuité) des développements, en restreignant au maximum l'impact des modifications apportées au code source au cours du temps : impacts limités aux seuls objets qu'ils concernent (encapsulation).
- Favorise la réutilisation des codes déjà existants en proposant un découpage modulaire.
- Favorise l'extension des codes déjà existants en proposant une modélisation hiérarchique des problèmes (héritage).
- Complémente la programmation procédurale en lui superposant un système de découpe modulaire plus naturel et facile à mettre en œuvre.



## L'analyse et la conception orientées objet

5

- Affronter la complexité d'un problème en le découpant naturellement et intuitivement en parties plus simples.
- S'inspire de notre manière cognitive de découper la réalité qui nous entoure.
- Découpage en classes dont les instances se délèguent mutuellement un ensemble de services.
- Découpage vertical des classes qui héritent entre elles d'attributs et de méthodes existants à différents niveaux d'une hiérarchie se basant sur la classification et la spécialisation des objets.



## Un principe architectural

6

- C'est toute une manière de concevoir un programme et la répartition de ses parties fonctionnelles qui est en jeu. Le but est l'harmonie.
- Les fonctions et les données ne sont plus d'un seul tenant mais éclatées en un ensemble de modules (classes) reprenant chacun :
  - une sous-partie de ces données
  - et les seules fonctions qui les manipulent
- Principe de développement de modules dont le couplage est réduit au minimum :
  agir localement, penser globalement

## Architecture et réutilisabilité

7

- Construire l'architecture d'un logiciel orienté objet est difficile.
- Construire cette architecture de manière à ce que ses éléments soient les plus réutilisables possible est encore plus dur :
  - Il faut trouver les objets pertinents à factoriser afin de les représenter sous forme de classes de bonne granularité (pas trop spécifiques mais répondant au problème…) ;
  - Construire leur interface;
  - Etablir la hiérarchie et les liens entre ces classes.



## Architecture et réutilisabilité

8

- L'architecture construite doit être spécifique au problème à résoudre mais aussi suffisamment générale pour faciliter la résolution de futures problèmes.
- Obtenir une architecture flexible et réutilisable est très difficile.
- La conception orientée objet est avant tout une question d'expertise : les développeurs expérimentés réutilisent souvent les bonnes solutions qu'ils ont développés.
- C'est pourquoi on retrouve des architectures récurrentes dans beaucoup de systèmes.



## SOLID

9

- Les bonnes architectures suivent généralement les 5 principes de conception représentés par l'acronyme SOLID :
  - **S** (Single responsability) : une classe ou une fonction ne devrait avoir qu'une seule responsabilité/fonctionnalité.
  - **O** (Open/closed) : un module devrait être ouvert à l'exention mais fermé à la modification.
  - **L** (Liskov substitution) : toute instance d'une classe de base devrait pouvoir être substituée par une instance d'une classe dérivée en gardant le bon comportement.
  - **I** (Interface segregation) : il vaut mieux plusieurs petites interfaces spécifiques réduites aux besoins de plusieurs clients qu'une interface générale qui tente de couvrir tous les besoins de tous les clients.
  - **D** (Dependency inversion) : il faut dépendre des abstractions et non des implémentations.



## Des outils pour la représentation et la conception

10

- UML (Unified Modeling Language) : langage graphique de modélisation des données et des traitements.
  - Formalisation aboutie et non-propriétaire de la modélisation objet utilisée en génie logiciel.
  - Standard défini par l'Object Management Group (OMG).
- Les design patterns :
  - Transposition de la pratique des design patterns architecturaux (bâtiment) dans l'univers du logiciel.
  - Le but est de capitaliser l'expérience dans le domaine de la conception (architecturale) de logiciels orientés objet dans des « patrons de conceptions ».



## UML

11

- UML propose 13 types de diagrammes (dernière version = 2.4.1).
- UML n'étant pas une méthode, son utilisation est laissée à l'appréciation de chacun.
- UML se décompose en plusieurs sous-ensembles
  - Les vues : les observables du système. Décrivent le système d'un point de vue qui peut être organisationnel, dynamique, temporel, architectural, géographique, logique, etc. En combinant toutes ces vues il est possible de définir (ou retrouver) le système complet.
  - Les diagrammes : sont des éléments graphiques. Décrivent le contenu des vues, qui sont des notions abstraites. Les diagrammes peuvent faire partie de plusieurs vues.
  - Les modèles d'élément : sont les briques des diagrammes UML, ces modèles sont utilisés dans plusieurs types de diagramme.
- Le **diagramme de classe** est généralement considéré comme l'élément central d'UML.



## Design patterns

12

- Un design pattern est une solution de conception commune à un problème récurrent dans un contexte donné.
- L'idée est la réutilisation d'une solution éprouvée à une problématique souvent rencontrée.
- Un design pattern propose une solution sous la forme d'un ensemble de classes.
- Un design pattern ne propose pas de code contenant la solution mais un plan de résolution exprimé dans un langage graphique de modélisation (UML).
- Permet de proposer les briques structurelles d'une solution élégante.

1

# Classes (implémentation et représentation)



## Classes et Objets [Meyer, 2008]

2

- Un type abstrait de données (TAD) est un ensemble d'objets définis par la liste des opérations, ou caractéristiques, qui s'appliquent à ces objets, ainsi que les propriétés de ces opérations (indépendamment d'une implémentation).
- Une classe est un type abstrait de données munie d'une implémentation.
- Les objets sont construits à partir d'une classe par un processus appelé instanciation. Une classe est un modèle (d'implémentation d'un TAD) et un objet est une instance d'un tel modèle.
- Une classe est un texte logiciel : elle est statique et existe indépendamment de toute exécution.
- Un objet instancié d'une classe est une structure de données créé dynamiquement qui existe seulement dans la mémoire d'un calculateur durant l'exécution d'un programme.



## Classes [Muller et Gaertner, 2003]

3

- Une classe donne une description abstraite d'un ensemble d'objets qui partagent des caractéristiques communes.
- Dans la plupart des langages orientés objet, la classe est réalisée directement par une construction syntaxique qui englobe les notions de type, de description et de module.



## Classe (description)

4

- Pour décrire une classe, on doit pouvoir décrire :
  - la nature structurelle des objets de la classe : de quoi il est fait, composé (attributs) C'est ce qui en particulier va déterminer la place mémoire que va occuper un objet en tant que donnée (implémentation du TAD).
  - ce que l'on peut faire avec l'objet : la liste des opérations possibles (spécifications->TAD, type), comment on fait ces opérations (implémentation du TAD).
- Une classe X se décrit par :
  - un ensemble d'attributs (la structure de donnée);
  - un ensemble de méthodes (les opérations).



## Attributs

5

- Les attributs représentent les propriétés des objets de la classe.
- Les informaticiens ont admis un ensemble de « types primitifs » d'attribut (dont on connait la taille mémoire pour coder une valeur).
- Un attribut peut être ou peut représenter un autre objet avec lesquels l'objet principal collabore : ce dernier peut alors déléguer certaines tâches à l'objet représenté par l'attribut.
- Chaque objet possède ses propres attributs.



## Méthodes et communication entre objets

6

- Une méthode est un regroupement d'instructions (c'est une fonction) qui s'exécute toujours sur un objet.
- Un objet est communiquant au travers de ses méthodes.
- Les objets collaborent entre eux en s'envoyant des messages qui consistent à utiliser les méthodes.

## Définition d'une classe en C++

7

- Une **définition de classe** regroupe entre accolades `{}` différents membres :
  - des données membre (attributs) : de types prédéfinis ou de classes,
  - des fonctions membre (méthodes).
- La définition de la classe se termine par un `;` (point virgule).

```
struct NomDeLaClasse {
  int x; // attribut
  void f(); // méthode
};
```



## Des attributs et des méthodes…

8

- Le type `struct` du C qui a été généralisé en intégrant les concepts de la POO.
- Les attributs (variables et constants) se déclarent de la même manière que pour le C.
- La déclaration d'une méthode est un prototype de fonction :

```
type de retour + identificateur
+ liste d'arguments entre parenthèses
   (type1 [+ id1], …, type2 [+ id2])
[+ modificateur]
```

- Un modificateur est un mot clé (comme `const` ou `final`) donnant une propriété particulière à la méthode.



## Classe (type) et Instances de classe

9

- Une fois qu'une classe a été définie, on peut créer des instances de classes de la même manière que des variables d'un type structuré :

```
NomDeLaClasse instClasse;
NomDeLaClasse* ptInstClasse=&instClasse;
```

- Un membre est accessible directement via n'importe quelle instance de classe suivant la syntaxe :

```
instClasse.nomAttribut
instClasse.nomMéthode(paramètres)
ptInstClasse->nomAttribut
ptInstClasse->nomMéthode(paramètres)
```



## Définition des méthodes

10

La définition d'une méthode peut se faire :

- à l'intérieur de la définition de la classe

  le compilateur tente alors d'en faire une méthode `inline` (voir le chapitre sur les fonctions `inline`). Remarque : le compilateur peut ignorer cette « requête ». Il est conseillé d'utiliser cela pour les méthodes courtes.

- à l'extérieur de la définition de la classe en utilisant l'opérateur `::` précédé du nom de la classe (la classe définissant son propre espace de nom).

  Cette définition doit se trouver dans une unité de compilation unique (`.cpp`).



## Méthodes `inline`

11

Pour rendre une méthode `inline`, on peut :

- soit fournir directement sa définition dans la déclaration même de la classe. Dans ce cas le qualificatif `inline` n'a pas à être utilisé.
- soit procéder comme une fonction ordinaire en fournissant une définition en dehors de la déclaration de la classe.
  - Le qualificatif `inline` doit alors apparaître à la fois devant la déclaration et dans sa définition.
  - Les définitions de ces fonctions seront fournies à la suite de la déclaration de la classe, dans le même fichier d'entête (`.h`).



## Argument implicite

12

- Une méthode comporte toujours un argument qui n'apparaît pas dans la liste des arguments de l'entête : on parle alors d'argument implicite.
- Il s'agit de l'objet sur lequel s'applique la fonction. Tout attribut qui apparaît à l'intérieur de la définition correspond à celle de cet objet.
- L'expression `this` désigne l'adresse de cet argument implicite.
- Ainsi, si d est un attribut de la classe, d et `this->d` sont équivalents dans la définition d'une méthode.

## Opérations et méthodes

13

- Les méthodes constituent les implémentations du concept d'opération dans les types de données abstraits.
- [Meyer, 2008] Pour les TDA, on distingue plusieurs catégories d'opérations :
  - Commande : opération qui peut modifier un objet.
  - Requête : opération qui renvoie des informations sur un objet (sans le modifier).
  - Créateurs (opération de créations) : opération qui produit des instances
- D'un point de vue implémentation, on distingue aussi généralement les accesseurs (en lecture et en écriture) qui permettent d'accéder aux propriétés.



## Méthodes const

14

- Le programmeur doit préciser, parmi les fonctions membres, lesquelles sont autorisées à opérer sur des objets constants (ou considérés comme constants par l'intermédiaire d'un référent pointeur ou référence).
- Le mot clé `const` est utilisé dans leur déclaration.
- Le modificateur `const` s'applique à l'argument implicite.
- L'argument implicite est alors considéré comme constant.
- Les instructions figurant dans la définition de la méthode ne doivent pas modifier la valeur des attributs de l'objet.
- L'implémentation des requêtes devraient être des méthodes `const`.



## Méthodes const

15

- Il est possible de surcharger une méthode en se fondant sur la présence ou l'absence du qualificatif `const` :

```
struct A {
  void f(); // utilisée par les objets non constants
  void f() const; /* utilisée par les objets
    constants (ou considérés comme constants par
    l'intermédiaire d'une référence const) */
};
```

- Une méthode `const` peut être appliquée à n'importe quel objet constant ou non.
- Le qualificatif `mutable` est utilisé pour désigner les attributs que l'on veut pouvoir modifier (avec une méthode) même lorsque l'objet est constant.



## Les objets et le partage

16

- Chaque objet dispose en propre de chacun de ses attributs.
- Les méthodes ne sont générées qu'une seule fois. La plupart des éditeurs de liens n'introduisent que les fonctions réellement utilisées.



## Portée d'une classe

17

- La plupart du temps, les classes sont déclarées à un niveau global ou dans un `namespace`.
- Il est permis de définir des classes à l'intérieur d'une autre classe :

```
struct A {
   struct B { /*…*/ };
};
```

- Le nom complet pour utiliser cette classe est alors `A::B` (la classe `A` est un espace de nommage).
- Il est permis de déclarer/définir des classes locales à une fonction. Dans ce cas, leur portée est limitée à cette fonction.



## Une classe = une responsabilité

18

- Une bonne conception implique d'une classe ne devrait avec qu'un seul type de responsabilité (c'est-à-dire une seule fonctionnalité).
- Cela permet de limiter les dépendances de la classe par rapport à ses clients.
- Il s'agit du S (Single responsability) de l'acronyme SOLID.

## Une classe en UML

19

- Une classe est représentée par un rectangle séparée en trois parties :
  - la première partie contient le nom de la classe ;
  - la seconde contient les attributs de la classe ;
  - la dernière contient les méthodes de la classe.

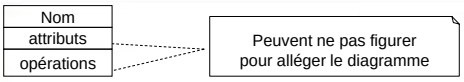
| Enseignant |
|---|
| age: entier<br>grade: texte |
| parler()<br>ecrireTableau() |

| Etudiant |
|---|
| age: entier<br>niveau: texte |
| ecoute()<br>prendreNote() |



## Une classe en UML

20




- **Syntaxe des attributs :**

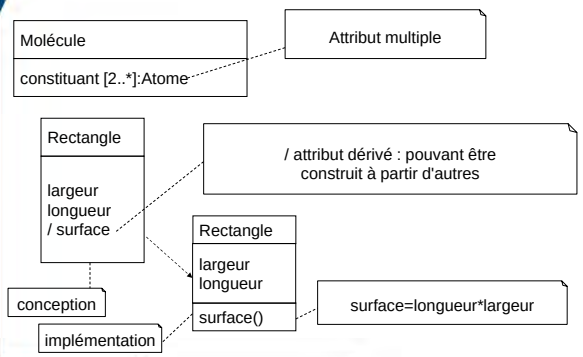
  visibilité nom [multiplicité] : type = valeur_initiale { propriété }
- **Syntaxe des opérations :**

  visibilité nom(arguments) : type { propriété }
- Les éléments « multiplicité », « valeur initiale » et « propriété » sont optionnels.
- Les éléments « visibilité », « type » et « arguments » existent toujours mais peuvent ne pas figurer pour alléger le diagramme.



## Attributs multiples, attributs dérivés

21






## Types et propriétés

22

- Types :
  - Les types *primitifs* : booléen, entier, réel, chaîne de caractère...

    **ex :** longueur : entier
  - Les types énumérés :

    **ex :** couleur : enum{ noir, blanc }
  - Les types classes :

    **ex :** coordonnées : Point
- Propriétés :
  - Permettent d'exprimer des contraintes comme l'aspect non modifiable d'un attribut, *etc.*



## Arguments des opérations

23

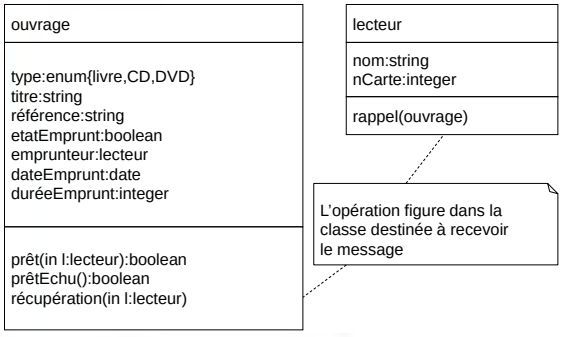
- Syntaxe :

  direction nom : type = valeur_par_défaut
- Direction :
  - in : paramètre d'entrée ne pouvant être modifié (par défaut si non précisée)
  - out : paramètre de sortie
  - inout : paramètre d'entrée pouvant être modifié



## Exemple

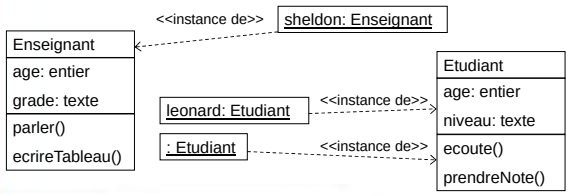
24



## Un objet en UML

25

- En UML, un objet se représente sous forme d'un rectangle avec le nom de l'objet souligné.
- Le nom de la classe de l'objet est indiqué après « : ».
- Un objet peut être anomyme et ne pas comporter de nom



1

# Le principe d'encapsulation



## Objets et services

2

- Dans la description de tout objet, l'orienté objet encourage à séparer :
  - la partie utile pour les autres objets qui collaborent avec lui : ce que les autres doivent savoir de lui afin de solliciter ses services;
  - la partie nécessaire à son fonctionnement propre afin de mettre en œuvre ces services.



## L'encapsulation des données

3

- Principe fondamental de la POO.
- C'est une règle consistant à cacher les données d'une classe aux utilisateurs de la classe.
- Il s'agit d'empêcher l'accès aux données : les données sont encapsulées et leur accès ne se fait que par le biais de méthodes.
- Par conséquent, l'interface d'une classe obéissant à ce principe n'expose jamais ses attributs mais seulement des méthodes.
- Favorise un couplage faible.



## L'encapsulation des données

4

- Tous les langages de programmations orientés objets offrent des limiteurs d'accès permettant d'implémenter le principe d'encapsulation des données.
- Les limiteurs traditionnels sont :
  - **Publique (`public`)** : les utilisateurs de la classe (par ex., méthodes d'autres classes) peuvent accéder aux membres possédant le niveau de visibilité publique. Il s'agit du plus bas niveau de protection des données.
  - **Privé (`private`)** : l'accès aux membres privés est limité aux méthodes de la classe propriétaire. Il s'agit du niveau le plus élevé de protection des données.
  - **Protégé (`protected`)** : voir héritage



## class vs. struct

5

- Le type `class` en C++ est l'intégration des concepts "objet" dans la structure `struct` existant en langage C.
- La notion de classe est maintenant introduite par l'un des deux mots réservés suivants :
  - `struct` : tous les membres (attributs et méthodes) sont publics par défaut
  - `class` : tous les membres (attributs et méthodes) sont privés par défaut.



## Parties privés et publiques

6

- Mots clés réservés : `public` et `private`.

```
class /* ou struct */ {
  public:
  //…
  private:
  //…
};
```

- Ils peuvent apparaître plusieurs fois et dans n'importe quel ordre dans une déclaration de classe.
- Ils signalent que les entités qui suivent sont publiques ou privées (délimiteurs de zones).

## Parties privés et publiques

7

- Les membres `private` ne sont accessibles que par l'intermédiaire des méthodes de la classe.
- L'unité de protection est la classe : une méthode peut accéder à tous les membres de sa classe, *ie.* un objet d'une classe peut accéder à la partie privée (ou publique) d'un objet de la même classe.
- Un membre `public` est accessible directement via n'importe quelle instance de classe suivant la syntaxe habituelle.



## Accesseurs

8

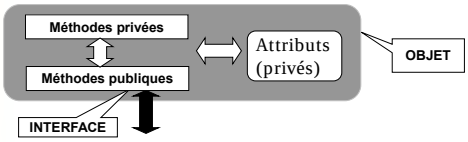
- Pour respecter le principe d'encapsulation, on définit en général des accesseurs qui permettent d'accéder en lecture et en écriture aux données (attributs).
- Traditionnellement, les accesseurs en lecture commencent par `get` (ou `Get`) et les accesseurs en écriture par `set` (ou `Set`).
- Les accesseurs en lecture devraient être des méthodes `const` puisqu'elles ne sont pas censées modifier un attribut de l'objet.



## Encapsulation et interface

9

- L'encapsulation permet d'offrir une interface orientée services et responsabilités, c'est-à-dire, d'offrir aux utilisateurs de la classe une interface indiquant clairement quels services sont offerts et quelles sont les responsabilités de cette classe.




- L'interface correspond à la partie publique de la classe.



## Motivations de l'encapsulation

10

- Permet de changer les structures de données d'une classe sans modifier l'interface de celle-ci et donc sans modifier les classes qui l'utilisent.
- Cette situation arrive fréquemment lorsque l'on veut augmenter l'efficacité (rapidité de traitement) d'une classe ou d'un module, il faut souvent modifier les structures de données en conséquence.
- [Meyer, 2008] *Pourquoi est-il si néfaste d'utiliser une représentation particulière comme spécification ? Plus de 17% des coûts des logiciels (dans la maintenance) viennent du besoin de prendre en compte des changement de format de données qu'ils manipulent.*



## Motivations de l'encapsulation

11

- L'application systématique de l'encapsulation empêche le couplage fort par espace commun ou par contenu (le couplage n'existe qu'au travers des méthodes).
- *La modularité est la propriété d'un système qui a été décomposé en un ensemble cohérent et stable de modules faiblement couplés* (Booch, 1994).
- Permet d'ajouter des règles de validation et des contraintes d'intégrité comme, par exemple :
  - limiter le domaine des valeurs qu'une variable peut prendre (validité)
  - vérifier que cette valeur n'entre pas en conflit avec les valeurs d'autres attributs (intégrité).



## Mauvaise programmation…

12

- La plupart des langages de programmation orientés objet n'obligent le programmeur à protéger les attributs.
- Ainsi il est souvent possible de les déclarer publiques : ceci doit être évité au maximum.

## Encapsulation en UML

13

- En UML, les membres (attributs ou méthodes) privés sont précédés du signe – alors que les membres publics sont précédés du signe +.

| Etudiant |
| --- |
| - age: entier<br>- niveau: texte |
| + ecouter()<br>+ prendreNotes() |



## Encapsulation et amitié

14

- La POO pure impose l'encapsulation des données.
- Cette contrainte s'avère gênante dans certaines circonstances.
- La notion d'amitié propose un mécanisme pour autoriser des fonctions extérieures à la classe de pouvoir accéder aux membres privés de la classe.
- A utiliser avec beaucoup de précaution !!!



## Amitiés : `friend`

15

Le mot clé `friend` permet de déclarer des amitiés dans la classe.

Toutes les déclarations d'amitié se font dans la définition de la classe.

Une déclaration d'amitié vaut pour une déclaration locale de l'élément ami.

Il existe plusieurs situations d'amitié :

- fonction indépendante, amie d'une classe :
  `friend` puis le prototype de la fonction.
  Si une fonction est amie de plusieurs classes : une déclaration d'amitié dans chaque classe.
- méthode d'une classe, amie d'une autre classe :
  `friend` puis le prototype de la méthode avec résolution de portée.
- toutes les méthodes d'une classe, amies d'une autre classe :
  `friend` puis le nom de la classe amie.



## Amitiés : `friend`

16

- Si dans une classe `A`, on a une déclaration d'amitié
  `friend int B::f(char, A);`
  le compilateur doit connaître les caractéristiques de `B` (la définition de `B` doit avoir été compilée avant celle de `A`).
- Pour comprendre la déclaration de `int f(char, A)` dans la classe `B`, le compilateur n'a pas besoin de connaître précisément les caractéristique de `A`. Il lui suffit de savoir qu'il s'agit d'une classe : déclaration `class A;`



## Préférer les petites interfaces

17

- Lors de la conception des interfaces, il faut favoriser les petites interfaces. En effet, on mesure le degré d'encapsulation des données au nombre de fonctions/méthodes qui peuvent y accéder. [Meyers2011]
- En ce sens, une fonction non membre, ou la délégation de certaines opérations à des méthodes de classes non amies ne remet pas en cause de le principe d'encapsulation.

Ainsi, plutôt que :

```
Duree Duree::getDoubleDuree() const
      {return somme(*this,*this); }
```

il vaut mieux définir :

```
Duree doubleDuree(const Duree& d)
      { return somme(d,d); }
```



## Ségrégation des interfaces

18

- Dans la conception d'une interface, il vaut mieux avoir une petite interface réduite aux besoin des clients afin de favoriser un couplage faible.
- Il s'agit du I (Interface segregation) de l'acronyme SOLID.
- Les classes abstraites et l'utilisation du design pattern Adapter peuvent permettre de réduire l'interface d'une classe trop générale aux besoins stricts d'un client.

1

## Naissance d'un objet



## Initialisation d'un objet

2

- A priori les objets suivent les règles habituelles concernant leur initialisation par défaut :
  - Chaque attribut est initialisé de la façon par défaut qui existe en fonction de son type.
  - Pour les types primitifs, sauf exception, aucune initialisation n'est faite.
- Il est donc nécessaire de faire appel à une méthode pour initialiser les données.
- Une telle démarche oblige à compter sur l'utilisateur.
- Ceci peut remettre en cause la validité des données.



## Constructeur

3

- La POO offre un mécanisme très puissant pour traiter ces problèmes : le constructeur.
- Il s'agit d'une méthode (définie comme les autres méthodes) qui sera appelée à chaque création d'objet.
- Une classe peut posséder plusieurs constructeurs (surcharge) offrant différentes possibilités d'initialisation.
- Ils seront utilisés quelle que soit la classe d'allocation de l'objet : statique, automatique ou dynamique.
- Un constructeur sert généralement à initialiser les attributs.
- Il peut faire des tâches complexes comme de l'allocation dynamique de zones mémoire.



## Constructeur par défaut

4

- Si aucun constructeur n'a été défini pour la classe, le compilateur en génère un automatiquement.
- Il est sans paramètre.
- Le constructeur par défaut génère pour chaque attribut les instructions qui mènent au même résultat qu'une non initialisation de la donnée :
  - Attribut d'un type primitif : aucune initialisation.
  - Attribut de type classe : appel du constructeur sans argument (qui peut être un constructeur par défaut).
- Ainsi, il n'est pas obligatoire de définir des constructeurs. Cependant, les attributs d'un objet ne seront alors pas forcément initialisés.



## Constructeurs et encapsulation

5

- Les constructeurs peuvent être publics ou privés.
- En pratique, ils sont la plupart du temps publics.
- Néanmoins, il peut y avoir de bonnes raisons (rares) de faire le contraire.



## Définition d'un constructeur en C++

6

- Pour définir un constructeur, il suffit de définir une méthode portant le même nom que la classe.
- Il ne possède pas de type de retour.
- Il peut comporter un nombre quelconque de paramètres, éventuellement aucun.
- Un constructeur peut être surchargé pour pouvoir initialiser de différentes manières un objet.

## Constructeur : initialisation avec « : »

7

- On peut initialiser les attributs en utilisant « : » entre l'entête de la méthode et le début de la définition de la méthode.
- Après « : », les attributs sont initialisés avec la syntaxe « `identificateur(expression)` ». Les attributs sont séparés par des virgules.
- On est obligé d'utiliser une initialisation avec « : », pour tous les attributs qui requièrent une initialisation à la définition:
  - initialisation d'un attribut constant (dont chaque objet possède sa propre variante);
  - initialisation d'un attribut qui est une référence;
  - initialisation d'un objet pour lequel seul des constructeurs avec arguments ont été définis.



## Initialisation dans la définition de la classe

8

- Depuis C++11, il est possible d'initialiser directement les attributs à l'intérieur de la définition de la classe :

```
class A {
public:
    int x=0, y=10, u=x*y-1;
    char str[10]="truc";
    std::string str2="truc2";
    A(int a):x(a){} // initialisation de x=0 ignoré
};
```

- Si un attribut apparait dans la liste d'initialisation du constructeur (avec « : »), son initialiseur par défaut est ignoré.
- **Attention** : cette possibilité n'existe pas en C++98.



## Constructeur et initialisation

9

- Tout objet doit être initialisé à la définition en utilisant un constructeur.
- Si au moins un constructeur a été défini pour une classe, le constructeur par défaut n'existe plus :
  - L'utilisateur de la classe est obligé d'utiliser un constructeur existant;
  - Si tous les constructeurs ont au moins 1 paramètre, il n'est plus possible d'instancier un objet sans fournir d'argument.
- Si un constructeur sans argument est nécessaire et qu'il y a au moins un constructeur avec argument défini, alors il faut définir un constructeur sans argument.
- En C++11, lorsque les opérations du constructeur par défaut conviennent, on préfère l'instruction « `=default` » après le prototype du constructeur sans argument plutôt que d'en fournir une définition.



## Constructeur avec un paramètre

10

- Dès que la classe `A` définit un constructeur avec un paramètre de type `T` (prototype= `A::A(T)`), les définitions d'objet suivantes sont équivalentes :

```
A a(x);    A a=A(x);
A a=x; // x devient un initialisateur de a
```

où `x` est une expression d'un type compatible (conversion si elle est possible) avec le type `T`.

- De plus, la conversion implicite de T en A est mise en place par le compilateur.
- Si on veut interdire cette conversion implicite, il faut utiliser le mot clé **`explicit`** devant le constructeur.



## Construction d'un tableau d'objets

11

Définition ou allocation d'un tableau d'objets `T` :

- Si la classe `T` comporte un constructeur sans argument, celui-ci sera appelé successivement pour chacun des éléments du tableau.
- Sinon il est obligatoire de fournir un initialisateur pour chacun des éléments du tableau afin de garantir le passage par un constructeur.

1

# Mort d'un objet



## Destructeur

2

- Un objet pourra posséder un (seul) destructeur qui est une méthode appelée au moment de la destruction d'un objet (juste avant sa libération mémoire).
- Il sera utilisé quelle que soit la classe d'allocation de l'objet : statique, automatique ou dynamique.
- Le destructeur sert en général à libérer des zones critiques comme de la mémoire ou toute tâche complexe qu'il peut être utile de faire avant la destruction de l'objet.



## Destructeur par défaut

3

- Si aucun destructeur n'a été défini pour la classe, le compilateur en génère un automatiquement qui ne fait rien.
- Il n'est donc pas obligatoire de définir un destructeur.



## Destructeur et encapsulation

4

- Un destructeur peut être public ou privé.
- En pratique, il est la plupart du temps public.
- Néanmoins, il peut y avoir de bonnes raisons (rares) de définir le destructeur dans la partie privée.



## Définir un destructeur

5

- Pour définir un destructeur de la classe `A`, il suffit de définir une méthode portant le nom `~A()`.
- Il ne possède pas de type de retour.
- Il ne possède pas d'argument.
- Il ne peut pas être surchargé.
- Depuis C++11, on peut explicitement indiquer que l'on souhaite utiliser le destructeur par défaut avec le mot clé `default` :

```
class A {
public:
   ~A()=default;
};
```



## Appel du destructeur

6

- Le destructeur est appelé automatiquement avant la libération mémoire (automatique ou dynamique) de l'objet.
- Le destructeur de chaque attribut est aussi appelé automatiquement avant la libération mémoire de l'objet :
  - attribut d'un type primitif : aucun traitement spécifique;
  - attribut de type classe : appel du destructeur de l'objet composé (qui peut être un destructeur par défaut).
- La destruction (automatique ou dynamique) d'un tableau d'objets `T` provoque l'appel du destructeur de `T` et la libération de l'espace correspondant pour chacun des éléments du tableau (du dernier élément au premier).
- L'appel du destructeur pour les objets automatiques définis dans un bloc se fait toujours à la fin du bloc, dans l'ordre inverse dans lequel ils ont été construits.

1

# Les membres statiques



## Attributs statiques

2

- On déclare avec le qualificatif `static` les attributs que l'on souhaite voir exister en un seul exemplaire pour tous les objets de la classe : c'est une partie partagée par tous les objets.
- Les attributs statiques sont des sortes de variables globales dont la portée est limitée à la classe.
- Les attributs statiques existent indépendamment des objets de la classe (même si aucun objet de la classe n'a encore été créé).
- Leur initialisation ne peut donc plus être faite par le constructeur de la classe.



## Initialisation des attributs statiques

3

- Il est impossible d'initialiser un attribut statique lors de sa déclaration (dans la déclaration de la classe) à l'exception des membres statiques constants.
- Un membre statique doit être initialisé explicitement (à l'extérieur de la déclaration de la classe) dans une unité de compilation.
- Un membre statique n'est pas initialisé par défaut à zéro.
- Un attribut déclaré `static` existe dès que sa classe est chargée en mémoire. L'attribut existe (et est accessible) même s'il n'existe pas d'instance de la classe.



## Méthodes statiques

4

- L'intérêt de ces méthodes est qu'elles n'ont pas besoin d'objet pour être exécutées. Il suffit d'utiliser le nom de la classe, suivie de `::`, suivi du nom de la méthode.
- Il n'y a donc pas d'argument implicite.
- Conséquences :
  - `this` n'est donc pas défini dans les méthodes statiques.
  - Une méthode statique ne peut pas être `const`.
  - Une méthode statique n'a pas accès aux attributs et aux méthodes non statiques de la classe.



## Exemples d'application

5

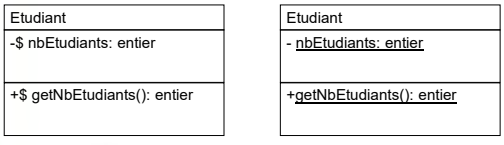
- Compteurs d'objets d'une classe.
- Classe `AgentIntelligent` disposant d'un attribut statique agrégeant des pointeurs vers tous les objets de cette classe (afin qu'un objet de cette classe puisse connaître et communiquer avec n'importe quel autre).
- Une classe représentant un type numérique avec sa valeur min et sa valeur max comme attributs statiques.



## Membres statiques en UML

6

- Le signe $ est utilisé pour indiquer un membre statique. Il est généralement ajouté à côté de l'indicateur de visibilité +, # ou – (encapsulation).
- On peut aussi indiquer qu'un membre est statique en le soulignant.



1

# Surcharge des opérateurs du C++



1

2

## Surcharge d'opérateurs

- On peut surcharger n'importe quel opérateur s'il porte sur au moins un objet en utilisant le mot clé `operator`.
- Il n'est pas possible de surdéfinir des opérateurs portant sur des types de base.
- Le symbole suivant le mot clé `operator` doit obligatoirement être un opérateur défini pour les types de base : il n'est pas possible de créer de nouveaux symboles.
- Il faut conserver la pluralité (unaire, binaire) de l'opérateur initial.
- Les opérateurs conservent leur priorité relative et leur associativité entre eux.



2

3

## Surcharge d'opérateurs

- L'opérateur « . » ne peut pas être surchargé.
- Les opérateurs de cast, l'opérateur `new` et l'opérateur `delete` peuvent être surchargés.
- `new` peut être surchargé pour les types de base.
- Certains opérateurs doivent obligatoirement être définis comme membres d'une classe : `[]`, `()`, `new` et `delete`.
- Si un opérateur @ a été surchargé, l'opérateur @= n'a pas pour autant été surchargé.
- Si les opérateurs `!` et == ont été surchargés, l'opérateur `!=` n'a pas pour autant été surchargé.
- C++ ne fait aucune hypothèse sur la commutativité éventuelle d'un opérateur surchargé.



3

4

## Surcharge d'opérateurs binaires

Deux manières de procéder :

- Méthodes (fonctions membres) de `T` :
  - `T::operator@(T);`
  - `T::operator@(X);`
- Fonctions non membres de `T` :
  - `operator@(T,T);`
  - `operator@(T,X);`



4

5

## Surcharge d'opérateurs unaires

Deux manières de procéder :

- Fonctions membres de `T` :
  - Opérateur unaire préfixe : `T::operator@();`
  - Opérateur unaire postfixe : `T::operator@(int);`
- Fonctions non membres de `T` :
  - Opérateur unaire préfixe : `operator@(T);`
  - Opérateur unaire postfixe : `operator@(T,int);`



5

6

## Surcharge de `operator<<`

- Opérateur très souvent surchargé pour représenter l'insertion d'un élément dans une structure représentant un ensemble:

```
releve_de_temperature<<3<<5.6<<7<<8.1;
```



6

## Surcharge de `operator<<`

7

- Opérateur très souvent surchargé pour des affichages sur des flux `std::ostream`.

```
std::ostream& operator<<(std::ostream& f, const Truc& t);
std::cout<<t1<<t2; // insertion multiple
```

- C'est obligatoirement une fonction non membre puisque le premier argument est une référence sur un objet `std::ostream`.
- La possibilité d'une insertion multiple n'est possible que si l'opérateur renvoie une référence sur ce même objet `std::ostream`.

antoine.jouglet@utc.fr Programmation Orientée Objet

7

## Surcharge de `operator[]`

8

- Pour que `a[i]` soit une *lvalue*, il est nécessaire que la valeur de retour fournie par l'opérateur soit une référence.
- C++ impose de surcharger cet opérateur sous forme de fonction membre.
- En général, on définit un second opérateur `operator[] const` destiné uniquement aux objets constants, qui permettra la lecture uniquement.

antoine.jouglet@utc.fr Programmation Orientée Objet

8

## Surdéfinition de `operator()`

9

- Lorsqu'une classe surdéfinit l'opérateur `operator()`, on dit que les objets auxquelles elle donne naissance sont des objets fonctions.
- Ils peuvent être utilisés de la même manière qu'une fonction.

antoine.jouglet@utc.fr Programmation Orientée Objet

9

1

# Objets et conversions



## Conversion de type

2

- Une conversion est explicite lorsque l'on fait appel à un opérateur de cast.
- Une conversion est implicite si elle n'est pas mentionnée par l'utilisateur mais mise en place par le compilateur en fonction du contexte :
  - Dans les affectations : conversion forcée dans le type de la variable réceptrice.
  - Dans les appels de fonction : conversion forcée d'un argument dans le type déclaré du prototype.
  - Dans les expressions : pour chaque opérateur, il y a conversion éventuelle de l'un de ses arguments dans le type de l'autre.



## Généralisation des conversions de type

3

- Soient deux classes A et B, les conversions se généralisent :
  - un constructeur de la classe A recevant un argument de type B réalise une conversion de B en A : `A::A(B)`
  - au sein d'une classe A, on peut définir un opérateur de cast réalisant la conversion de A vers un autre type de classe B : il suffit de surcharger `A::B()`



## Conversion de type : B en A

4

- Définition de :
  - `A::A(B)`
  - `A::A(B&)`
  - `A::A(const B&)`
- Si on veut transmettre les arguments par références et que l'on souhaite des conversions implicites grâce au constructeur, il faut utiliser des `const`.
- Cela peut être pratique de ne pas le mettre pour empêcher des conversions implicites.
- On peut interdire l'utilisation du constructeur pour des conversions implicites en mettant le qualificatif **`explicit`** devant le constructeur.



## Conversion de type : A en B

5

- Définition de :
  - `A::B()`
- Un opérateur de cast doit toujours être défini comme une fonction membre (méthode).
- Le type de la valeur de retour (qui est alors celui défini par le nom de l'opérateur) ne doit pas être mentionné.
- Si on veut pouvoir utiliser cet opérateur à l'extérieur de la classe, il devra être public.



## Conversions

6

- Les conversions définies par l'utilisateur (cast ou constructeur) ne sont mises en œuvre que lorsque cela est nécessaire.
- Il ne faut pas définir simultanément la même conversion A vers B en prévoyant à la fois un constructeur `B::B(A)` et un cast `A::B()` : ambiguïté dès qu'une conversion de A en B est nécessaire.

1

# Détection des erreurs et gestion des exceptions



## Erreurs

2

- La détection des erreurs est au cœur de la fiabilité et de la robustesse des logiciels.
- On peut distinguer les erreurs de programmation et les erreurs d'utilisation.
- Les erreurs de programmation sont les erreurs de syntaxe, de typage et de conception dues au développeur d'un module :
  - Les erreurs de syntaxe et de typage sont détectées par le compilateur.
  - Il faut pouvoir faciliter la détection des erreurs de conception.
- Les erreurs d'utilisation sont les erreurs dues à l'utilisateur d'un module (qui lui-même peut être le développeur d'un autre module) :
  - il faut pouvoir détecter ces situations exceptionnelles d'utilisation du module lors de son développement;
  - il faut pouvoir laisser l'utilisateur décider comment traiter l'erreur.



## La macro `assert` (entête <cassert>)

3

- La macro `assert` permet de tester les préconditions ou les postconditions d'une fonction (membre ou non-membre) :

```
double diviser(double x, double y) {
      assert((y!=0));
      return x/y;
}
```

- Il est possible d'ajouter des chaînes de caractères afin de mieux comprendre l'erreur :

```
double diviser(double x, double y) {
      assert((y!=0 && "diviseur nul"));
      return x/y;
}
```

- Les doubles parenthèses ne sont nécessaires que dans une expression comportant des parenthèses car `assert` est une macro qui doit être correctement parsée.



## `assert` (entête <cassert>)

4

- La macro assert dépend de la macro NDEBUG :

```
#ifdef NDEBUG
#define assert(condition) ((void)0)
#else
#define assert(condition)
     /*implémentation de assert*/
#endif
```

- Si la macro `NDEBUG` est définie, `assert()` ne fait rien.
- Si la macro `NDEBUG` n'est pas définie, `assert()`appelle `abort()` lorsque l'expression évaluée retourne 0.



## `static_assert` (C++11)

5

- La déclaration

« `static_assert(expr, message);` »

permet de déclencher une erreur au moment de la compilation si `expr` n'est pas évaluée comme égale à `true`.

- `expr` doit être une expression convertible en `bool` évaluable à la compilation
- `message` est une chaine de caractères de type `const char*`.
- Depuis C++17, `message` peut être omis.
- Cette déclaration est souvent utilisée avec les patrons déclarées dans le header `<type_traits>`.



## Conditions exceptionnelles

6

- Un logiciel, aussi fiable (éprouvé, debuggé) soit-il peut rencontrer des conditions exceptionnelles qui risquent de compromettre la poursuite de son exécution…

```
class Fraction {
    int num; // numérateur
    int den; // dénominateur
    Fraction(int n, int d): num(n),den(d) {
          if (d==0) // Que faire ??????
    }
};
```

## Séparer la détection d'un incident de son traitement

7

- Dans les programmes importants, il est rare que la détection d'un incident et son traitement puissent se faire dans la même partie du code.
- La dissociation devient nécessaire dans le développement de composants réutilisables destinés à être utilisés dans divers programme :
  - les incidents exceptionnels ne peuvent être détectés que par les composants;
  - mais c'est à l'utilisateur de choisir la conduite à adopter en cas d'incident.



## Traitement standard des erreurs en C

8

- La technique la plus répandue consiste à fournir un code d'erreur comme valeur de retour des différentes fonctions :
  - Avantage : sépare la détection de l'anomalie de son traitement;
  - Inconvénient : fastidieuse car elle implique l'examen systématique des valeurs de retour avec une retransmission du code à travers la hiérarchie des appels;
  - Inconvénient : difficulté de maintenance du programme.



## Mécanisme des exceptions

9

- C'est un moyen très puissant de gestion des anomalies.
- Il Découple totalement la détection d'une anomalie (levée d'une exception) de son traitement (gestionnaire d'exception).
- Il assure une gestion convenable des objets automatiques (gestion de la mémoire).
- La **levée d'une exception** est une rupture de séquence d'instructions déclenchée par une instruction `throw` suivie d'une expression d'un type donné.
- Il y a alors branchement à un ensemble d'instructions nommé **gestionnaire d'exception** choisi en fonction du type de l'exception déclenchée.



## Avantages

10

Contrairement au classique code d'erreur renvoyé par une fonction :

- Une exception se propage depuis l'appelé vers l'appelant jusqu'à ce qu'elle rencontre un bloc de code qui s'occupe de la traiter . Le compilateur prend en charge cette remontée.
- Le programmeur n'a donc plus à se soucier de tester la réussite ou non des fonctions qu'il appelle au moyen d'un grand nombre de tests.
- Une exception doit être traitée : elle ne peut pas être ignorée : si elle ne l'est pas dans la fonction qui en est à l'origine, elle doit l'être dans l'une des fonctions appelantes.



## Déclencher une exception

11

- Lorsque que le programme détecte une anomalie, il déclenche (lève, lance, …) une exception grâce à l'instruction `throw`.
- Il ne s'agit pas de traiter l'erreur mais d'interrompre le déroulement du programme.
- Le rôle de `throw` est de transmettre de l'information (sur l'anomalie) au gestionnaire qui traitera cette anomalie.



## Valeur et type d'une exception

12

- L'instruction `throw` est accompagnée d'une expression d'un type quelconque (une classe, un type primitif) choisi par le développeur.
- La valeur contient les informations nécessaires au traitement de l'anomalie par son gestionnaire;
- Le type de la valeur déterminera le gestionnaire d'exception qui « capturera » et qui aura donc la responsabilité de traiter l'anomalie.
- Pour distinguer les exceptions les unes des autres, on utilise alors généralement des types dédiés (des classes) à des anomalies spécifiques.

## Exemple

13

```
class FractionException{
  string info;
public:
  FractionException(const string& s):info(s){}
  const string& get_info() const { return info; }
};
class Fraction { ...
public:
  Fraction(int n, int d): num(n),den(d) {
    if (d==0) // déclencher une exception
      throw FractionException("dénominateur nul");
  }
};
```



## Après le déclenchement d'une exception…

14

- La séquence d'instructions est interrompue.
- Cependant :
  - les variables automatiques des blocs dont on provoque la sortie sont supprimées;
  - cela entraîne l'appel du destructeur de tout objet automatique déjà construit et devenant hors de portée.
  - Attention : ce mécanisme ne s'applique pas aux objets dynamiques.
- L'exécution du programme continue dans le gestionnaire d'exception destiné à traiter cette anomalie (s'il existe).



## Observer les exceptions

15

- C'est l'utilisateur qui décide quelles sont les parties du programme devant être surveillées (celles où des anomalies sont potentiellement détectables).
- Un bloc d'instructions devant être surveillé doit être inclus dans un bloc dit « try » :

```
try {
  /* instructions susceptibles de
  déclencher (ou pas) des exceptions */
  …
}
```



## Capturer les exceptions

16

- Pour être traitée, une exception déclenchée dans un bloc « `try` » doit être capturée par un gestionnaire d'exception.
- Lorsqu'une exception est transmise à un bloc « `try` », un gestionnaire du type mentionné avec l'instruction `throw` est recherché dans les différents **blocs « `catch` »** associés au bloc « `try` » :

```
try { …}
catch(type1 var1){ /* instructions */ }
catch(type2 var2){ /* instructions */ }
…
```

- En général, un gestionnaire d'exception ne comporte d'instruction d'arrêt de l'exécution (`exit`, `abort`).
- Après l'exécution des instructions du gestionnaire concerné, l'exécution continue à la première instruction suivant le dernier gestionnaire `catch`.



## Exemple

17

```
int main(){
  try {
      Fraction fa(1,2); // ok, pas d'exception
      …
      Fraction fb(3,0); // aïe aïe aïe !
      …
  }
  catch(int i){ cout<<i<<"\n";      }
  catch(FractionException e){
      cout<<e.get_info()<<"\n";
  }
  cout<<"reprise de l'exécution\n";
  return 0;
}
```



## Choix du gestionnaire

18

- Un gestionnaire d'exception convient si :
  - Il correspond au type exact mentionné dans `throw`. Le qualificatif `const` (ou la référence) ne compte pas.
  - Il correspond à un type de base du type mentionné dans `throw`. Cela permet de regrouper plusieurs exceptions que l'on peut traiter plus ou moins finement.
  - Il correspond à un pointeur ou une référence sur une classe de base du type mentionné dans `throw` (lorsque ce type est lui-même un pointeur ou une référence).
  - Le gestionnaire par défaut est utilisé : `catch(...)` permet de capturer un gestionnaire d'un type quelconque.

## Choix du gestionnaire

19

- Les gestionnaires sont essayés dans l'ordre de leur définition.
- Dès qu'un gestionnaire correspond, il est exécuté en ignorant les éventuels gestionnaires suivants.
- L'ordre dans lequel sont définis les blocs `catch` est donc important.

- Quand une exception est levée par une fonction, le gestionnaire est d'abord recherché dans l'éventuel bloc `try` associé à cette fonction.
- Si ce gestionnaire n'est pas trouvé ou si aucun bloc `try` n'est associé, la recherche est poursuivie dans un éventuel bloc `try` associé à une fonction appelante et ainsi de suite.



## Absence de gestionnaire convenable

20

- Si aucun gestionnaire n'est trouvé, la fonction `terminate` est exécutée. Par défaut cette dernière appelle la fonction `void abort();`
- Cette fonction génère le signal `SIGABRT` qui par défaut provoque l'arrêt du programme en retournant un code d'erreur de terminaison critique au système.
- Le programme est alors terminé sans exécuter les destructeurs des objets automatiques ou statiques et sans appeler d'autres fonctions.
- La fonction ne retourne jamais à la fonction appelante.

- Il est possible de demander qu'à la place de `terminate` soit appelée une fonction de notre choix dont l'adresse est fournie avec l'instruction `set_terminate`.
- Il est cependant nécessaire que cette fonction mette fin à l'exécution du programme, qu'elle n'effectue pas de retour et qu'elle ne lève pas d'exception.



## Retransmettre une exception

21

- L'instruction `throw` sans expression dans un bloc `catch` retransmet l'exception au niveau englobant (avec la même valeur reçue). Permet par ex de compléter un traitement.

- Attention : si cette instruction n'existe pas, un gestionnaire d'un niveau englobant ne sera jamais appelé (même si sa correspondance avec le type de l'exception est plus direct).



## `noexcept`

22

- Depuis C++11, il est conseillé de spécifier quelles sont les fonctions pour lesquelles nous avons une garantie qu'elles ne déclenchent aucune exception.

- Pour cela, il suffit d'ajouter le mot clé `noexcept` après les paramètres de la fonction et l'éventuel mot clé `const` :

```
type fx(type 1 argk, …, typek argk) noexcept;
type fx(type 1 argk, …, typek argk) const noexcept;
```

- Les compilateurs sont alors capables d'utiliser cette information pour optimiser le code.

1

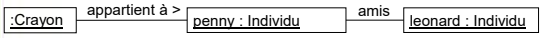
# Liens entre objets
# Associations entre classes



## Liens entre objets

2

- Les traits qui relient les objets symbolisent des liens qui existent entre les objets

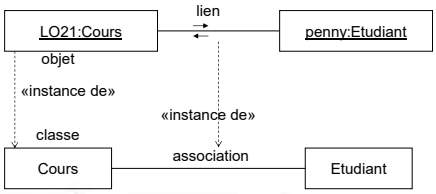





## Liens et associations

3

- Les messages (appels de méthodes) entre objets circulent le long des liens.
- L'association exprime la connexion entre classes correspondant à une famille de liens : c'est **une abstraction des liens** qui existent entre les objets instances de ces classes. Elle est le reflet d'une connexion qui existe dans le domaine d'application.

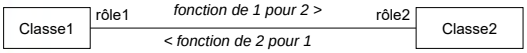




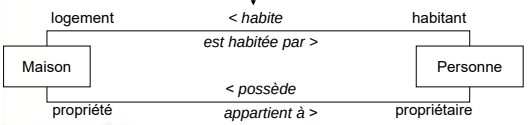

## Nommage des associations, rôles

4

- Les associations peuvent être nommées (fonctions : verbes) ainsi que les extrémités d'associations (rôles : substantifs)




- Il est possible d'avoir plusieurs associations de nature différentes entre deux mêmes classes



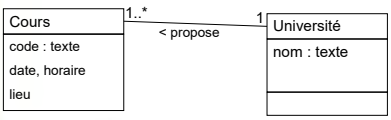



## Valeurs de multiplicités

5

- Chaque extrémité d'une association peut porter une information de multiplicité qui précise le nombre d'instances qui participent à l'association.
- La multiplicité apparaît à proximité de la classe concernée.

| | |
|---|---|
| 1 | un seul |
| 0..1 | zéro ou 1 |
| M..N | de M à N |
| * | de zéro à plusieurs |
| 0..* | de zéro à plusieurs |
| 1..* | de 1 à plusieurs |

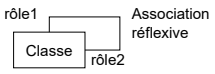


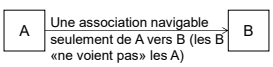



## Réflexivité, Navigabilité

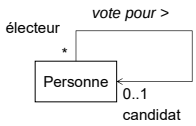
6

- Une association peut relier une classe à elle-même.
- Les rôles devraient alors apparaître.



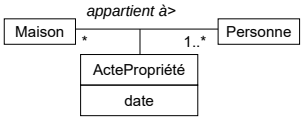

- On peut restreindre le sens de navigation (sens possibles d'envoi de messages entre objets instances de la classe le long des liens instances de l'association).
- Par défaut une association est navigable dans les 2 sens.



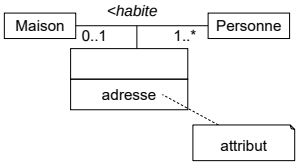
## Classes-Associations

7

- Il est possible de représenter une association par une classe pour y associer des attributs ou opérations




- Si la classe-association n'a pas elle-même d'association, le nom est facultatif

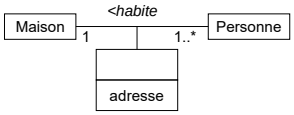


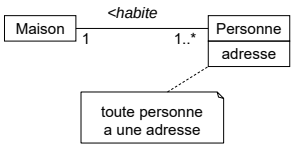



## Classes-Associations

8

- Dans le cas d'association 1-N (multiplicité 1 d'un côté), la classe-association est en général transformable en attribut du côté classe de multiplicité N



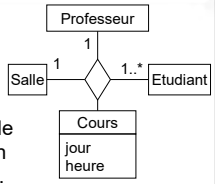





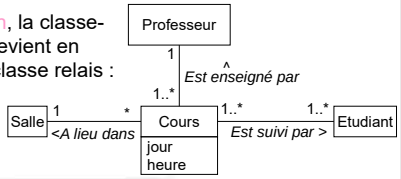

## Arité des associations

9

- En général, les associations sont binaires. UML permet toutefois l'expression d'arités supérieures (analyse).
- Cette notation est parfois source de confusion dans les multiplicités (un étudiant n'a pas qu'un professeur).




- En conception, la classe-association devient en général une classe relais :






## Attributs représentant un objet [implémentation]

10

- Lors de l'implémentation d'une solution orientée objet, les associations entre classes s'expriment souvent au travers d'attributs objet.
- Lors de la définition d'une classe A, on peut définir un attribut représentant une instance d'objet B de la même classe (A=B) ou d'une autre classe (A≠B).
- Cet attribut peut être :
  - l'objet lui-même (type B)
  - une référence ou un pointeur sur cet objet.
- La nature de la représentation (objet ou adressage indirect) dépend :
  - de la nature de l'association entre les classes A et B (agrégation, composition ?) [conception, implémentation]
  - des spécifications techniques (allocation dynamique, types partiellement définis, …) [implémentation]



## Force des couplages

11

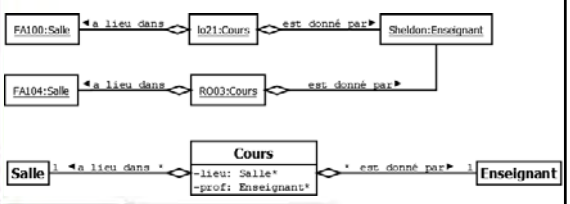
- Une relation exprime une forme de couplage entre classes.
- La force de ce couplage dépend de la nature de la relation.
- Par défaut, l'association exprime une relation à couplage faible : les classes associées restent relativement indépendantes l'une de l'autre.
- On peut exprimer des couplages plus forts avec l'agrégation et la composition.



## L'agrégation

12

- L'agrégation est une forme particulière d'association qui exprime un couplage plus fort entre classes.
- Elle représente une relation composant (agrégé) / composé (agrégat)
- Elle représente une connexion bidirectionnelle dissymétrique.
- Elle se représente avec un petit losange creux du côté de l'agrégat.



## L'agrégation

13

- Une agrégation peut notamment (mais pas nécessairement) exprimer :
  - qu'une classe (« l'agrégat ») possède des éléments d'une autre classe (instances de la classe agrégée),
  - qu'une classe (« l'agrégat ») possède se définit en fonction d'éléments d'une autre classe (instances de la classe agrégée),
  - qu'un changement d'état d'une classe, entraîne un changement d'état d'une autre,
  - qu'une action sur une classe, entraîne une action sur une autre.
- A un même moment, une instance d'élément agrégé peut être liée à plusieurs instances de la classe agrégatrice (l'élément agrégé peut être partagé).
- Une instance qui a été agrégée peut exister sans l'agrégat (et inversement) : les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être indépendants.



## La composition

14

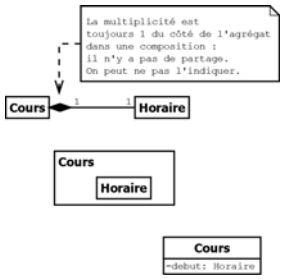
- La composition est une forme d'agrégation avec un couplage plus important.
- Ce couplage indique que les composants ne sont pas partageables.
- La destruction de l'agrégat entraîne la destruction des composants agrégés.
- Les termes conteneur et composite sont parfois utilisés pour désigné l'agrégat.
- La valeur maximale de multiplicité du côté de l'agrégat ne doit pas être supérieure à 1.



## La composition

15

- Plusieurs possibilités pour la représentation :






## Agrégation et composition

16

- L'agrégation et la composition sont des vues subjectives.
- Lorsqu'on représente (avec UML) qu'une molécule est « composée » d'atomes, on sous-entend que la destruction d'une instance de la classe "Molécule", implique la destruction de ses composants, instances de la classe "Atome" (cf. propriétés de la composition).
- Bien qu'elle ne reflète pas la réalité cette abstraction de la réalité nous satisfait si l'objet principal de notre modélisation est la molécule...
- Il faut se servir de l'agrégation et de la composition pour ajouter de la sémantique aux modèles lorsque cela est pertinent.



## Problématiques associées [implémentation]

17

- Responsabilité du cycle de vie d'un objet attribut (création / destruction).
- Copie et affectation entre objets (copie partielle ou profonde ?).
- Persistance de la validité des liens entre objets quand les cycles de vie sont liés.



## L'idiome Pimpl : « Pointer to IMPLementation »

18

- L'idiome Pimpl est une technique d'implémentation couramment utilisée en C++.
- Elle consiste à remplacer les données membres d'une classes (les attributs) par un unique pointeur vers une structure contenant les données.
- Les données sont alors disponibles indirectement au travers du pointeur. Ce pointeur est donc destiné à implémenter une composition entre la classe initiale et la structure contenant les données.
- Le but est d'obtenir des ABI (Application Binary Interface) stables et réduire les temps de compilation.

1

# Copies et affectations entre objets



## Construction par recopie : contexte

2

- Situations dans lesquelles il est nécessaire de construire un objet à partir d'un autre objet de même type :
  - un objet est initialisé, lors de sa définition, avec un autre objet de même type.
  - la valeur d'un objet doit être transmise par valeur en argument à une fonction. Dans ce cas, il est nécessaire de créer, dans un emplacement local à la fonction, un objet qui soit une copie de l'argument effectif.
  - un objet est renvoyé par valeur comme résultat d'une fonction; il faut alors créer, dans un emplacement local à la fonction appelante, un objet qui soit une copie du résultat.



## Constructeur de recopie : contexte

3

- Initialisation par recopie : création d'un objet par recopie d'un objet existant de même type.
- Mécanisme prévu : le constructeur de recopie.
- Si un tel constructeur n'existe pas, un traitement par défaut est prévu : le constructeur de recopie par défaut qui effectue une copie de chacun de ses membres.
- Si l'objet comporte des objets attributs, la recopie par défaut se fait membre par membre (appel de leur constructeur de recopie).
- L'affectation n'est pas une situation d'initialisation par recopie.



## Constructeur de recopie : implémentation

4

- Un constructeur par recopie d'une classe `T` est un constructeur qui prend en argument une référence d'un autre objet `T`.
- Son unique argument doit être transmis par **référence**
- Les deux formes `T::T(T&)` et `T::(const T&)` peuvent exister au sein d'une même classe.
- Si on redéfinit un constructeur de recopie, aucune recopie n'est faite de manière automatique. C'est au constructeur de prendre tout en charge.



## Instructions effectuées par le constructeur de recopie

5

- Ces instructions sont fortement dépendantes de la relation entre les deux objets impliqués dans l'opération.
- On doit nécessairement se poser des questions dans le cas d'une classe possédant des attributs représentant d'autres objets. Si une classe `A` possède un attribut représentant un objet d'une classe `B`, les instructions effectuées dépendent alors de la force du couplage qui lie un objet `A` à son attribut `B` (agrégation, composition ?).
- La force du couplage doit permettre de savoir si on doit faire une copie profonde (création d'un objet `B` indépendant) ou partielle (deux objets `A` partagent le même objet `B` par l'intermédiaire d'une référence ou d'un pointeur) après la recopie.



## Opérations effectuées par le constructeur de recopie

6

- Dans le cas d'un attribut de type `B`, le lien est une composition (un objet `A` inclut un objet `B`).
- Dans le cas d'un attribut de type pointeur (ou référence) sur `B`, le lien peut être une composition ou une agrégation (dépend de qui a la responsabilité d'un objet `B`).
- S'il s'agit d'une composition, il faut s'assurer que les opérations du constructeur de recopie prennent bien en charge la duplication de l'objet `B` (copie profonde).

## Contextes spécifiques

7

- `T t=T(…);` : il s'agit d'une déclaration comportant un initialisateur constitué d'une expression de type `T`.
  Cette déclaration est traitée comme `T t(…);`
  Aucun constructeur de recopie n'est donc appelé pour cet objet.
- Si le constructeur de recopie d'une classe est privé, toute tentative d'initialisation par recopie d'un objet (à la construction ou lors d'une transmission par valeur) conduira à un message d'erreur.



## Activation ou désactivation du constructeur de recopie par défaut (C++11)

8

- Il est possible de déclarer explicitement que l'on utilise le constructeur de recopie par défaut en utilisant le mot clé `default`.
- Il est possible de désactiver le constructeur de recopie par défaut explicitement en utilisant le mot clé `delete`.
- Cette dernière forme est recommandée pour interdire la duplication d'un objet y compris par les méthodes membres de la classe.



## Opérateur d'affectation

9

- La possibilité d'affectation globale s'étend aux objets de même type : elle recopie des valeurs des attributs (publics ou privés).
- On peut redéfinir son comportement en surchargeant l'opérateur `operator=`.
- C++ impose que l'opérateur `operator=` soit une méthode de la classe.
- Eviter l'affectation d'un objet à lui-même en vérifiant l'adresse de l'objet passée en argument avec `this`.



## Opérateur d'affectation

10

- Si un argument de la fonction membre `operator=` est transmis par référence, il est nécessaire de lui associer le qualificatif `const` si l'on souhaite pouvoir utiliser un objet constant à droite de l'opérateur d'affectation.
- Attention : une transmission par valeur de l'argument signifie appel du constructeur de recopie (en supplément…).
- Il est conseillé aussi de prévoir une valeur de retour pour l'affectation (affectations multiples) transmise par référence ou par valeur.



## Recopie et affectation

11

- Si une classe dispose de pointeurs sur des parties allouées dynamiquement composées par ses objets, la copie d'objets de la classe aussi bien par le constructeur de recopie par défaut que par l'opérateur d'affectation par défaut n'est pas satisfaisante.
- Il est nécessaire de munir la classe des 4 méthodes suivantes :
  - constructeurs (chargés de l'allocation de certaines parties de l'objet)
  - destructeur (libère correctement tous les emplacements dynamiques créés par l'objet)
  - constructeur de recopie
  - opérateur d'affectation



## Forme canonique d'une classe

12

```
class A {
public:
  A(…);
  A(const A&);
  ~A();
  A& operator=(const A&);
};
```

- Depuis C++11, deux nouveaux concepts sont apparus : le move-copy-constructor et le move-assignment-operator permettant de réutiliser les ressources d'objets temporaires (rvalues).

1

# Les pointeurs intelligents C++11

[Meyers, 2014]



## Inconvénients du pointeur « brut »

2

- La déclaration d'un pointeur brut n'indique pas s'il pointe sur un seul objet ou sur un tableau.
- La déclaration ne précise pas si nous devons détruire l'élément sur lequel il pointe lorsque nous n'en avons plus besoin (i.e. si le pointeur « détient » l'élément pointé).
- Si nous devons détruire l'élément pointé, rien n'indique si nous devons employer `delete` ou un autre mécanisme.
- Même si nous savons que nous pointons sur un seul élément ou sur un tableau et que nous devons utiliser `delete` (ou `delete[]`), il est parfois difficile d'être sûr que la destruction n'aura lieu qu'une fois.
- Il n'y en général aucun moyen de savoir si un pointeur « pendouille », i.e s'il pointe vers une zone mémoire qui ne contient plus l'objet sur lequel le pointeur est supposé pointer.



## Eviter les pièges des pointeurs bruts

3

- Les pointeurs intelligents (entête `<memory>`) permettent d'encapsuler les pointeurs bruts, en se comportant comme les pointeurs encapsulés mais en proposant des mécanismes permettant d'éviter de nombreux problèmes.
- Ils permettent d'éviter les fuites de ressources en s'assurant que les objets alloués dynamiquement sont détruits de la bonne manière et au bon moment :
  - Utiliser `unique_ptr` pour la gestion d'une ressource à propriété exclusive.
  - Utiliser `shared_ptr` pour la gestion d'une ressource à propriété partagée.
  - Utiliser `weak_ptr` pour des pointeurs de type `shared_ptr` qui ne participent pas à la propriété partager et qui peuvent pendouiller.



## `std::unique_ptr`

4

- Taille identique et efficacité similaire à un pointeur brut pour les mêmes opérations.
- **Incarne la sémantique de propriété exclusive** : un `unique_ptr` non nul détient toujours l'élément sur lequel il pointe.
- Un déplacement (avec l'opérateur `move`) d'un `unique_ptr` transfère la propriété depuis le pointeur source vers le pointeur destination. Seuls les `unique_ptr` non-`const` sont transférables.
- La copie d'un `unique_ptr` est interdite.
- Lors de la destruction un `unique_ptr` non nul libère sa ressource.
- Par défaut, la destruction de la ressource se fait en appliquant `delete` sur le pointeur brut encapsulé. Il est possible de préciser un autre supprimeur mais celle-ci fait alors partie du type du `unique_ptr` (version à deux paramètres du patron).
- L'`unique_ptr` est par ex. souvent utilisé comme type de retour d'une fonction qui fabrique des objets d'une hiérarchie (le code appelant s'en appropriant la propriété dans le cas d'une composition).
- Un `std::unique_ptr` est convertible implicitement en `std::shared_ptr`.



## `std::unique_ptr` : exemple

5

```
#include <iostream>
#include <memory>
int main () {
                                          // foo   bar    p
  std::unique_ptr<int> foo;               // null
  std::unique_ptr<int> bar;               // null  null
  int* p = nullptr;                       // null  null  null
  foo = std::unique_ptr<int>(new int(10)); // (10)  null  null
  bar = std::move(foo);                   // null  (10)  null
  p = bar.get();                          // null  (10)  (10)
  *p = 20;                                // null  (20)  (20)
  p = nullptr;                            // null  (20)  null
  foo = std::unique_ptr<int>(new int(30)); // (30)  (20)  null
  bar = std::move(foo);                    // null  (30)  null
  p = bar.release();                      // null  null  (30)
  *p = 40;                                // null  null  (40)
  return 0;
}
                                          // null  null  null
```



## `std::unique_ptr`

6

- Un `std::unique_ptr` existe sous deux formes, l'une pour les objets individuels, l'autre pour les tableaux :

```
auto objet = std::unique_ptr<int>(new int(10));
auto tab = std::unique_ptr<int[]>(new int[30]);
```

- En conséquence, il n'y a jamais d'ambiguïté sur le type.
- Il n'existe pas d'opérateur d'indexation `[]` dans la forme adaptée aux objets individuels.
- Il n'existe pas d'opérateur de déréférencement `*` et `->` dans la forme adaptée aux tableaux.
- Depuis C++14, il est recommandé d'utiliser `std::make_unique` pour allouer la ressource exclusive :

```
auto foo = std::make_unique<int>(); // *foo=0
auto foo2 = std::make_unique<int>(42); // *foo2=42
auto tab = std::make_unique<int[]>(5); // 5 zéros
```

## std::shared_ptr

7

- Réunir le meilleur de deux mondes : un système qui opère de façon automatique pour récupérer de la mémoire allouée (comme le ramasse-miettes), mais qui s'applique à toutes les ressources et de manière prévisible (comme les destructeurs).
- Lorsqu'un objet est manipulé au travers de `std::shared_ptr`, la gestion de son cycle de vie est assurée par ces pointeurs avec une propriété partagée.
- L'objet n'est détenu par aucun `std::shared_ptr` en particulier.
- Tous les `shared_ptr` pointant sur cet objet collaborent pour assurer sa destruction lorsqu'il n'est plus utile.
- Lorsque le dernier `shared_ptr` pointant sur l'objet arrête de pointer dessus, ce `shared_ptr` détruit l'objet concerné.
- Il est aussi possible de préciser autre supprimeur que `delete` au moment de la construction du pointeur. Contrairement aux `unique_ptr`, la gestion en est plus souple car la fonction de destruction ne fait pas partie du type du pointeur.
- Depuis C++14, il est conseillé d'utiliser `std::make_shared()` pour créer des `shared_ptr`. Cette fonction ne permet pas de préciser un supprimeur.
- Il n'existe pas d'API pour manipuler des tableaux au travers d'un `shared_ptr`.



## Le compteur de références

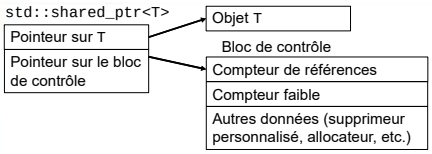
8

- Pour savoir qu'il est le dernier à pointer sur une ressource, le `std::shared_ptr` consulte le compteur de références de cette ressource.
- Ce compteur permet de savoir à tout instant le nombre de `std::shared_ptr` pointant dessus.
- Ce compteur de référence est encapsulé dans une structure de donnée appelé « bloc de contrôle ».
- Impact sur les performances :
  - La taille des `std::shared_ptr` est deux fois plus importante que les pointeurs bruts : ils contiennent un pointeur brut (vers la ressource) et un pointeur vers le bloc de contrôle.
  - La mémoire associée au bloc de contrôle doit être allouée dynamiquement.
  - Les incrémentations et les décrémentations du compteur de références doivent être atomiques (en général plus lentes que les opérations non atomiques).



## Le bloc de contrôle [Meyers, 2014]

9




- Un bloc de contrôle est mis en place par la fonction qui crée le premier `shared_ptr` sur l'objet.
- `std::make_shared()` crée toujours un bloc de contrôle.
- Un bloc de contrôle est créé lorsqu'un `std::shared_ptr` est construit à partir d'un `unique_ptr` (qui sera fixé à `nullptr` puisqu'il n'assume plus la propriété exclusive de l'objet).
- Lorsqu'un constructeur de `std::shared_ptr` est appelé avec un pointeur brut, il crée un bloc de contrôle (il ne faut donc pas créer deux `shared_ptr` à partir d'un pointeur brut : d'où une préférence pour utiliser `make_shared()`).
- Les `shared_ptr` construits à partir de `shared_ptr` et de `weak_ptr` ne provoquent pas la construction d'un nouveau bloc de contrôle.



## std::weak_ptr

10

- On utilise les `weak_ptr` pour avoir des pointeurs qui opèrent comme des `shared_ptr` sans participer au partage de propriété (ils n'affectent pas le compteur de références).
- Ils sont en général créés à partir d'un `shared_ptr` avec lesquels ils seront en lien.
- Potentiellement, ils peuvent alors pointer sur une ressource qui a déjà été détruite.

```
auto sp=std::make_shared<int>(); // le compteur est à 1
…
std::weak_ptr<int> wp(sp); // le compteur est toujours à 1
…
sp=nullptr; // compteur=0, l'objet est détruit et wp pendouille.
```

- La méthode `expired()` renvoie `true` s'il l'objet pointé a expiré.



## std::weak_ptr

11

- Les `weak_ptr` ne peuvent pas être déréférencés ni comparés à `nullptr`.
- Pour cela, il faut utiliser la méthode `lock()` qui renvoie un `shared_ptr` nul si l'objet est périmé ou pointant sur l'objet avec une participation au partage de propriété si l'objet n'est pas périmé :

```
std::shared_ptr<int> ps1=wp.lock();
auto ps2=wp.lock(); // avec auto, c'est plus simple
if (ps2!=nullptr)…
```

- Une application intéressante est son utilisation dans l'implémentation du **design pattern observateur** : l'observable encapsule un ensemble de `weak_ptr` pointant sur des ressources observatrices gérées par des `shared_ptr`. Ces `weak_ptr` ne participent pas à la propriété mais peuvent savoir au moment de l'émission d'un message aux observateurs si l'un deux a été détruit (pour ne pas invoquer une méthode dessus).

1

# Hiérarchie de classes



## Classes et instances

2

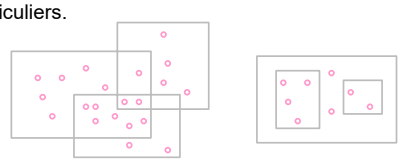
- Tout objet est instance d'une classe et toute les instances d'une classe représentent le même type d'éléments :
  - Toutes les instances répondent au même ensemble de messages et utilisent le même ensemble de méthodes pour le faire.
  - Toutes les instances ont les mêmes attributs.
- Pour l'instant, chaque objet est instance d'exactement une classe : il n'y a pas d'intersection entre classes.



## Intersections et Inclusions entre classes

3

- Pouvoir faire des intersections ou des inclusions entre classes est essentiel d'un point de vue conception orientée objet.
- On veut pouvoir construire facilement des objets qui sont substantiellement similaires mais qui diffèrent sur des cas particuliers.





## Sous-classes et super-classes

4

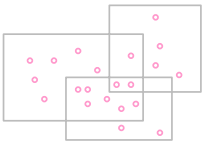
- Une classe A peut inclure toutes les instances d'une classe B.
- B est une sous-classe de A.
- A est une super-classe de B..
- Les instances de B partagent les même propriétés que A (attributs/méthodes) avec des spécificités supplémentaires.




## Appartenance à plusieurs classes

5

- Certains objets sont instances de plusieurs classes en même temps : elles ont plusieurs super-classes.
- Ces instances possèdent les propriétés de ces différentes classes.





## Inclusion des propriétés

6

- Les objets instances d'une classe donnée sont décrits par les propriétés caractéristiques de leur classe, mais également par les propriétés caractéristiques de toutes leur super-classes.
- La propriété caractéristique d'une sous-classe englobe la propriété caractéristique de toutes les superclasses.

## Les hiérarchies de classes

7

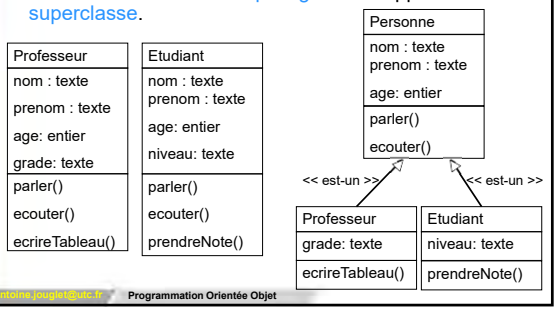
- Les hiérarchies de classes ou classifications permettent de gérer la complexité en ordonnant les objets au sein d'arborescences de classes d'abstraction croissante.
- La généralisation et la spécialisation sont des points de vue portés sur les hiérarchies de classes.



## La généralisation

8

- La généralisation consiste à factoriser les éléments communs (attributs et méthodes) d'un ensemble de classes dans une classe plus générale appelée superclasse.






## La spécialisation

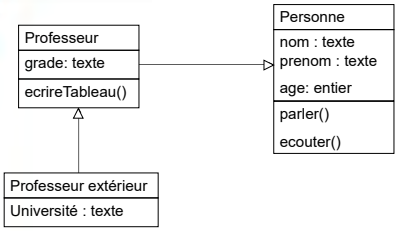
9

- La spécialisation permet d'exprimer les particularités d'un ensemble d'objets non discriminés par les classes déjà identifiées.
- Les nouvelles caractéristiques sont représentées par une nouvelle classe, sous-classe d'une des classes existantes.
- Technique efficace pour l'extension cohérente d'un ensemble de classes.



## La spécialisation

10






## Généralisation et spécialisation

11

- La généralisation et la spécialisation sont deux points de vue antagonistes du concept de classification.
- Elles expriment le sens dans lequel une hiérarchie de classe est exploitée.
- Dans toute application réelle, les 2 points de vue sont mis en application.



## Généralisation et spécialisation

12

- La généralisation est plutôt employée une fois que les éléments du domaine ont été identifiés, afin de dégager une description détachée des solutions (généralement au cours d'une conception initiale). [conception]
- La spécialisation est à la base de la programmation par extension et de la réutilisation : les nouveaux besoins sont encapsulés dans des sous-classes qui étendent les capacités de la classe de base ou spécialisent les comportements. [implémentation] [conception]

## Généralisation et spécialisation

13

- Demande des compétences différentes selon le point d'entrée dans l'arborescence.
- L'identification des superclasses fait appel à la capacité d'abstraction, indépendamment des connaissances techniques.
- La réalisation des sous-classes demandent une expertise approfondie d'un domaine particulier.
- La généralisation (spécialisation) est une relation transitive, antiréflexive, fortement antisymétrique.

antoine.jouglet@utc.fr Programmation Orientée Objet

## Classification

14

- La classification n'est pas toujours une opération triviale.
- Les classifications doivent, avant tout, bien discriminer les objets.
- Les bonnes classifications sont stables et extensibles.
- Il n'y a pas une classification mais des classifications, chacune adaptée à un usage donné.
- Une fois les critères de classification arrêtés, il faut les suivre de manière cohérente et uniforme.
- L'ordre d'application des critères est souvent arbitraire et conduit à des décompositions covariantes qui se traduisent par des modèles isomorphes.

antoine.jouglet@utc.fr Programmation Orientée Objet

## Le principe ouvert/fermé

15

- La stabilité signifie qu'une hiérarchie ne devrait pas être remise en cause dans le futur (le code existant résultant de la classification ne devrait pas être modifié par une évolution).
- L'extensibilité signifie qu'une hiérarchie devrait pouvoir évoluer facilement par spécialisation. On parle de respecter principe ouvert/fermé.
- Il s'agit du O (Open/closed) de l'acronyme SOLID.

antoine.jouglet@utc.fr Programmation Orientée Objet

1

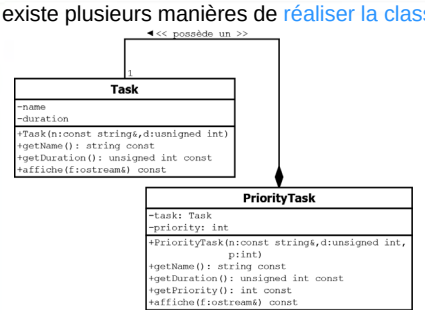
# Réaliser la classification
# L'héritage



## Réaliser la classification

2

- Il existe plusieurs manières de réaliser la classification.




- Une possibilité est la relation-client « possède-un » : la caractéristique de la super-classe est agrégée ou composée par la sous-classe.



## L'héritage

3

- En programmation objet, la technique la plus utilisée repose sur l'héritage entre classes.
- Le concept d'héritage (classes dérivées) constitue l'un des fondements de la POO.
- Il est à la base des possibilités de réutilisation de composants logiciels (en l'occurrence de classes).
- Il autorise à définir une nouvelle classe dite « dérivée » à partir d'une classe existante dite « de base », au sein d'une hiérarchie de classes.



## L'héritage

4

- La classe dérivée héritera des potentialités (attributs et méthodes) de la classe de base, tout en lui ajoutant de nouvelles fonctionnalités sans qu'il soit nécessaire de modifier la classe de base.
- Les membres de la classe de base existent (accessibles sous certaines conditions) dans la classe dérivée, comme s'ils avaient été déclarés dans la classe dérivée.
- L'héritage n'est pas limité à un seul niveau : une classe dérivée peut, à son tour, devenir classe de base pour une autre classe.
- L'héritage est alors outil de spécialisation croissante.
- En C++, l'héritage multiple est possible : une classe hérite alors des possibilités de plusieurs classes.



## Mise en œuvre de l'Héritage

5

Héritage public (cas le plus fréquent) :

```
class A { //classe de base
public :
   A();
   ~A();
};
class B : public A { // classe dérivée
public :
   B();
   ~B();
};
```



## Conséquences

6

- Quand on construit un objet de type B, il possède implicitement une partie de type A.
- Toute donnée ou méthode publique de la partie A est visible dans toute méthode de la classe B comme si ces méthodes avaient été définies dans B.

## Héritage : protection (dérivation publique)

7

- Les méthodes d'une classe dérivée n'ont pas accès aux membres privés de la classe de base.
- Les membres `protected` sont privés pour l'utilisateur de la classe tout en restant accessibles aux méthodes d'une éventuelle classe dérivée, tout en restant inaccessibles aux utilisateurs de la classe dérivée.



## Héritage : protection et amitié

8

- Lorsqu'une classe dérivée possède des fonctions amies, ces dernières disposent exactement des mêmes autorisations d'accès que les fonctions membres de la classe dérivée.
- Les déclarations d'amitié ne s'héritent pas : si `f` a été déclarée amie de `A` et si `B` dérive de `A`, `f` n'est pas automatiquement une amie de `B`.



## Héritage : construction

9

- Pour créer un objet de type B, un constructeur de B doit être défini (par défaut ou par le concepteur) :
  - La partie A d'un objet B doit être construite. Un constructeur de B doit donc faire appel au constructeur de A (implicitement ou explicitement).
  - Un constructeur de B peut faire appel explicitement à un constructeur de A et lui transmettre des informations : le mécanisme est le même que dans le cas d'initialisation d'objets membres de B (avec :).
  - Le constructeur de B peut alors être complété par ce qui est spécifique à B. L'appel au constructeur de A précède toujours ce qui est spécifique à B.



## Appel implicite/explicite du constructeur de la superclasse

10

- Si B ne comporte pas de constructeur, alors que la classe de base en comporte, le problème de la transmission des informations attendues par le constructeur de la classe de base se pose. Il doit donc y avoir un appel explicite du constructeur de la classe de base.
- Si la classe dérivée ne possède pas de constructeur (le compilateur en génère alors un par défaut), la seule situation possible est que la classe de base dispose d'un constructeur sans argument (par défaut ou défini par son concepteur) pour un appel implicite de ce constructeur (sinon il y aura une erreur à la compilation).



## Héritage : destruction

11

- Un destructeur peut être défini pour la classe dérivée B (sinon il y en aura un défini par défaut).
- Lors de la destruction d'un objet de type B :
  - il y aura automatiquement appel au destructeur de type B (généré par défaut ou non),
  - puis appel à celui de A (généré par défaut ou non).

1

# Héritage et redéfinition



## Redéfinition d'une méthode existante dans la superclasse

2

- Lorsqu'une classe B hérite d'une classe A, on peut redéfinir dans B les méthodes définies dans A.
- Ainsi, on peut spécialiser le comportement d'une méthode.
- La redéfinition des méthodes d'une classe dérivée s'étend aux attributs : on peut redéfinir un attribut de la classe de base dans la classe dérivée (pour changer son type par exemple).
- Les méthodes et les données de la classe de base seront toujours disponibles avec `classe_base::`.



## Redéfinition et masquage

3

- Lorsqu'une méthode est redéfinie dans une classe dérivée, elle masque toutes les méthodes de même nom de la classe de base (les surdéfinitions).
- Ainsi, la recherche d'une fonction surchargée se fait dans une seule portée : soit celle de la classe concernée, soit celle de la classe de base, mais jamais les deux à la fois.
- Permet d'empêcher un héritage « accidentel » de surcharges présentes dans une classe ascendante distante.



## Redéfinition et masquage

4

```
class A {
public:
  void f(int);
  void f(double);
};
```

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

```
void test(){
     B b;
     b.f(); //ok
     b.f(3); // erreur
}
```



## Rétablir la visibilité des surcharges de la superclasse

5

- Pour rétablir la visibilité des surcharges d'une méthode de la superclasse dans la classe fille, il faut utiliser une `using`-déclaration :

```
class B : public A {
public:
  using A::f;
  void f();
};
```

```
void test(){
     B b;
     b.f(); //ok
     b.f(3); //ok
}
```

- Il faut le faire pour chaque méthode de la classe fille qui surcharge ou redéfinit une méthode de la classe de base.

1

# Héritage et redéfinition
# - du constructeur de recopie
# - de l'opérateur d'affectation



## Recopie

2

- S'il n'y a pas de constructeur de recopie dans la classe B la partie de B appartenant à la classe A sera traitée comme un membre de type A.
- En particulier le constructeur de recopie de A (par défaut ou non) sera appelé.



## Recopie

3

- S'il y a un constructeur de recopie dans B : il doit prendre en charge l'intégralité de la recopie de l'objet.
- Il reste possible d'utiliser le mécanisme de transmission des informations entre constructeurs.
- En général, on souhaitera que le constructeur de recopie de A soit appelé à ce niveau.
- Dans ces conditions, on voit que ce constructeur doit recevoir en argument une partie seulement de l'objet de la classe dérivée.
- Pour cela on fait intervenir la conversion implicite d'un objet de la classe dérivée en objet de la classe de base (voir principe de substitution).



## Affectation

4

- B n'a pas surdéfini `operator=` :

L'affectation de deux objets de type B se déroule membre à membre en considérant que la partie héritée de A constitue un membre.
La partie héritée de A est traitée par l'affectation prévue dans la classe A.



## Affectation

5

- B a surdéfini `operator=` : l'opérateur `operator=` de A ne sera pas appelé. Il faut que `B::operator=` prenne en charge toute l'affectation, y compris les membres hérités de A.

Le plus facile est d'utiliser l'opérateur d'affectation de la classe A dans la surcharge de `B::operator=` en utilisant l'opérateur de résolution de portée :

```
B& B::operator=(const B& b) {
   A::operator=(b);
   /* … */
   return *this;
}
```

1

# Héritage Multiple



## Héritage multiple

2

- En C++, une classe peut hériter de plusieurs classes en même temps.
- Cette classe hérite alors des attributs et des méthodes des différentes classes.

```
class A {
};

class B {
};

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



## Héritage Multiple

3

- L'ordre d'appel des constructeurs est le suivant :
  - constructeurs des classes de base, dans l'ordre où les classes de base sont déclarées dans la classe dérivée.
  - constructeur de la classe dérivée.
- On peut distinguer l'appel d'une méthode ou d'un attribut qui porte le même nom dans deux classes mères avec l'opérateur de résolution de portée.



## Double héritage d'une même classe

4

D hérite deux fois d'une classe A par l'intermédiaire de deux classes B et C qui dérivent de A.

```
class A {
public :
  int x,y;
};

class B : public A {
};

class C : public A {
};

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



## Double héritage d'une même classe

5

- Dans ces conditions, les attributs de A (méthodes et attributs apparaissent deux fois dans D).
- En ce qui concerne les méthodes, cela est manifestement inutile (ce sont les mêmes fonctions), mais sans importance puisqu'elles ne sont pas réellement dupliquées (il n'en existe qu'une pour la classe de base).
- En revanche, les attributs (ici, x et y) seront effectivement dupliqués pour tous les objets de type D.

1

# Le principe de Substitution v.s. Conversions implicites de la classe dérivée vers la classe de base



## Le principe de substitution

2

- Le principe de substitution [Liskov, 1987] : il doit être possible de substituer n'importe quel objet instance d'une sous-classe à n'importe quel objet instance de sa superclasse sans que la sémantique du programme écrit dans les termes de la superclasse ne soit affectée, c'est-à-dire sans altérer les propriétés désirables du programme concerné.
- Toutes les propriétés de la classe de base doivent être valables intégralement pour la classe dérivée :
  - une sous-classe ne peut pas avoir des préconditions plus fortes que celles de sa superclasse;
  - une sous-classe ne peut pas avoir des postconditions plus faibles que celles de sa superclasse.
- Un besoin d'hériter partiellement est le signe que la relation d'héritage considérée ne réalise pas vraiment une relation de classification.
- Il s'agit du L de l'acronyme SOLID (Liskov substitution).



## Substitutions/Conversions en C++

3

- En pratique, les substitutions sont mises en œuvre à travers les conversions implicites d'un objet de la classe dérivé en objet de la classe de base.
- Lors d'une dérivation publique, il y a alors existence de conversions implicites :
  - d'un objet d'un type dérivé dans un objet de type de base;
  - d'un pointeur ou d'une référence sur une classe dérivée vers un pointeur ou une référence sur une classe de base.



## Conversion d'un objet en objet d'une classe parent

4

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

A a;
B b;
a=b;
```

Seule la partie « A » contenue dans l'objet b est alors copiée dans a.

Il y a perte de l'information spécifique à la classe B.

Ne permet pas de respecter le principe de substitution par rapport aux appels de méthodes.



## Pointeurs, références et héritage

5

- Un pointeur ou une référence d'un type d'objet peut recevoir l'adresse d'un objet d'une classe descendante :

```
A a; B b;
A* pta=&a; B* ptb=&b; pta=ptb;
```

- L'opération inverse est illégale mais est réalisable avec un opérateur de cast (voir cast dynamique).

```
A& refa=b; //est légale
B& refb=a; //est illégale mais possible avec cast dynamique
B& refb=dynamic_cast<B&>(a);
B* ptb=dynamic_cast<B*>(&a);
```



## Conversions et principe de substitution

6

- Les conversions implicites mises en place par le compilateur ne sont, en général, pas suffisantes pour respecter le principe de substitution.
- En effet, l'appel d'une méthode (surchargée pour la classe de base et la classe dérivée) pour l'objet pointé conduit systématiquement à appeler la méthode correspondante au type du pointeur et non au type de l'objet pointé effectivement.
- Il y a **ligature statique** (typage statique). Le type d'un objet pointé est alors interprété lors de la compilation :

  `pta->f();` et `refa.f();`

  appellent toujours `A::f();`

1

# Le polymorphisme



## Le polymorphisme

2

- Le terme polymorphisme décrit la caractéristique d'un élément qui peut prendre plusieurs formes, comme l'eau qui se trouve à l'état solide, liquide ou gazeux.
- En informatique, le polymorphisme désigne un concept orienté objet selon lequel un nom d'objet peut désigner des instances de classes différentes issues d'une même arborescence en assurant une reconnaissance dynamique du type réel de l'objet (en ce qui concerne l'appel des méthodes).



## Pointeurs, références et héritage

3

- Pour obtenir l'appel de la méthode correspondant au type de l'objet pointé (référencé), il est nécessaire que le type réel de l'objet ne soit pris en compte qu'au moment de l'exécution (le type de l'objet désigné par un même pointeur ou une référence pourra varier au fil du déroulement du programme).
- On parle de **ligature dynamique** ou de typage dynamique ou encore de **polymorphisme**.



## Les méthodes virtuelles

4

- Pour permettre la ligature dynamique d'une méthode à partir d'un pointeur ou d'une référence de la classe de base, il suffit que cette fonction soit déclarée `virtual` dans la classe de base.
- Indique au compilateur que les éventuels appels de la méthode à partir d'un pointeur ou d'une référence doivent utiliser une ligature dynamique.
- Implique la mise en place d'un dispositif par le compilateur permettant de n'effectuer le choix de la méthode qu'au moment de l'exécution de la méthode (choix basé sur le type exact de l'objet ayant effectué l'appel).



## Les méthodes virtuelles

5

- Dans les classes dérivées il n'est pas nécessaire de déclarer la méthode comme virtuelle. L'information serait redondante.
- À partir du moment où une méthode `f` a été déclarée `virtual` dans une classe `A`, elle sera soumise à la ligature dynamique dans `A` et dans toutes les classes descendantes de `A`.
- Une méthode virtuelle ne peut pas être inlinée dans les appels nécessitant une ligature dynamique (appels à partir d'un pointeur ou d'une référence).



## Les méthodes virtuelles

6

- La redéfinition d'une méthode virtuelle n'est pas obligatoire.
- On peut surcharger une méthode virtuelle, chaque surcharge pouvant être virtuelle ou non.
- Si on a défini une méthode virtuelle dans une classe et qu'on la surcharge dans une classe dérivée avec des arguments différents, il s'agira alors bel et bien d'une autre méthode. Si cette dernière n'est pas déclarée virtuelle, elle sera soumise à une ligature statique. De plus, la règle sur la recherche d'une fonction surchargée dans une seule portée est toujours valable.

## Valeurs covariantes

7

- La redéfinition d'une méthode virtuelle doit utiliser exactement le même type de retour.
- Il existe une exception pour les valeurs de retour covariantes : il s'agit du cas où la valeur de retour d'une méthode virtuelle est un pointeur ou une référence sur la classe de base.
- La redéfinition de cette méthode virtuelle dans une classe dérivée peut alors se faire avec un pointeur ou une référence de cette classe dérivée.



## Les méthodes virtuelles : mécanisme

8

- D'une manière générale, lorsqu'une classe comporte au moins une méthode virtuelle, le compilateur lui associe une table contenant les adresses de chacune des méthodes virtuelles correspondantes.
- Tout objet d'une classe comportant au moins une méthode virtuelle se voit attribuer par le compilateur, outre l'emplacement mémoire nécessaire à ses membres données, un emplacement supplémentaire de type pointeur contenant l'adresse de la table associée à sa classe.



## Le principe Ouvert / Fermé

9

- une classe doit être extensible par héritage (ouverte)
- sans remettre en cause le code existant (fermé)



## Le polymorphisme mis en œuvre par le programmeur

10

- On peut mettre en oeuvre le polymorphisme à la main :

```
void fx(A& t){
  B* pt=dynamic_cast<B*>(&t);
  if (pt!=0) pt->affiche(); // ligature à  B::affiche
  else t.affiche(); // ligature à A::affiche
}
```

- Mais, on ne respecte pas le principe ouvert/fermé :
- Si on ajoute une classe qui hérite de `A` ou de `B` (on modifie la hiérarchie issue de `A`), on doit remettre en cause le code de la fonction `fx`.



## Choix du caractère virtuel des méthodes

11

- Seule une méthode (fonction membre d'une classe) peut être virtuelle.
- Un constructeur ne peut pas être virtuel.
- En revanche, un destructeur peut être virtuel. Dans une classe qui peut être sous-classée, le destructeur devrait toujours être virtuel (principe de substitution).
- Une méthode susceptible d'être redéfinie dans une sous-classe devrait toujours être virtuelle afin de respecter le principe de substitution.
- Une méthode non-virtuelle ne devrait jamais être redéfinie pour respecter le principe de substitution.

1

# Classes abstraites



## Classes abstraites

2

- On peut définir des classes destinées non pas à instancier des objets, mais simplement à donner naissance à d'autres classes qui seront construite sur le même « modèle d'interface ».
- En POO, on appelle cela des **classes abstraites**.
- Ce modèle sera « instancié » au travers de classes qui seront créées par héritage à partir de la classe abstraites.
- Par opposition à la classe abstraite, nous dirons que ces classes sont des « classes concrètes ».



## Méthodes abstraite

3

- Ce modèle est défini par la déclaration de méthodes dont le comportement ne sera pas défini : on parlera de « méthodes virtuelles pures » ou de « méthodes abstraites ».
- La propriété abstraite est donc appliquée par extension à des méthodes.
- Ce sont les méthodes dont l'interface a été déclarée mais dont le comportement n'a pas été défini : elles ne possèdent pas de corps.
- La définition du comportement d'une méthode virtuelle pure est « retardée dans les sous-classes » de ce modèle.



## Classes abstraites

4

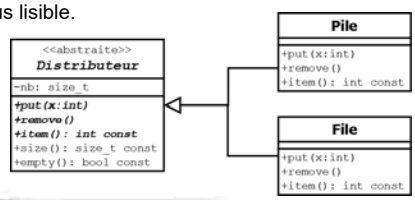
- Les classes abstraites ne peuvent pas être instanciées directement : elles ne donnent pas naissance à des objets.
- En effet le comportement des méthodes abstraites n'est pas encore connu.
- Elles servent de spécification générale pour manipuler les objets instances des sous-classes : elles permettent de définir des mécanismes généraux.




## Classes abstraites en UML

5

- En UML, le nom d'une classe abstraite est mis en italique.
- Il en est de même pour les méthodes virtuelles pures.
- Cependant, il est préférable d'ajouter la propriété « abstraite » après le nom de la classe et dans la déclaration des méthodes virtuelles pures pour rendre cet aspect plus lisible.






## Classes abstraites en C++

6

- En C++, une classe abstraite est une classe qui contient au moins une méthode abstraite dite **fonction virtuelle pure**.
- Les fonctions virtuelles pures sont des méthodes dont la définition est nulle.
- La méthode est déclarée mais elle ne possède pas de corps. Il est remplacé par « `=0` » juste après l'entête.
- Il n'est plus possible de déclarer des objets de cette classe (elle n'est pas instanciable).
- Une fonction déclarée virtuelle pure dans une classe de base doit être définie dans une classe dérivée, sinon la classe dérivée est aussi abstraite.

## Classes abstraites : intérêts

7

- Le principal intérêt de cette démarche est :
  - réduire le niveau de détails dans les descriptions des sous-classes;
  - retarder l'implémentation de certaines méthodes dont on connait l'existence conceptuellement au niveau de la classe abstraite mais que l'on ne peut entièrement préciser que dans les sous-classes.



## Classes abstraites : avantages

8

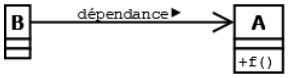
- Les classes abstraites facilitent l'élaboration de logiciels génériques, aisément extensibles par sous-classement (héritage).
- L'ensemble des mécanismes qui servent de modèle pour les fonctions des applications est construit à partir des éléments généraux fournis par les classes abstraites.
- Les spécificités et les extensions sont encapsulées dans des sous-classes concrètes.



## Dépendances entre classes

9

- Lorsqu'une instance d'une classe B appelle une méthode `f()` sur une instance d'une classe A, il y a une association qui représente une dépendance de B vis-à-vis de A à l'exécution.



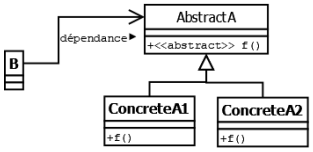

- Cette dépendance peut avoir deux niveaux :
  - celle vis-à-vis de l'interface de A;
  - celle vis-à-vis de l'implémentation du comportement de A.
- Quand il y a dépendance de B vis-à-vis de l'implémentation de A, les modules A et B ne peuvent pas évoluer séparément : un changement du module A implique une recompilation de A aussi bien que de B.



## Inversion des dépendances

10

- L'utilisation des classes abstraites permet de casser la dépendance au code source.
- Pour cela, il suffit de définir une classe abstraite `AbstractA` pour la fonctionnalité remplie par A et de rendre B dépendante de l'interface `AbstractA` et non plus d'une implémentation.
- Les différentes implémentations de `AbstractA` qui existeront au travers des différentes classes concrètes A peuvent alors évoluer indépendamment de B.




- Il s'agit du D (Dependancy inversion) de l'acronyme SOLID.

1

# Héritage
# public, protected et private



## Héritage `public`

2

- Dérivation `public`: si la classe B hérite de la classe A, l'accès aux membres publiques de A est autorisée à l'utilisateur de la classe B.
- Quand elle est correctement utilisée, la dérivation `public` implique une relation **est_un** entre B et A : un objet B est aussi un objet A (principe de substitution).



## Héritage `private` et `protected`

3

- Dérivation `private` ou `protected` : si la classe B hérite de la classe A, l'accès aux membres publics de A est interdit à l'utilisateur de la classe B (mais reste permis par les méthodes de B).
- Ceci a un intérêt lorsque :
  - toutes les fonctions utiles de la classe de base ont été redéfinies dans la classe dérivée et qu'il n'y a aucune raison de laisser l'utilisateur accéder aux anciennes.
  - que l'on adapte l'interface d'une classe de base, de manière à répondre à certaines exigences (souvent sémantiques).



## Héritage `private` et `protected`

4

- Les membres protégés de A restent accessibles aux fonctions membres et aux fonctions amies de B.
- Dérivation `private` : ils seront considérés comme privés pour toute classe dérivée de B.
- Dérivation `protected` : les membres publics et protégés de A seront considérés comme protégés pour toute classe dérivée de B.
- Il est possible dans une dérivation privée ou protégée de laisser public un membre de la classe de base en le redéclarant explicitement dans l'espace public.



## « est_implémentée_en_terme_de »

5

- la dérivation `private` (`protected`) signifie une relation **est_implementée_en_terme_de** entre B et A (et non plus une relation est_un) :
  - la classe B est implémentée en utilisant la classe A.
  - le but est de réutiliser avantageusement des composants de la classe A (sans qu'il y ait pour autant une relation conceptuelle entre A et B).
- L'utilisation de l'héritage `private` (`protected`) est donc une technique d'implémentation (et non de conception).
- L'héritage privé signifie un héritage d'implémentation : l'interface de la classe de base est ignorée.
- Une alternative (souvent préférable) pour implémenter la relation est_implémentée_en_terme_de est d'utiliser la composition (voir design pattern Adapter).



## Conséquences

6

- Le principe de substitution n'est plus à l'œuvre dans le cadre de l'héritage privé.
- Un objet de la classe dérivée ne peut plus être converti en objet de la classe de base.
- Il en est de même pour les conversions de pointeurs ou de références de la classe dérivée vers des pointeurs ou des références de la classe de base.
- Le polymorphisme n'a alors plus de sens dans ce contexte.

1

# Transtypage



## Cast dynamique

2

- Si on sait qu'un pointeur de type `A*` (ou une référence de type `A&`) pointe en fait sur un objet de type `B`, on peut utiliser le cast dynamique pour le convertir en `B*`, (respectivement en `B&`).

```
B obj;
A* pt=&obj;
A& ref=obj;
B* pt2=dynamic_cast<B*>(pt);
B& ref2=dynamic_cast<B&>(ref);
```



## Cast dynamique

3

- L'opérateur `dynamic_cast` ne peut être utilisé que dans un contexte de polymorphisme, c'est-à-dire qu'il doit exister au moins une méthode virtuelle dans la classe de base.
- L'opérateur `dynamic_cast` aboutit si l'objet réellement pointé est d'un type identique ou d'un type descendant au type d'arrivée demandé.
- Lorsque l'opérateur n'aboutit pas :
  - il fournit le pointeur nul s'il s'agit d'une conversion de pointeur,
  - il déclenche une exception `bad_cast` s'il s'agit d'une conversion de référence.



## Autres opérateurs de cast

4

- A noter l'existence d'autres opérateurs de cast qui devraient être utilisés dans tous les autres cas (et qui sont beaucoup plus fiables que les opérateurs de cast issus du C):
- **`static_cast`** pour les conversions indépendantes de l'implémentation.
- **`reinterpret_cast`** pour les conversions dont le résultat dépend de l'implémentation (par ex conversions entiers vers pointeurs)
- **`const_cast`** pour ajouter ou supprimer à un type le modificateur `const` (les types de départ et d'arrivée ne devant différer que par `const`).

1

# Identification dynamique de type (Run Time Type Identification)



## Run Time Type Identification

2

- Des structures de données sont mises en place par le compilateur pour connaître le type réel des objets dans des situations de polymorphisme.
- Ces informations ne sont générées par le compilateur que pour les classes polymorphiques (contenant au moins une méthode virtuelle) et seulement si l'option Run Time Type Information du compilateur est utilisée (c'est en général le cas par défaut).



## Type d'un objet

3

- Il est possible lors de l'exécution de connaître le véritable type d'un objet même s'il est désigné par un pointeur ou une référence d'une classe mère.
- L'opérateur unaire **`typeid`** prend un objet en argument et fournit en résultat un objet de type prédéfini **`type_info`** contenant des infos sur le type.
- On ne peut pas instancier directement un objet `type_info` car les constructeurs sont privés.
- Le seul moyen d'obtenir un tel objet est d'utiliser l'opérateur `typeid`.
- De même, on ne peut pas faire d'affectation entre objets `type_info` (on ne peut donc pas stocker l'information).



## Type d'un objet

4

- Utiliser le fichier d'entête `<typeinfo>` (namespace `std`).
- La classe `type_info` contient la fonction membre `name()`, laquelle fournit une chaîne de caractères `const char*` représentant le nom du type. Ce nom qui n'est pas imposé par la norme dépend de l'implémentation.
- De plus, la classe dispose de l'opérateur binaire `operator==` qui permet de comparer les deux types.



## Exemple

5

```
Class A {
… // au moins une méthode virtuelle…
};
Class B : public class A{
…
};
A a; std::cout<<typeid(a).name();//A
B b; std::cout<<typeid(b).name();//B
A* pt=&b; std::cout<<typeid(pt).name();//A*
          std::cout<<typeid(*pt).name();//B
A& ref=b; std::cout<<typeid(ref).name();//B
```

1

# Héritage

# Interface et comportement Problématiques



## Interface et comportement

2

Deux parties dans une méthode de classe :

- son interface (son prototype) qui indique comment utiliser la méthode en précisant le type de sa valeur de retour ainsi que celui de ses paramètres;
- son comportement qui correspond aux instructions exécutées lorsque l'on applique la méthode sur un objet.



## Méthode non-virtuelle : intentions du programmeur

3

- But de déclarer une méthode non-virtuelle : avoir des classes dérivées qui héritent de l'interface de cette méthode avec une implémentation obligatoire.
- En effet, quand une méthode est non-virtuelle, elle n'est pas supposée avoir un comportement différent dans les classes dérivées puisque la méthode n'est pas polymorphique.
- Non respect de cette règle = manquement au principe de substitution.
- Une méthode non-virtuelle spécifie un invariant durant une spécialisation : le comportement n'est pas supposé changer quelque soit le degré de spécialisation.
- Une méthode non-virtuelle ne doit pas être redéfinie dans les sous-classes.



## Méthode virtuelle : intention du programmeur

4

- But de déclarer une méthode virtuelle (non pure) : avoir des classes dérivées qui héritent :
  - de l'interface de cette méthode,
  - d'un comportement par défaut.
- Le comportement peut être éventuellement redéfini pour obtenir un comportement spécialisé.
- Le polymorphisme permet d'obtenir le bon comportement dans tous les contextes où un objet d'une classe dérivée se substitue à un objet de la classe de base.



## Méthode virtuelle pure : intention du programmeur

5

- But de déclarer une méthode virtuelle pure (abstraite) : avoir des classes dérivées qui héritent seulement de l'interface.
- La définition du comportement est retardé et spécialisé dans les classes descendantes concrètes.
- Le comportement d'une méthode virtuelle pure doit être défini dans une sous-classe.



## Résumé

6

Si une classe B hérite d'une classe A :

- on dit que la classe B hérite de l'interface et du comportement (qui ne devrait pas être redéfini) des méthodes non-virtuelles de A;
- on dit que la classe B hérite de l'interface et d'un comportement par défaut (qui peut être redéfini) des méthodes virtuelles de A;
- on dit que la classe B hérite de l'interface des méthodes virtuelles pures de A (le comportement de ces méthodes doit être défini dans B si c'est une classe concrète).

## Restreindre les possibilités, éviter les erreurs ?

7

- Comment faire en sorte qu'une classe ne puisse plus être spécialisée ?
- Comment faire en sorte qu'une méthode virtuelle ne puisse plus être redéfinie dans les classes filles d'une classe ?
- Comment éviter que le développeur d'une classe fille redéfinisse par erreur (de conception) une méthode non virtuelle d'une classe mère ?



## Héritage partiel de comportement ?

8

- L'héritage du comportement fonctionne en tout ou rien :
  - soit il est correct pour la sous-classe,
  - Soit il doit être entièrement redéfini.

  Comment faire en sorte d'hériter de l'interface d'une classe et seulement partiellement du comportement d'une méthode ?

  Comment factoriser ce qui est commun tout en laissant la possibilité de spécificités ?



## Hériter uniquement du comportement ?

9

- L'héritage d'interface est différent de l'héritage de comportement.
- Dans le cadre de l'héritage public, les classes héritent toujours de l'interface.
- Il arrive que dans certains cas, on souhaite hériter du comportement pour le réutiliser alors que l'interface n'est plus adaptée au contexte.
- Comment faire pour que B hérite des méthodes et des attributs de A mais pas de son interface ?

1

# Héritage : spécificateurs de redéfinition et d'héritage (C++11)



## Restreindre les possibilités, éviter les erreurs

2

- Comment faire en sorte qu'une classe ne puisse plus être spécialisée ?
- Comment faire en sorte qu'une méthode virtuelle ne puisse plus être redéfinie dans les classes filles d'une classe ?
- Comment éviter que le développeur d'une classe fille redéfinisse par erreur (de conception) une méthode non virtuelle d'une classe mère ?



## final

3

- Le mot clé `final` est utilisé pour indiquer qu'une classe ne peut plus être spécialisée :

```
class A {
    /*…*/
};

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

class C : public B {
    // erreur, B ne peut pas être spécialisée
};
```



## final

4

- Le mot clé `final` est aussi utilisé pour indiquer qu'une méthode virtuelle ne peut plus être redéfinie dans les classes filles.

```
class A {
    virtual void f();
};
class B : public A {
    void f() final; // f redéfinie dans B
};
class C : public A {
    void f(); // Erreur f, ne peut plus être redéfinie
};
```



## override

5

- Ce mot clé `override` est utilisé pour indiquer explicitement (mais il n'est pas obligatoire) qu'une méthode sera une redéfinition d'une méthode qui existe déjà dans une classe mère.
- Permet au compilateur de détecter certaines erreurs d'écriture.
- Une utilisation systématique du mot clé lorsque l'on fait une redéfinition permet d'éviter de redéfinir par erreur une méthode non virtuelle.
- Permet à un utilisateur de comprendre qu'il existe une version précédente de la méthode dans une classe mère.



## override

6

```
struct A {
  virtual void f() const;
  void g();
};

struct B : A {
  void f() override; /* Erreur, il n'existe pas une
  telle méthode dans la classe A. */

  void f() const override; /* OK, la classe A
  contient une méthode avec le même prototype */

  void g() override; /* Erreur, B::g n'est pas
  (toujours) une redéfinition de A::g car A::g n'est
  pas virtuelle */
};
```

1

# Héritage virtuel



## Double héritage d'une même classe

2

D hérite deux fois d'une classe A par l'intermédiaire de deux classes B et C qui dérivent de A.

```
class A {
public :
  int x,y;
};

class B : public A {
};

class C : public A {
};

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



## Double héritage d'une même classe

3

- Dans ces conditions, les attributs de A (méthodes et attributs apparaissent deux fois dans D).
- En ce qui concerne les méthodes, cela est manifestement inutile (ce sont les mêmes fonctions), mais sans importance puisqu'elles ne sont pas réellement dupliquées (il n'en existe qu'une pour la classe de base).
- En revanche, les attributs (ici, x et y) seront effectivement dupliquées pour tous les objets de type D.



## Double héritage d'une même classe

4

- Y a-t-il redondance ? La réponse dépend du problème !
- Si l'on souhaite que D dispose de deux jeux de données (de A), on ne fera rien de particulier et on se contentera de les distinguer à l'aide de l'opérateur de résolution de portée : `A::B::x` et `A::C::x`, ou éventuellement, si B et C ne possèdent pas de membre x : `B::x` et `C::x`.



## Dérivation virtuelle

5

- On peut demander à C++ de n'incorporer qu'une seule fois les membres de A dans la classe D.
- Il faut le préciser dans les déclarations des classes B et C (pas dans celle de D !) que la dérivation à partir de A est virtuelle :

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



## Dérivation virtuelle

6

- Signifie que A ne devra être introduite qu'une seule fois dans les descendants éventuels de B ou C.
- Cette déclaration n'a donc pas d'effet sur les classes B et C elles mêmes.
- Avec ou sans le mot clé `virtual`, les classes B et C, se comportent de la même manière tant qu'elles n'ont pas de descendant.
- Le mot clé `virtual` peut être placé indifféremment avant ou après `public` ou `private`.
- Si B et C, dérivent virtuellement de A il n'y a qu'une seule partie A construite dans un objet D.
- Le choix des informations à fournir au constructeur de A a lieu non plus dans B ou C, mais dans D.
- C++ autorise (uniquement dans le cas de " dérivation virtuelle ") à spécifier, dans le constructeur de D, des informations destinées à A.

1

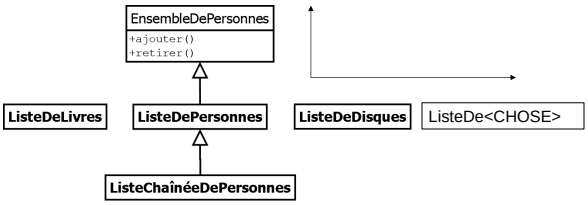
# La programmation générique



## Généralisation horizontale de type

2

- La généralisation verticale (par ex. au travers de l'héritage) permet de construire une hiérarchie de classes réutilisables.
- La généralisation horizontale consiste à rendre un concept indépendant du type des données manipulées






## Type de donné abstrait paramétré

3

- Il devient alors nécessaire de pouvoir proposer des types abstraits de données (des classes) qui peuvent être paramétrées par des types.
- L'ensembles des classes `liste_de_livres`, `liste_de_personnes`, `liste_d_entiers`, `liste_de_disques` pourrait alors être écrit: `liste_de<CHOSE>` où `CHOSE` représente un type arbitraire que l'on appelle paramètre formel générique et qui prend ici une valeur dans `{livres, personnes, entiers, disques}` appelés paramètres génériques réels.



## Programmation générique

4

- La *programmation générique* permet de créer des modèles de fonctions ou de classes en précisant que certains types ou certaines données sont des paramètres.
- Il s'agit donc de spécifier des instructions sur une donnée sans en connaître le type au moment de l'écriture de ces instructions.
- Elle permet alors l'implémentation d'algorithmes ou de modules indépendamment du type des données qui sont manipulées.
- Par exemple, cela permet l'implémentation d'un algorithme de tri indépendant du type des objets triés et de la fonction de comparaison qui peut différer d'un type d'objet à l'autre et de l'ordre souhaité.



## Typage fort v.s. programmation générique

5

- Il faut préserver les bénéfices de la sûreté de type via la déclaration explicite de types :
  - lisibilité : les déclarations explicites de type indiquent clairement l'utilisation attendue de chaque élément.
  - fiabilité : un compilateur peut détecter des utilisations non possibles avant d'exécuter le code.
- Le programmeur qui souhaite utiliser un modèle de classe ou de fonction générique est obligé de préciser l'ensemble des paramètres de type et de valeur dans tous les contextes d'utilisation.
- Les modèles sont donc « instanciés » à la compilation en précisant les types réels qui doivent être utilisés à la place des paramètres.
- Une implémentation spécifique à ces paramètres est alors compilée.



## Module générique

6

- Un type abstrait de données paramétré avec un paramètre formel générique est appelé module générique.
- Le processus d'obtention d'un module réel à partir d'un module générique s'appelle une dérivation générique (ou instanciation générique).

## Programmation générique en C++

7

- Basée principalement sur la notion de `template`.
- Permet de créer des modèles de fonctions ou de classes en précisant que certains types sont des paramètres.
- Ces modèles sont « instanciés » à la compilation en précisant les types réels qui doivent être utilisés à la place des paramètres : une implémentation spécifique à ces paramètres est alors compilée.
- C'est la première forme (et l'outils de base) de la méta-programmation en C++ : on crée des programmes qui vont donner naissance à d'autres programmes (avec le compilateur).



## Les patrons en C++

8

- La mention **`template`**`<class T>` précise que l'on a affaire à un **patron** dans lequel apparaît un paramètre de type nommé T.
- Dans la définition d'un patron, on utilise le mot clé **`class`** pour indiquer un type quelconque (classe ou non).
- La norme a introduit le mot clé **`typename`** qui peut (entre autres choses) se substituer à **`class`**.



## Les patrons en C++

9

- Les instructions de définition d'un patron ressemblent aux instructions habituelles mais en utilisant des types inconnus au moment de l'écriture du code.
- Ces instructions sont utilisées par le compilateur pour fabriquer (instancier) chaque fois qu'il est nécessaire les instructions correspondant à la fonction requise;



## Exemple d'une fonction patron

10

```
template<class T>
T min(T a, T b){
  if (a<b) return a;
  return b;
}

/* min est utilisable avec tout type T pour lequel
   l'opérateur < a été défini */
void main(){
  int n=4, p=12;
  float x=2.5, y=3.25;
  std::cout<<min(n,p)<<'\n';
  std::cout<<min(x,y)<<'\n';
  // comparaison d'adresses
  std::cout<<min(&x,&y)<<'\n';
}
```



## Avantages et conséquences

11

- Ce style de programmation permet de se concentrer sur les problèmes algorithmiques.
- La « distance sémantique » entre un algorithme et son implémentation diminue.
- Favorise une très grande réutilisabilité (un des grands but de la POO) :
  - Permet la programmation des structures de données de base de façon générique (STL).
  - Augmente la fiabilité des codes développés à partir de ces structures de données.

1

# Définir des nouveaux patrons en C++



## Définir des patrons utilisables dans plusieurs fichiers sources

2

- Les patrons sont des déclarations : leur présence est toujours nécessaire et il n'est pas possible de créer un module objet correspondant à un patron.
- En pratique, on placera les définitions de patron dans un fichier d'entête « .h » quand les patrons doivent être utilisées dans plusieurs fichiers sources.
- Deux inconvénients majeurs :
  - compilation lente car la définition doit être relue par le compilateur à chaque fois qu'un patron est utilisé;
  - ne permet pas de protéger le savoir faire des entreprises qui éditent des bibliothèques patron puisque leur code est accessible à tout le monde.



## Paramètres d'un patron

3

- Les paramètres d'un patron se déclarent en utilisant le mot clé **template**. Les paramètres sont déclarés entre les chevrons <> et séparés par des virgules.
- Il est possible de déclarer deux types de paramètre :
  - des paramètres de type,
  - des paramètres expressions.
- Les paramètres de type d'un patron sont précédés du mot clé **class** ou du mot clé **typename** (les deux notations sont équivalentes).

```
template<typename T1, typename T2> /* … patron …*/
```

- Les paramètres expression (similaires aux paramètres de fonctions) indiquent des constantes d'un type donné dont les valeurs seront connue au moment de l'instanciation du patron.

```
template<int val1, double val2> /* … patron …*/
```

- Les paramètres de type et expression d'un patron peuvent être mélangés.

```
template<typename T, int n> /* … patron …*/
```



## Instanciation d'un patron

4

Pour utiliser un patron, il suffit de placer après l'identificateur de la classe ou de la fonction les paramètres effectifs que l'on souhaite utiliser entre chevrons <> et séparés par des virgules :

```
template<class T, int n> void fx(const T& z) {
        cout << z <<","<< x << "\n";
}
template<class T, int n> class tableau{
        T tab[n]; //...
};
void main(){
        // instanciation du patron de fonction
        // avec T=double et n=4
        fx<double,4>(3.14);
        // instanciation du patron de classe
        // avec T=int et n=10
        tableau<int,10> mon_tab;
}
```



## Paramètres par défaut

5

Dans la définition d'un patron, il est possible de fournir des valeurs par défaut pour certains paramètres, suivant un mécanisme semblable à celui utilisé pour les paramètres de fonctions usuelles, en attribuant des valeurs en partant du paramètre le plus à droite :

```
template<class T, int n=0> void fx(const T& z) { /*...*/ }

template<class T, class U=float, int n=3> class A { /*...*/ };

void f(){
        fx<int,4>(18); // n=4
        fx<double>(3.14); // n=0
        A<int,char> a1; //n=3
        A<int> a2; //U=float, n=3
}
```



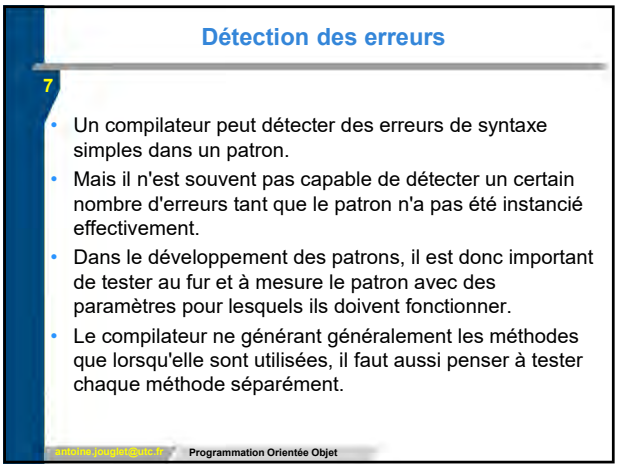
## Types utilisables pour l'instantiation

6

- À un paramètre de type peut théoriquement correspondre n'importe quel type effectif (standard ou classe).
- Cependant plusieurs éléments peuvent intervenir pour contraindre les instanciations possibles.
- On peut imposer qu'un paramètre de type corresponde à un pointeur (T*):

```
template<class T> void fct(T* x);
```

- Dans la définition d'un patron peuvent apparaître des instructions qui s'avéreront incorrectes lors de la tentative d'instanciation pour certains types.


Détection des erreurs
7
• Un compilateur peut détecter des erreurs de syntaxe simples dans un patron.
• Mais il n'est souvent pas capable de détecter un certain nombre d'erreurs tant que le patron n'a pas été instancié effectivement.
• Dans le développement des patrons, il est donc important de tester au fur et à mesure le patron avec des paramètres pour lesquels ils doivent fonctionner.
• Le compilateur ne générant généralement les méthodes que lorsqu'elle sont utilisées, il faut aussi penser à tester chaque méthode séparément.
antoine.jouglet@utc.fr
Programmation Orientée Objet

1

# Patrons de fonctions



## Fonctions patron

2

- On distingue les
  - les paramètres du patron : les paramètres déclarés avec `template`;
  - les paramètres ordinaires.
- Les deux types de paramètre sont mélangés dans l'entête d'une fonction patron :

```
template<class T>
T somme(const T* tab, size_t n) {...}
```



## Fonctions patron

3

- Les paramètres de type peuvent intervenir à n'importe quel endroit de la définition d'un patron :
  - dans l'entête;
  - dans des déclarations de variables locales (éventuellement de l'un des types de paramètres);
  - dans des instructions exécutables (par ex new, `sizeof(...)`).



## Déduction automatique des types

4

- Pour instancier un patron, il suffit de préciser entre chevrons les valeurs des paramètres :

```
void f() {
      int tab[5]={1,2,3,4,5};
      int res=somme<int>(tab,5);
}
```

- Cependant, le compilateur est souvent de capable de déduire avec quels types il doit instancier les paramètres de type avec un mécanisme qui est similaire à celui utilisé par le mot clé `auto` :

```
void f(){
      int tab[5]={1,2,3,4,5};
      int res=somme(tab,5);
}
```



## Déduction automatique des types

5

- Pour que les types soient déduits par le compilateur, il est nécessaire que chaque paramètre de type apparaisse au moins une fois dans les paramètres expression du patron avec une exacte correspondance de type.

```
template<class T> T min(T a, T b){
      if (a<b) return a;
      return b;
}
void f(){
      int i=1; char c=18;
      min(i,i); // ok
      min(i,c); // erreur
      min(c,i); // erreur
}
```

- Il est possible de réintervenir sur le mécanisme d'identification de type en instanciant les paramètres de type pour lever les ambiguïtés.



## Surcharge des patrons de fonction

6

- Il est possible de surcharger un patron de fonctions.

```
template<class T> T somme(T a, T b) { return a + b; }
template<class T> T somme(T* a, T b) { return *a + b; }
template<class T> T somme(T a, T* b) { return a + *b; }
template<class T> T somme(T* a, T* b) { return *a + *b; }
template<class T> T somme(T a, T b, T c) { return a + b + c; }

void main(){ int n=12, p=15;
  somme(n,p); // patron 1
  somme(&n,p); // patron 2
  somme(n,&p); // patron 3
  somme(&n,&p); // patron 4
  somme(n,n,n); // patron 5
}
```

## Ambiguïtés sur le choix des patrons de fonctions

7

Comme pour les surcharges de fonction, le compilateur ne doit pas se trouver face à plusieurs choix équivalents créant une ambiguïté.

- Par exemple, on ne peut pas trouver deux patrons, l'un correspondant à une transmission par valeur, l'autre à une transmission par référence :

```
template<class T> void fct(T a) { ... }
template<class T> void fct(T& a) { ... }
void main(){ int n; fct(n); /* ambiguïté ! */}
```

- On ne peut pas non plus trouver deux patrons, l'un correspondant à une transmission (par valeur, référence ou adresse) non-`const`, l'autre à une transmission (par valeur, référence ou adresse) `const` :

```
template<class T> void fct(T a) { ... }
template<class T> void fct(const T a) { ... }
void main(){
  int n;
  fct(n); // erreur
}
```

antoine.jouglet@utc.fr Programmation Orientée Objet

## Ambiguïtés sur le choix des patrons de fonctions

8

Par contre, deux patrons l'un correspondant à une transmission par référence non-const (resp. un pointeur non-const), et l'autre correspondant à une référence-const (resp. un pointeur const) sur un même paramètre de type peuvent coexister.

- Le premier sera utilisé en cas de correspondance avec des lvalues non-const (resp. des valeurs de type pointeur non-const) et l'autre pour des lvalues const (resp. des valeurs de type pointeur const) :

```
template<class T> void fct(T& a) { ... } // patron 1
template<class T> void fct(const T& a) { ... } // patron 2

void f(){
      int n;
      fct(n); // ok patron 1
      const double pi=3.14;
      fct(pi); // ok patron 2
}
```

antoine.jouglet@utc.fr Programmation Orientée Objet

1

# Patrons de classes



## Patron de classe

2

Pour créer un patron de classe, il suffit de faire précéder la définition de la classe par une instruction `template<…>` qui contiendra la liste des paramètres (comme pour les fonctions).

- Tous les paramètres doivent être explicitement précisés lors de l'instanciation du patron.
- Il n'existe pas de mécanisme de déduction automatique des types comme celui qui existe pour les patrons de fonction.
- Il n'est pas possible de surcharger un patron de classe contrairement aux patrons de fonctions. Il sera par contre possible de le spécialiser.



## exemple

3

```
template<class T, size_t n> class Tableau {
  T tab[n];
public:
  Tableau(T v=T()){ for(size_t i=0; i<n; i++) tab[i]=v; }
  void affiche() const;
};

// déf. d'une méthode en dehors de la déf. du patron
template <class T, size_t n>
void Tableau<T,n>::affiche() const {
  for(size_t i=0; i<n; i++) std::cout<<tab[i]<<"\n";
}

inline void f(){
  Tableau<int,10> t1(0);
  Tableau<double,5> t2;
  t2.affiche();
}
```



## Les patrons de classe

4

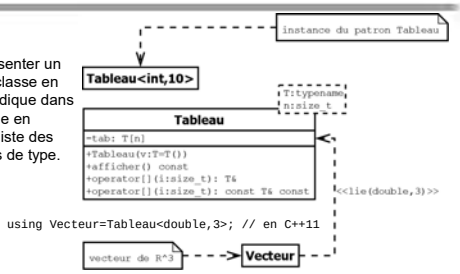
- Lorsqu'une méthode est définie en dehors de la classe, il est nécessaire de rappeler au compilateur l'ensemble des paramètres du patron avec `template`.
- L'opérateur de résolution de portée est lui-même paramétré.
- Les méthodes définies en dehors de la définition du patron ne sont pas inline même si elles sont définies dans le fichier d'entête.



## Patron de classe et UML

5

- Pour représenter un patron de classe en UML, on indique dans un rectangle en pointillé la liste des paramètres de type.




- On peut alors aussi représenter une classe instance de cette classe patron,
  - soit en précisant les paramètres effectifs entre chevrons,
  - soit en précisant qu'une classe donnée est une instance du patron de classe en utilisant le stéréotype `<<lie()>>` (ou `<<bind()>>` en anglais) pour préciser comment sont instanciés les paramètres.



## Patrons de classes et de fonctions

6

- Il est possible qu'un argument formel d'un patron de fonction utilise un patron de classe :

```
template<class T, size_t n>
void fct(Tableau<T,n>& t){ /*...*/ }
```

## Patrons de classes et membres statiques

7

- Un patron de classes peut comporter des membres statiques.

```
template<class T> class A {
      static T x;
      static int nb;
};

// définition des attributs statics
template<class T> T A<T>::x=T();
template<class T> int A<T>::nb=0;
```

- Dans ce cas, chaque instance de la classe dispose de son propre jeu de membres statiques.



## Patron de méthodes

8

- Le mécanisme de patrons de fonctions peut s'appliquer à une méthode de classe ordinaire.

```
class A{
  /*...*/
  template<class T> void fct(T a);
};
```

- Il peut aussi s'appliquer à une méthode d'un patron de classe en définissant de nouveaux paramètres :

```
template<class T>
class B {
  /*...*/
  template<class U> void fct(T a, U b);
};
```



## Identité de classes patrons

9

- Deux classes patrons correspondent au même type si leurs paramètres de type correspondent exactement au même type et si leurs paramètres expressions ont la même valeur.



## Héritage entre classes avec des patrons

10

- On peut définir une classe (patron ou non) qui hérite d'une classe patron :

```
template<class T> class A {
public :
  void f() { /*...*/ }
};

template<class T> class B : public A<T> {/*...*/};

template<class T> class C :
  public A<int>, public A<double>, public A<T> {/*...*/};

template<class X, class Y> class D : public B<X> {/*...*/};

// classe effective
class E : public A<int> {/*...*/};
```

- Dans tous les cas, on doit préciser avec quels paramètres est instancié le patron de classe.



## Appel d'une méthode d'une classe de base

11

- Attention, le compilateur refuse une utilisation directe d'une méthode de la classe de base dans une méthode de la classe héritée :

```
template<class T> class C :
   public A<int>, public A<double>, public A<T> {/*...*/
   void g() {
         f(); // refusé par le compilateur
   }
};

template<class X, class Y> class D : public B<X> { /*...*/
public:
   void g() {
         f(); // refusé par le compilateur
   }
};
```

- Le compilateur ne peut pas toujours savoir quelle est la bonne méthode de la classe de base à appeler.
- Une version spéciale du patron de la classe de base pourrait ne pas proposer la méthode appelée.



## Appel d'une méthode d'une classe de base

12

- 3 moyens pour indiquer au compilateur explicitement que l'on souhaite cet appel :
  - Utilisation de `this` (conseillé pour bénéficier du polymorphisme) :
    `void g() { this->f(); }`
  - Utilisation d'une déclaration using dans la classe :

```
template<class T> class C :
  public A<int>, public A<double>, public A<T>{
/*...*/
public:
  using A<T>::f;
};
```

  - Utilisation de l'opérateur de résolution de portée :
    `void g() { A<int>::f(); A<T>::f(); }`

1

# Le mot clé typename



## typename

2

- Le mot clé `typename` peut remplacer le mot `class` dans la liste des paramètres d'un patron. Il est aussi utilisé pour signaler au compilateur qu'un **identificateur dépendant d'un paramètre de type d'un patron** est un type :

```
template<class T> class A {
     void f() {
         T::It* x; // It: attribut static ou type défini dans T ?
     }
};
```

- En effet, un paramètre générique réel peut être une classe définie par l'utilisateur, à l'intérieur de laquelle des types sont définis.
- Par défaut, un compilateur considère qu'un identificateur dépendant d'un paramètre de type n'est pas un type.
- Il est alors nécessaire d'utiliser le mot clé `typename` pour les introduire.



## Exemple

3

```
struct A {
  typedef int Y; // type défini dans A
};

template <class T> class X {
   /* La classe template X suppose que le
  type générique T définisse un type Y */
  typename T::Y i;
};

X<A> x; /* A peut servir à instancier une
  classe à partir de la classe template X */
```



## Les compilateurs et typename

4

- Alors que dans l'exemple précédent, il devrait être obligatoire d'utiliser `typename` (cf. norme), beaucoup de compilateurs sont plus souples par rapport à cet aspect.
- Cependant il est préférable d'utiliser `typename` car :
  - On évite de perdre du temps à chercher une erreur cryptique quand son utilisation devient complètement requise.
  - Le code sera plus portable d'un compilateur à l'autre.
  - Le code sera plus conforme aux intentions du programmeur et des erreurs de conception pourront être plus facilement détectables par le compilateur.



## typename v.s. class

5

- Les 2 déclarations suivantes sont équivalentes :

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

- Argument en faveur de l'utilisation du `typename` : `class` semble indiquer que T doit être une classe alors que les types primitifs peuvent aussi être utilisés…



## typename v.s. class

6

- Argument en faveur de l'utilisation de `class` :

```
template<typename T, typename T::value_type>
class A;
```

Dans cette déclaration de classe patron :

- seule la première utilisation de `typename` permet de déclarer un paramètre de type;
- la deuxième utilisation déclare un type (comme `int`) qui dépend de T : c'est donc un paramètre expression.

Il peut être alors plus lisible d'utiliser class pour les paramètres de type et réserver l'utilisation du `typename` pour les situations où cela est nécessaire :

```
template<class T, typename T::value_type n>
class A { ... };
```

# Patrons de conception

1

# Les patrons de conception (design patterns)



## Architecture et réutilisabilité

2

- Construire l'architecture d'un logiciel orienté objet est difficile.
- Construire cette architecture de manière à ce que ses éléments soient les plus réutilisables possible est encore plus dur :
  - il faut trouver les objets pertinents à factoriser afin de les représenter sous forme de classes de bonne granularité (pas trop spécifiques mais répondant au problème…) ;
  - construire leur interface;
  - établir la hiérarchie et les associations entre ces classes.



## Architecture et réutilisabilité

3

- L'architecture construite doit être spécifique au problème à résoudre mais aussi suffisamment générale pour faciliter la résolution de futures problèmes.
- L'expérience montre qu'obtenir une architecture flexible et réutilisable est très difficile, voir impossible, à obtenir du premier coup.
- La conception orientée objet est avant tout une question d'expertise : les développeurs expérimentés réutilisent souvent les bonnes solutions qu'ils ont développés.
- C'est pourquoi on retrouve des architectures récurrentes dans beaucoup de systèmes.



## Design patterns

4

- 1995 : Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four: GoF).
  *Design patterns : elements of reusable object-oriented software*, Addison-Wesley
- Transposition de la pratique des design patterns architecturaux (bâtiment) dans l'univers du logiciel.
- Le but est de capitaliser l'expérience dans le domaine de la conception (architecturale) de logiciels orientés objet dans des « patrons de conception ».



## Qu'est ce qu'un design pattern ?

5

- Un design pattern est une solution de conception commune à un problème récurrent dans un contexte donné.
- L'idée est la réutilisation d'une solution éprouvée à une problématique souvent rencontrée.
- Un design pattern propose une solution sous la forme d'un ensemble de classes.
- Un design pattern ne propose pas de code contenant la solution mais un plan de résolution exprimé dans un langage graphique de modélisation (UML).
- Permet de proposer les briques structurelles d'une solution élégante.



## Design patterns

6

- Ce sont donc des petites solutions d'architecture objet, adaptées à des problèmes de modélisation.
- Exemples :
  - Comment construire un singleton, c'est à dire une classe qui garantit l'unicité de son instance ?
  - Comment ajouter dynamiquement des responsabilités à des objets ?
  - Comment programmer la fonction annuler (undo) ?

## Design patterns

7

- Ne pas confondre design pattern et framework. Un design pattern ne décrit pas un ensemble architectural, il permet seulement de résoudre des problèmes de modélisation bien circonscrits et généralement limités à quelques classes.
- Ces recettes techniquement éprouvées ont toujours été imaginées afin d'offrir une réutilisabilité et une évolutivité optimales.
- Souvent, lors d'une utilisation, un design pattern doit être adapté au contexte.



## Avantages

8

Les design patterns permettent :

- de réutiliser plus facilement et plus rapidement les bonnes solutions d'architecture et de conception;
- de rendre ces solutions plus accessibles aux développeurs de nouveaux systèmes.
- de choisir parmi les différentes alternatives les solutions de conception qui rendent les systèmes réutilisables tout en éliminant celles qui compromettent cette réutilisabilité.
- d'améliorer la documentation et la maintenance d'un système en fournissant une documentation explicite des classes et de leurs interactions.



## GoF

9

- Le livre propose 23 solutions à des problèmes récurrents dans les applications informatiques.
- Solutions réparties en 3 catégories :
  - Créationnelles : centrées sur la problématique de construction d'objets.
  - Structurelles : ciblées sur les schémas associatifs entre les classes.
  - Comportementales : caractérisent les moyens utilisés par les classes (et les objets) pour interagir entre elles et se distribuer les responsabilités.




## Présentation d'un design pattern

10

- Chaque design pattern décrit un problème (récurrent dans la conception) et décrit le noyau de la solution qu'il faut adapter au problème.
- En général, un design pattern est décrit avec 4 éléments essentiels :
  - son nom;
  - la description du problème;
  - la solution du problème;
  - les conséquences de la solution.



## Le nom

11

- Exprime en un mot ou deux mots le problème, sa solution et ses conséquences.
- Important car il permet d'augmenter notre vocabulaire (commun) dans le domaine de la conception orienté objet.
- Permet de concevoir à un haut niveau d'abstraction (en pensant à des patterns dans leur ensemble plutôt qu'à leurs spécificités)
- Permet une communication facilité entre les différents concepteurs d'un système ou au travers d'une documentation de ce système.



## Le nom

12

- Abstract factory (Fabrique abstraite)
- Adapter (Adapteur)
- Bridge (Pont)
- Builder (Constructeur)
- Chain of responsability (Chaîne de responsabilité)
- Command (Commande)
- Composite (Composite)
- Decorator (Décorateur)
- Facade (Façade)
- Factory method (Méthode de fabrication)
- Flyweight (Poids mouche)

## Le nom

13

- Interpreter (Interpréteur)
- Iterator (Itérateur)
- Mediator (Médiateur)
- Memento
- Observer (Observateur)
- Prototype (Prototype)
- Proxy
- Singleton (Singleton)
- State (Etat)
- Strategy (Stratégie)
- Template method (Patron de méthode)
- Visitor (Visiteur)



## Le problème

14

- Décrit la situation dans laquelle on peut appliquer le design pattern.
- Il explique le problème et son contexte :
  - Il peut expliquer des problème de conception comme par ex. comment représenter des algorithmes comme des objets.
  - Il peut décrire les structures des objets ou des classes symptomatiques.
- Il peut inclure une liste de conditions qui doivent être rencontrées pour que l'application du pattern soit pertinente.



## La solution

15

- Décrit les éléments qui entrent en jeu dans la solution de conception, leurs relations, leurs responsabilités et leurs collaborations.
- Ne décrit pas un design ou une implémentation spécifique.
- C'est une description abstraite de la conception et de comment l'arrangement général des éléments permet de résoudre le problème.



## Les conséquences

16

- Sont les résultats et les compromis de l'application d'un design pattern.
- Elles sont importantes :
  - pour l'évaluation des différentes alternatives possibles pour résoudre un problème (et donc le choix);
  - pour l'évaluation des coûts induits par la réorganisation due à l'application du design pattern;
  - pour comprendre les bénéfices obtenus par l'application du design pattern;
  - pour confronter les coûts induits et les bénéfices…

1

# Design Pattern Template Method (le patron de méthode)



## Le patron de méthode : factoriser des algorithmes incomplets

2

- C'est un des design patterns les plus utilisés.
- Design pattern comportemental.
- Son but est de factoriser des parties d'algorithmes (c'est-à-dire des parties du comportement) :
  - la trame de base générale d'un algorithme est définie au niveau de la classe de base.
  - la trame est complétée dans les classes dérivées pour les parties qui sont dépendantes de ces classes dérivées.



## Le patron de méthode

3

- Le patron de méthode est donc une méthode (pas forcément virtuelle) d'une classe de base abstraite :
  - implémentant un algorithme commun à toutes les sous-classes de la classe de base;
  - qui délègue son travail à des méthodes virtuelles pures;
  - le reste de l'algorithme dépendant des sous-classes étant implémenté à leur niveau (implémentation des méthodes virtuelles pures).
- Cela évite en particulier d'avoir à redéfinir entièrement un comportement au niveau des classes dérivées.



## Applicabilité

4

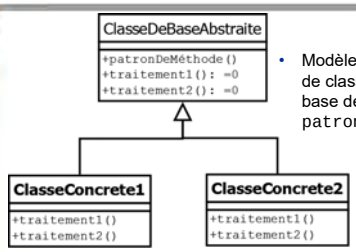
le patron de méthode est utilisé :

- Pour implémenter la partie invariante d'un algorithme une seule fois en laissant l'implémentation des comportements qui diffèrent au niveau de sous-classes.
- Quand un comportement général émergeant dans un ensemble de sous-classes doit être localisé dans une classe commune pour éviter la duplication de code.
- Pour contrôler les possibilités d'extension : on peut définir un patron de méthode qui appelle des opérations « hook » à des points spécifiques permettant des extensions à ces points (voir après).



## Formalisation du modèle

5




- Modèle basé sur une hiérarchie de classes dont la classe de base définit une méthode `patronDeMéthode()`.
- La classe de base déclare des méthodes de `traitement` (généralement virtuelles pures) utilisées par le patron de méthode. Elles sont définies dans les classes concrètes dérivées.
- `patronDeMéthode()` est alors polymorphe sans être pour autant forcément virtuelle … en se reposant sur le polymorphisme des fonctions de `traitement`.



## Modèle général d'implémentation

6

```
class ClasseDeBaseAbstraite {
public:
  void patronDeMethode() {
      ...
      traitement1(); traitement2();
  };
  virtual void traitement1()=0;
  virtual void traitement2()=0;
};

class ClasseConcrete1: public ClasseDeBaseAbstraite {
public:
  void traitement1() { ... /* définition */ ... }
  void traitement2() { ... /* définition */ ... }
};
```

## Exemple

7

```
class Distributeur {
   size_t nb=0;
public:
   Distributeur()=default;
   virtual void put(int x)=0;
   virtual void remove()=0;
   virtual int& item()=0;
   bool empty() const { return nb==0; }
   virtual ~Distributeur()=default;
   // un template method
   virtual void clear() { while(!empty()) remove(); }
};

class Pile : public Distributeur { /*…*/ };
class File : public Distributeur { /*…*/ };
```



## Conséquences

8

- C'est une technique fondamentale en matière de réutilisabilité de code.
- Le code est plus évolutif et facile à maintenir : si l'on souhaite modifier l'algorithme, seul le patron de méthode sera affecté.
- Il y a inversion de contrôle : ce ne sont plus les sous-classes qui appellent des méthodes de la classe de base, mais le contraire (connu sous le nom du Principe d'Hollywood : « Ne nous appelez pas, on vous rappellera… »).



## Méthodes utilisées par les patrons de méthode

9

Les patrons de méthodes peuvent appeler différents types d'opérations :

- Des méthodes concrètes (de la classe de base abstraite, des classes dérivées ou de classes clientes);
- Des méthodes virtuelles pures, i.e. des traitements qui ont été abstraits et délégués aux sous-classes;
- Des opérations primitives (les traitements abstraits);
- Des méthodes de fabrication (voir le design pattern « méthode de fabrication »);
- Des opérations « hook » qui fournissent des comportements par défaut que les sous-classes peuvent redéfinir si nécessaire. Le comportement par défaut peut très bien ne rien faire.



## Opérations « hook »

10

Une sous-classe peut étendre le comportement d'une opération de la classe de base en redéfinissant l'opération et en rappelant l'opération de la classe parente explicitement :

```
class ClasseDeBase {
    virtual void operation() {
     /* comportement de base (indispensable)...*/
    }
};
class ClasseDerivee : public ClasseDeBase{
  void operation() override {
          ClasseDeBase::operation();
          // extension du comportement
          //...
  }
};
```

- Malheureusement, il est facile d'oublier de rappeler l'opération héritée…



## Opérations « hook »

11

- Une solution consiste alors à transformer l'opération en utilisant le design pattern « *template method* » pour donner à la classe de base le contrôle sur comment son comportement peut être étendu :

```
class ClasseDeBase {
   void operation() {
     /* comportement de base (indispensable)*/
     HookOperation(); // point d'extension possible...
   }
   virtual void HookOperation()  {
      // éventuel comportement par défaut...
   }
};
class ClasseDerivee : public ClasseDeBase{
   void ClasseDerivee::HookOperation() override {
     // extension contrôlée du comportement...
   }
};
```



## Opérations « hook »

12

- Dans une telle implémentation, l'utilisateur de la classe abstraite doit alors bien distinguer parmi les opérations :
  - les opérations hook : celles qui peuvent si besoin être redéfinies dans les classes dérivées pour étendre un comportement.
  - les opérations abstraites : celles qui doivent être obligatoirement être définies dans les classes dérivées.

1

# Design Pattern Strategy (Policy) (stratégie)



## Le patron de méthode : factoriser des algorithmes incomplets

2

- Design pattern qui est très souple.
- Design pattern comportemental.
- Il fait intervenir des objets qui représentent des algorithmes que l'on appelle des stratégies.
- Ces stratégies facilitent le choix dynamique des traitements à exécuter.
- Ce design pattern est mis à profit dans les algorithmes de la STL :
  - Objets fonctions dans les algorithmes standards.
  - Comparateurs dans les conteneurs associatifs.
  - Objets `allocator`.



## Applicabilité

3

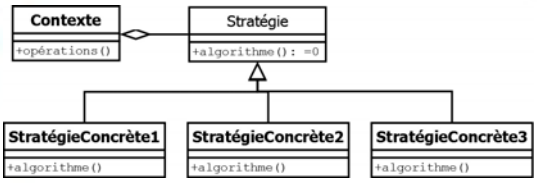
le modèle stratégie est utilisé :

- Quand il est nécessaire de pouvoir manipuler plusieurs algorithmes différents qui mettent en œuvre une même fonctionnalité (par ex. avec différents comportements en temps et en espace mémoire).
- Quand plusieurs classes diffèrent seulement d'un point de vue comportemental : les stratégies proposent un moyen de configurer une classe avec un comportement.
- Quand une classe utilise les données d'un client qu'elle ne doit pas connaître : les stratégies permettent de ne pas exposer des structures de données complexes spécifiques à des algorithmes.



## Formalisation du modèle

4






## Participants

5

Le modèle est constitué d'1 ou plusieurs classes `Contexte` qui doivent accéder à une fonctionnalité (méthode de la classe qui utilise une stratégie abstraite).

- Les mises en œuvre de la stratégie sont réalisées dans des classes concrètes.
- `Contexte` peut, à l'exécution, en fonction de certains critères, choisir différents algorithmes qui réalisent la même fonctionnalité.
- `Strategie` est une classe abstraite qui décrit l'interface de ses filles `StrategieConcretei`.
- Chaque classe concrète met en œuvre à sa façon la méthode `algorithme`.
- Pour changer d'algorithme, il suffit à la classe `Contexte` de changer son objet `Strategie`.



## Modèle général d'implémentation

6

```
class Strategie{
   // ... traitements communs
public:
   virtual void algorithme()=0;
};

class StrategieConcrete1 : public Strategie {
public :
   void algorithme() { /* implémentation */ }
};

class Contexte {
   Strategie* ma_strategie;
public:
   void execute() { ma_strategie->algorithme();}
   void setStrategie(Strategie* s) { ma_strategie=s; }
};
```

## Conséquences

7

- Le choix d'un algorithme se fait dynamiquement, en fonction du contexte, et d'une façon très simple : il suffit de créer une instance de la classe qui encapsule l'algorithme désiré.
- Le choix de l'algorithme n'est pas irréversible : il peut être changé en cours d'exécution en changeant d'objet `Strategie` associé au contexte.
- Les algorithmes sont mis en œuvre dans des classes à part, ce qui contribue à clarifier le code : déchargée de ses fonctions parfois lourdes et complexes, la classe `Contexte` est plus lisible.
- Intégrer un nouvel algorithme est très simple : il suffit de définir une nouvelle classe concrète qui dérive de `Strategie`.



## Conséquences

8

- Cependant, ce modèle augmente le nombre de classes dans le projet ce qui demande une certaine rigueur dans l'organisation.
- L'exécution d'un algorithme donné nécessite la création de l'objet `Strategie` ce qui peut affaiblir légèrement les performances s'il faut changer souvent de stratégie.



## Implémentation avec les template

9

```
template<class Strategie>
class Contexte {
  Strategie ma_strategie;
public:
  void execute() { ma_strategie.algorithme();}
};
...
Contexte<StrategieConcrete1> unContexte;
```

- Une autre des nombreuses implémentations possibles du modèle stratégie consiste à définir la classe `Contexte` sous la forme d'un template paramétré avec une classe `Strategie`.
- L'inconvénient de cette méthode est de figer l'algorithme utilisé par un contexte : il n'est plus possible de le modifier dynamiquement.
- Toutefois, il simplifie la programmation.



## Implémentation avec les fonctions de rappel

10

- Un autre type d'implémentation très courante, notamment dans la STL est de transmettre des fonctions de rappel comme paramètre d'une fonction dont le comportement est à customiser.
- Ces fonctions de rappel sont alors des stratégies du comportement final.

1

# Design Pattern Adapter (Wrapper) (adaptateur)



## Adapter

2

- Design pattern qui permet d'adapter l'interface d'une classe existante à une autre classe existante.
- Design pattern qui permet à des classes de travailler ensemble alors qu'à l'origine leurs interfaces sont incompatibles.
- Design pattern structurel.



## Applicabilité

3

le modèle « adapter » est utilisé :

- Quand on veut utiliser une classe existante et que son interface ne correspond pas à nos besoins.
- Quand on veut créer une classe réutilisable qui coopère avec des classes sans rapport entres elles, *i.e.* qui n'ont pas forcément des interfaces compatibles.



## Exemple

4

- Par exemple, supposons que l'on souhaite implémenter la classe concrète `Pile` issue de la hiérarchie `Distributeur` alors que l'on dispose déjà de la classe `Liste` suivante qui permet de stocker des valeur de type `int` dans une liste doublement chaînée :

```
class Liste {
      //...
public:
      Liste(); // construite une liste vide
      void ajout_en_queue(int x);
      void ajout_en_tete(int x);
      int& tete();
      int& queue();
      void retirer_tete();
      void retirer_queue();
};
```



## Participants

5

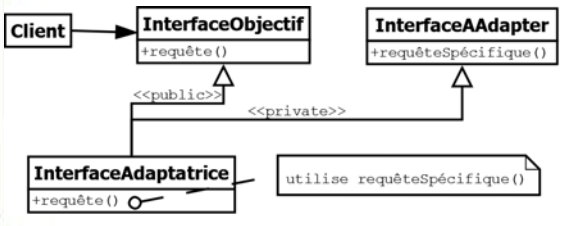
- La classe `InterfaceObjectif` définit l'interface spécifique dont le client a besoin.
- Le `Client` collabore avec des objets en se conformant à l'interface de `InterfaceObjectif`.
- La classe `InterfaceAAdapter` définit une interface existante qui a besoin d'être adaptée.
- La classe `InterfaceAdaptatrice` adapte l'interface de `InterfaceAAdapter` pour se conformer à `InterfaceObjectif`.



## Formalisation du modèle I

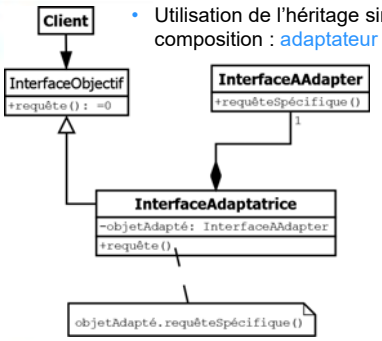
6

- Utilisation de l'héritage multiple :
  adaptateur de classe



## Formalisation du modèle II

7




- Utilisation de l'héritage simple et de la composition : adaptateur d'objet



## Modèle général d'implémentation

8

```
class InterfaceObjectif {
public:
  virtual void requete(); //éventuellement pure
};

class Client {
  InterfaceObjectif* x;
public:
  void execute(){ x->requete(); }
};

class InterfaceAAdapter {
public:
  void requeteSpecifique();
};
```



## Modèle général d'implémentation

9

Héritage multiple (adaptateur de classe) :

```
class InterfaceAdaptatrice :
 public InterfaceObjectif,private InterfaceAAdapter {
public:
  void requete() {
      InterfaceAAdapter::requeteSpecifique();
  }
};
```

Héritage simple et composition (adaptateur d'objet) :

```
class InterfaceAdaptatrice :
           public InterfaceObjectif {
  InterfaceAAdapter* adaptee;
public:
  void requete() { adaptee->requeteSpecifique(); }
};
```



## Conséquences : adaptateur de classe

10

- Un adaptateur de classe ne peut pas être utilisé si on veut adapter une `InterfaceAAdapter` et toutes ses sous-classes.
- Permet de redéfinir certains comportements de `InterfaceAAdapter` puisque `InterfaceAdaptatrice` en hérite.
- On peut avoir accès aux membres `protected` de la classe `InterfaceAAdapter` dans les méthode de `InterfaceAdaptatrice`.



## Conséquences : adaptateur d'objet

11

- Permet à une `InterfaceAdaptatrice` de pouvoir adapter `InterfaceAAdapter` ou n'importe laquelle de ses sous-classes en utilisant un adressage indirect (pointeur ou référence).
- Il est difficile de redéfinir les comportements de `InterfaceAAdapter` dans `InterfaceAdaptatrice`. La classe `InterfaceAAdapter` doit d'abord être sous-classée en redéfinissant ces comportements. Cette sous-classe est alors utilisée à la place de `InterfaceAAdapter`.
- On ne peut pas avoir accès aux membres `protected` de la classe `InterfaceAAdapter` dans les méthode de `InterfaceAdaptatrice`.

1

# Design Pattern Composite (Composé)



## Composite

2

- Design pattern qui permet de manipuler de la même façon un élément simple et un groupe d'éléments.
- Le groupe et l'élément disposent alors de la même interface d'utilisation.
- Design pattern structurel.

- Utilisé quand on veut qu'un client utilisant une hiérarchie de classes puisse ignorer (au niveau de l'interface) la différence entre une composition d'objets et des objets individuels : le client doit traiter les objets de cette structure de façon uniforme.



## Exemple

3

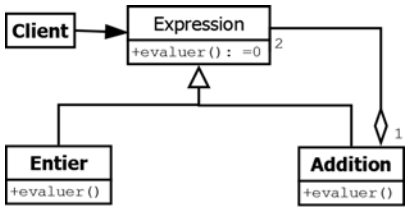
- On veut traiter des expressions mathématiques.
- Une expression est soit une constante entière soit un ensemble d'expressions liées par une opération (par exemple une addition).
- Pour évaluer des expressions, on doit donc pouvoir évaluer de la même manière une constante ou une opération entre expressions (qui peuvent être des constantes ou des opérations).



## Exemple de composite

4

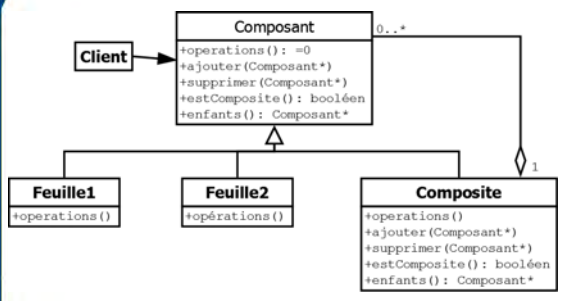
- La clé du composite est une classe abstraite qui représente à la fois les éléments primitifs et les éléments composés :






## Formalisation du modèle

5






## Participants

6

- La classe `Composant` :
  - Déclare l'interface pour la composition d'objet.
  - Met en œuvre le comportement par défaut.
  - Déclare une interface pour l'accès aux composants enfants (itérateurs, etc).

- La classe `feuille` :
  - Représente des objets feuille dans la composition (un objet feuille n'a pas d'enfant).
  - Définie le comportement des éléments primitifs dans la composition.

## Participants

7

- La classe **`Composite`** :
  - Définit un comportement pour les composants ayant des enfants
  - Stocke les composants enfants
  - Met en œuvre la gestion des enfants de l'interface `Composant`.
- La classe **`Client`** :
  - Manipule les objets de la composition à travers l'interface `Composite`.



## Exemple

8

```
class Expression {
public:
  virtual int evaluer() const =0;
  virtual ~Expression(){}
};
class Entier : public Expression {
  int e;
public:
  Entier(int x):e(x){}
  int evaluer() const;
};

// dans l'unité de compilation
int Entier ::evaluer() const { return e; }
```



## Exemple

9

```
class Addition : public Expression {
   Expression* ex1;
   Expression* ex2;
public:
   Addition(Expression* x1, Expression* x2):ex1(x1),ex2(x2){}
   int evaluer() const;
};

class Soustraction : public Expression {
   Expression* ex1;
   Expression* ex2;
public:
   Soustraction(Expression* x1, Expression* x2):ex1(x1),ex2(x2){}
   int evaluer() const;
};

int Addition::evaluer() const
       { return ex1->evaluer()+ex2->evaluer(); }
int Soustraction::evaluer() const
       { return ex1->evaluer()-ex2->evaluer(); }
```



## Exemple

10

```
int main(){
  Expression* e3= new Entier(3);
  Expression* e9= new Entier(9);
  Expression* a=new Addition(e3,e9);
  Expression* e7= new Entier(7);
  Expression* s=new Soustraction(a,e7);
  std::cout<<s->evaluer()<<"\n";
  delete e3; delete e9; delete a;
  delete e7; delete s;
}
```



## Avantages

11

- Le modèle Composite fournit à son utilisateur une interface unique pour un objet ou une collection d'objets.
- L'utilisateur n'a pas à se soucier de savoir si les objets manipulés sont simples ou composés :
  - il les manipule de la même façon;
  - il peut les stocker dans une même structure;
  - il peut profiter de la souplesse que fournit le polymorphisme pour la manipulation d'objets appartenant à une même hiérarchie de classes.



## Avantages / Inconvénients

12

- Un objet composé peut lui-même contenir d'autres objets composés. Les multiples niveaux d'imbrication sont néanmoins simples à manipuler.
- Il est très simple de définir d'autres composants : il suffit de définir une nouvelle classe dérivant de `Composant`.
- Il n'est pas évident d'interdire certains type d'objets `Composant` dans un `Composite` particulier.
- Cela suppose la plupart du temps la création d'une nouvelle classe dérivée de la classe `Composite` qui gère cette interdiction à l'exécution.

1

# Design Pattern Factory Method (Virtual Constructor) (Fabrique)



## Factory Method

2

- Design pattern qui définie une interface de classe pour créer un objet, mais en laissant les sous-classes décider quelle classe de création instancier.
- Ce design pattern délègue l'instanciation aux sous-classes.
- Permet de créer l'objet le mieux adapté à la sous-classe.
- Design pattern créationnel.



## Abstract method

3

- Une **fabrique** est un endroit du code où sont construits des objets.
- Le but de ce patron de conception est d'isoler la création des objets de leur utilisation.
- On peut ainsi ajouter de nouveaux objets dérivés sans modifier le code qui utilise l'objet de base.



## Applicabilité

4

le modèle « Factory Method » est utilisé :

- Quand une classe ne peut pas anticiper a priori les classes d'objets qu'elle doit créer : chacune des sous-classes doit s'occuper de cette tâche elle-même.
- Quand une classe doit spécifier à ses sous-classes les objets qu'elle crée.
- Quand une classe délègue des responsabilités à une ou plusieurs classes assistantes et que l'on veut localiser la connaissance de quelle classe assistante a quelle responsabilité.



## Exemple

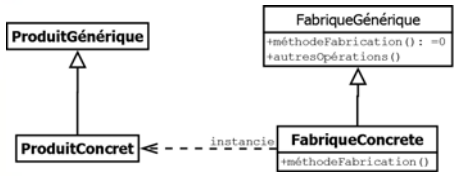
5

- Une application doit créer des documents de différents types issus de différentes sources.
- Cette création est complexe et dépend du type de document qui n'est pas forcément connu a priori : la création est dynamique et les types ne sont pas tous encore connus au moment du développement de l'application.



## Formalisation du modèle

6



## Participants

7

- La classe **`ProduitGénérique`** définie l'interface des objets que la méthode de fabrication doit créer.
- La classe **`ProduitConcret`** implémente l'interface de `ProduitGénérique`.
- La classe **`FabriqueGénérique`** déclare la méthode de fabrication qui retourne un objet de la classe `Produit`.
- La classe **`FabriqueConcrète`** redéfinit la méthode de fabrication qui retourne une instance de la classe `ProduitConcret`.



## Modèle Général d'Implémentation

8

```
class ProduitGenerique {
  /* ...*/
};
class ProduitConcret1 : public ProduitGenerique {
  /* ...*/
};
class FabriqueGenerique {
public:
  virtual ProduitGenerique* methodeFabrication()=0;
};
class FabriqueConcrete1 : public FabriqueGenerique {
public:
  ProduitConcret1* methodeFabrication();
};
```



## Modèle Général d'Implémentation

9

- Le plus souvent, les hiérarchies de classes produit et de fabrique sont presque parallèles :
  à chaque `ProduitConcret_i` correspond une `FabriqueConcrete_i`.
- Les classes `ProduitGenerique` et `ProduitGenerique` peuvent être des classes abstraite ou non (avec des fabrications par défaut, par ex.)



## Conséquences

10

- Plus une classe devient grande et plus elle devient difficile à maintenir. L'intérêt du modèle est alors de pouvoir déléguer une partie du travail d'un objet à d'autres objets qu'il créera lui-même en fonction de ses besoins.
- La délégation d'une partie du travail d'un objet fabrique à un objet produit permet de rendre plus évolutives ces 2 hiérarchies de classes, tout en conservant des liens forts entre elles.
- Ce modèle se base sur le mécanisme du polymorphisme. La méthode de fabrication permet le choix du produit à instancier en fonction de l'objet fabrique qui l'invoque.



## Plusieurs choix d'implémentation

11

- Le modèle peut être implémenté de plusieurs façons.
- Chaque choix d'implémentation correspond à un cas de figure spécifique.
- Il suffit à l'utilisateur de choisir la variante qui s'adapte le mieux à ses besoins :
  - Fabrication par défaut
  - Fabrication paramétrée
  - Fabrication avec template



## Fabrication par défaut

12

- La méthode de fabrication peut très bien ne pas être virtuelle pure et la classe `FabriqueGenerique` ne pas être abstraite.
- La méthode de fabrication peut alors fournir un objet `ProduitGenerique` par défaut qui est concret.
- Si ce produit par défaut convient dans une sous-classe de `FabriqueGenerique`, la méthode de fabrication n'est pas redéfinie.
- Dans ce cas, les hiérarchies de produits et de fabriques peuvent n'être que partiellement parallèles.

## Fabrication par défaut

13

```
class FabriqueGenerique {
public:
  virtual ProduitGenerique* methodeFabrication(){
      return new ProduitGenerique;
  }
};

class FabriqueConcrete1 : public FabriqueGenerique {
public:
  ProduitConcret1* methodeFabrication() (){
      return new ProduitConcret1;
  }
};
```



## Fabrication paramétrée

14

- Si la classe `FabriqueGenerique` ou l'une de ses sous-classes autorise l'utilisation de plusieurs types de produits, alors la méthode de fabrication doit posséder un ou des arguments qui permettent de sélectionner le type adéquat de produit à construire.



## Fabrication paramétrée

15

```
class Fabrique {
public:
  virtual Produit* methodeFabrication(TypeProduit id){
      switch(id){
      /* Produit1, …, Produitx dérivent de Produit */
      case P1 : return new Produit1();
      /*…*/
      case Px : return new Produitx();
      default : throw ProduitInconnu();
      }
  }
};
```



## Fabrication et template

16

- Une autre variante du modèle consiste à implémenter la classe `FabriqueGenerique` en tant que template paramétré par un paramètre `ProduitGenerique`.
- Deux fabriques de types différents peuvent très bien utiliser des produits de même type.
- Contrairement à la fabrication paramétrée, le choix du produit s'effectue à l'instanciation de l'objet `FabriqueGenerique` et ne peut plus être modifié en cours d'exécution.



## Fabrication et template

17

```
class Fabrique {
public:
  virtual Produit* methodeFabrication()=0;
};
template<class unProduit>
class FabriqueGenerique : public Fabrique{
public:
  virtual Produit* methodeFabrication(){
      return new unProduit();
  }
};
void f() {
  FacteurGenerique<Produit1> Fabrique1; /*… */ }
```



## Fabriques et pointeurs intelligents

18

- Dans le cas d'un transfert de propriété de la Fabrique à son client, il est recommandé en C++ d'utiliser des pointeurs intelligents.
- Le type `std::unique_ptr` est un bon candidat pour cela. Etant convertible implicitement en `std::shared_ptr`, cela laisse la liberté au client de savoir comment il va gérer la propriété de la ressource.
- Cela permet d'affranchir le client des éventuelles erreurs d'utilisation en s'assurant de la libération de l'objet si le client n'en prend pas la propriété.

1

# Design Pattern Singleton



## Singleton

2

- Design pattern qui permet de faire en sorte qu'une classe ne puisse instancier qu'un seul objet et d'en fournir un point d'accès global.
- Le modèle singleton permet une instanciation unique d'une classe de manière plus sûre que l'utilisation de variables globales.
- Design pattern créationnel.



## Applicabilité

3

le modèle « Singleton » est utilisé :

- Quand il doit y avoir exactement une instance d'objet d'une classe et que cet objet doit être accessible à plusieurs clients à partir d'un point d'accès connu.
- Quand cette unique instance doit être extensible par héritage et que les clients doivent pouvoir utiliser une instance étendue sans modifier leur code.



## Exemple

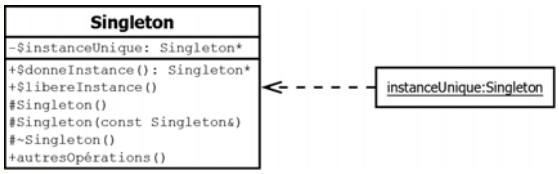
4

- Un manager d'objets.
- Une fabrique d'objets.
- Un moteur de jeu.
- Un pilote de périphérique.
- etc.



## Formalisation du modèle

5






## Participants

6

La classe `Singleton` :

- définit une opération statique qui permet aux clients d'accéder à l'unique instance.
- est responsable de la création et de la destruction de cette instance unique.

## Modèle Général d'Implémentation

7

```
class Singleton {
  private: // ou protected
      static Singleton* instanceUnique;
      Singleton();
      Singleton(const Singleton&);
      virtual ~Singleton();
      void operator=(const Singleton&);
  public:
      static Singleton& donneInstance();
      static void libereInstance();
      /*...*/
};
```



## Modèle Général d'Implémentation

8

```
Singleton* Singleton::instanceUnique=nullptr;

Singleton& Singleton::donneInstance() {
  if (instanceUnique==nullptr)
      instanceUnique= new Singleton;
  return *instanceUnique;
}

void Singleton::libereInstance(){
  delete instanceUnique;
  instanceUnique=nullptr;
}

Singleton::~Singleton(){}
```



## Modèle Général d'Implémentation

9

- Les constructeurs définis par défaut sont déclarés `private (ou protected)`. Ainsi si un utilisateur de la classe essaie d'instancier directement un objet `Singleton`, le compilateur signalera une erreur.
- Les utilisateurs accèdent à l'objet `Singleton` par l'intermédiaire de la méthode statique `donneInstance()`.
- Au premier appel, cette méthode crée l'objet `Singleton` et renvoie son adresse. Aux appels ultérieurs, elle se contente de renvoyer son adresse.
- L'instance ne peut pas être détruite directement par l'utilisateur car le destructeur est `protected`. Ceci est fait par l'intermédiaire de la méthode `libereInstance` qui permet de remettre à `0` le pointeur `uniqueInstance`.



## Modèle Général d'Implémentation

10

- La méthode `donneInstance()` peut aussi être développée avec des paramètres : ces paramètres peuvent être transmis à un constructeur privé lors de la création de l'unique instance.
- Si l'instance est déjà créée, il peut être décidé de remplacer l'instance par une nouvelle avec de nouveaux paramètres
- On peut aussi gérer plusieurs instances en même temps, mais dont une seulement peut être active à un instant donné.



## Singleton et héritage

11

- Il est possible de définir des classes dérivées d'une classe singleton.
- Dans l'implémentation, on peut alors décider:
  - s'il doit y avoir un seul objet par classe dérivée;
  - s'il doit y avoir un seul objet pour toute la hiérarchie de classe.



## Conséquences

12

- Un singleton résout le problème de l'instanciation unique d'une classe et fournit un moyen d'accès contrôlé à cette instance unique.
- Il évite l'utilisation de variables globales pour le stockage d'instances uniques.
- Il peut être étendu au « doubleton », « tripleton », etc., pour autoriser un nombre restreint d'instances.

1

# Design Pattern Iterator



## Le design pattern itérateur (curseur)

2

- Un itérateur est un objet qui permet de parcourir séquentiellement tous les éléments contenus dans un objet agrégateur (le plus souvent un conteneur : liste, arbre, etc).
- Ce parcours se fait sans exposer la structure de données du conteneur.
- L'**itérateur** est un design pattern comportemental.



## Le design pattern itérateur (curseur)

3

Le design pattern est utilisé :

- Pour accéder au contenu d'un objet agrégateur (ici le conteneur) sans exposer sa structure de donnée.
- Pour permettre de multiples « traversées » de l'objet agrégateur.
- Pour fournir une interface uniforme pour traverser un ensemble d'objets agrégateurs qui n'ont pas la même structure de données (itération « polymorphique »).



## Formalisation du modèle

4


- L'objet agrégateur agrège (ou compose) des éléments de type `T` et possède une méthode `first()` qui permet d'obtenir un objet `Iterateur`.
- Un itérateur possède une valeur qui désigne un élément donné d'un conteneur; on dit qu'il pointe sur l'élément.
- Un itérateur dispose de plusieurs opérations :
  - *accéder* (`currentItem()`) à l'élément pointé en cours.
  - *se déplacer* (`next()`) pour pointer vers l'élément suivant.
  - déterminer à tout moment si l'itérateur a épuisé la totalité des éléments du conteneur (`isDone()` qui renvoie vrai si la séquence d'itération est terminée).



## Conséquences

5

- Ce modèle permet de traverser un objet agrégateur de plusieurs manières différentes (par ex. des ordres différents) en proposant différentes implémentations.
- Ce modèle simplifie l'interface de l'agrégateur en rendant explicite le parcours possible des éléments agrégés.
- Plusieurs traversées du même agrégateur peuvent être effectuées. On peut avoir plusieurs traversées en cours avec un objet itérateur par progression.

# Bibliothèques

1

# La bibliothèque standard de C++



## Contenu de la bibliothèque standard C++

2

- Fournit le support des possibilités du langage tel que la gestion de mémoire et les informations de type à l'exécution.
- Fournit des informations concernant les aspects du langage définis par l'implémentation (limites des valeurs possibles pour les types primitifs).
- Fournit des fonctions qui ne peuvent être implémentées de façon optimale dans le langage lui-même pour chaque système (sqrt(), memmove(), …).
- Fournit des utilitaires évolués sur lesquels un programmeur peut s'appuyer afin d'assurer la portabilité (listes, maps, fonctions de tri, entrées/sorties, …).
- Fournit une base commune aux autres bibliothèques.



## Contraintes de conception

3

Les rôles d'une bibliothèque standard impose plusieurs contraintes concernant sa conception. Les possibilités fournies par la bibliothèque doivent :

- Être abordables par tous les programmeurs (même débutants).
- Être suffisamment efficaces pour offrir de véritables alternatives aux fonctions, classes et modèles codés par les programmeurs.
- Être libres de toute règle, ou offrir à l'utilisateur la possibilité de fournir des règles sous forme d'argument.
- Être primitives : un composant remplit une fonction à la fois.
- Être pratiques, efficaces et sûres pour les utilisations courantes.
- Être complètes : si la bibliothèque se charge d'une tâche, elle doit fournir une fonctionnalité suffisante.
- Soutenir les styles de programmation reconnus.
- Être développables : doit traiter les types utilisateurs de façon analogue à celle des types fournis par la bibliothèque.



## Utilitaires généraux

4

- `<utility>` : opérateurs et paires
- `<functional>` : objets fonction
- `<memory>` : allocateurs
- `<ctime>` : date et heure de style C



## Diagnostics

5

- `<exception>` : classe d'exceptions
- `<stdexcept>` : exceptions standards
- `<cassert>` : macro assert
- `<cerrno>` : gestion des erreurs de style C



## Entrées/Sorties

6

- `<iosfwd>` : prédéclarations des fonctions d'E/S
- `<iostream>` : opérations et objets `iostream` standards
- `<ios>` : base pour iostream
- `<streambuf>` : mémoires tampon de flux
- `<istream>` : modèles de flux d'entrée
- `<ostream>` : modèles de flux de sortie
- `<iomanip>` : manipulateurs
- `<sstream>` : flux vers/depuis les chaînes
- `<cctype>` : fonctions de classification de caractères
- `<fstream>` : flux vers/depuis les fichiers

## Support du langage

7

- `<limits>` : plages des valeurs numériques
- `<new>` : gestion de mémoire dynamique
- `<typeinfo>` : identification du type à l'exécution



## Numériques

8

- `<complex>` : opérations et nombres complexes
- `<valarray>` : opérations et vecteurs numériques
- `<numeric>` : opérations numériques généralisées
- `<cmath>` : fonctions mathématiques standard
- `<cstlib>` : nombres aléatoires de style C



## Conteneurs de la STL

9

- `<vector>` : tableau dynamique à une dimension
- `<array>` : tableau statique à une dimension
- `<list>` : liste doublement chaînée
- `<forward_list>` : liste simplement chaînée
- `<deque>` : file à double accès
- `<queue>` : file (FIFO)
- `<priority_queue>` : tas (file d'attente prioritaire)
- `<stack>` : pile (LIFO)
- `<map>` et `<multimap>` : tableaux associatifs (collection triée de paires clé/valeur)
- `<set>` et `<multiset>` : ensembles (collection triée de clés)
- `<unordered_map>` et `<unordered_multimap>` : collection de paires clé/valeur hachée par les clés
- `<unordered_set>` et `<unordered_multiset>` : collection triée de clés hachées par les clés



## la STL

10

- `<vector>` : tableau dynamique de T à une dimension
- `<array>` : tableau statique de T à une dimension
- `<list>` : liste doublement chaînée de T
- `<forward_list>` : liste simplement chaînée de T
- `<deque>` : file à double accès de T
- `<queue>` : file de T
- `<stack>` : pile de T
- `<map>` et `<multimap>` : tableaux associatifs de T
- `<set>` et `<multiset>` : ensembles de T
- `<bitset>` : tableau de booléen
- `<algorithm>` : algorithmes généraux
- `<iterator>` : itérateurs et leur supports (parcours de conteneurs)

1

# La classe string

- exemple de classe disponible dans la STL
- encapsule un ensemble de méthodes qui permettent de gérer une chaîne de caractères (affectation, concaténation, recherche de sous-chaînes, insertion ou suppression de sous-chaîne, etc).
- `#include <string>` // namespace std



## Constructeurs

2

Les constructeurs suivants sont disponibles:

- `string()`
- `string(size_t n, char c);`
- `string(const string& arg);`
- `string(const string& arg, size_t idx, size_t n);`
- `string(const char* arg);`



## Méthodes usuelles

3

- `const char* c_str()` : converts the contents of a string as a C-style, null-terminated.
- `void clear()` : erases all elements of a string.
- `bool empty()` : tests whether the string contains characters or not.
- `size_t length()` : returns the current number of elements in a string.
- `size_t size()` : returns the current number of elements in a string.
- `void resize(size_t s, char c=' ')` : specifies a new size for a string, appending or erasing elements as required.
- `size_t capacity()` : returns the largest number of elements that could be stored in a string without increasing the memory allocation of the string.
- `void reserve(size_t)` : sets the capacity of the string to a number at least as great as a specified number.



## Opérateurs usuels

4

- Affectation : `string& string::operator=(const string&)`
- Concaténation : += et +

- `const char& operator[](size_t pos) const;`
- `Char&& operator[](size_t pos);`

- Les opérateurs ==,!=,<,>,<=,>= sont disponibles : utilisés pour comparer (ordre lexicographique) un objet de type string à un autre objet de type `string`.

- Les opérateurs << et >> sont surchargés pour fonctionner avec des objets de type `ostream` et `istream`.
- Une fonction `getline` permet de saisir une phrase avec des espaces dans un objet de type `string`.



## Méthodes de recherche

5

- `find` : Searches a string in a forward direction for the first occurrence of a substring that matches a specified sequence of characters.
- `find_first_not_of` : searches through a string for the first character that is not any element of a specified string.
- `find_first_of` : searches through a string for the first character that matches any element of a specified string.
- `find_last_not_of` : searches through a string for the last character that is not any element of a specified string.
- `find_last_of` : searches through a string for the last character that is an element of a specified string.
- `rfind` : searches a string in a backward direction for the first occurrence of a substring that matches a specified sequence of characters.



## Méthodes de manipulation

6

- `replace`: replaces elements in a string at a specified position with specified characters or characters copied from other ranges or strings or C-strings.
- `substr`: copies a substring of at most some number of characters from a string beginning from a specified position.
- `swap`: exchange the contents of two strings.

1

# Quelques éléments sur les flux (iostream, fstream)



## Les Flux

2

- Fonctions d'état du flux :
  - `bool good() const` // la derniere opération a reussit
  - `bool eof() const` // fin de fichier ?
  - `bool fail() const` // l'opération suivante échouera
  - `bool bad() const` // le flux est corrompu
- permet d'accéder plus clairement aux constantes: `ios::eofbit, ios::failbit, ios::bad_bit, ios::goodbit==0`
- en général `ios::iostate=int`
- lire les indicateurs d'état d'E/S: `ios::iostate rdstate() const`
- définir les indicateurs d'état d'E/S : `void clear(ios::iostate f=ios::goodbit)`
- ajouter f aux indicateurs d'état d'E/S. `void setstate(iostate f) { clear(rdstate()|f); }`



## Lecture dans un flux istream

3

- `get()` et `getline()` traitent les espaces blancs comme les autres caractères
- `istream& get(char& c);` Lit un caractère dans le flux et le stoke dans `c`.
- `int get();` Lit un caractère dans le flux et le renvoie sous forme d'entier.
- `istream& get(char* c, int n, char term='\n');` S'arrête au caractere de fin ou quand `n-1` caractères sont lus, laisse le caractère de fin dans le flux, insère un `0` à la fin.
- `istream& getline(char* c, int n, char term='\n');` S'arrête au caractère de fin ou quand `n-1` caractères sont lus, supprime le caractère de fin du le flux, insère un `0` à la fin.
- `istream& read(char* c, int n)` lit n caractère au maxi dans `c[0]...c[n-1]`. Ne s'appuie pas sur un caractère de fin. Elle n'insère pas de caractère `0` dans la cible.



## Lecture dans un flux istream

4

- `ignore(int n=1, char c=EOF)` lit les caractères de la même façon que `read()` mais elle ne les stocke pas. S'arrête au caractère de fin `c`.
- `int gcount() const;` renvoie le nombre de caractères lus dans le flux par l'appel de la fonction d'entrée non formatée la plus récente.
- `putback(char c)` fait de c le prochain caractère à lire dans le flux.
- `int peek()` fournit le prochain caractère disponible dans le flux mais sans l'extraire du flux.



## Etat du format

5

- N'utiliser que les noms symboliques des bits de l'état car dépend de l'implémentation.
- `ios::skipws` : passer les espaces blancs
- `ios::left, ios::right (ios::adjustfield)` : calage à droite à gauche
- `ios::boolalpha` : affichage des booléens en `true/false`
- `dec, hex, oct (ios::basefields)` : base d'affichage
- `scientific, fixed (ios::floatfield)` : notation scientifique, point fixe



## Etat du format

6

- `fmtflags flags() const;` lit les indicateurs
- `fmtflags flags(fmtflags f);` définit les indicateurs : renvoie les anciens
- `fmtflags setf(fmtflags f);` ajoute un indicateur
- `fmtflags setf(fmtflags f, fmtfags champ);` ajoute un indicateur sur le champ
- `void unsetf(fmtflags mask);` // efface les indicateurs.
- `myostream.flags(myostream.flags() | ios_base ::show_pos);` équivalent à `myostream.setf(ios_base::show_pos);`
- La technique de l'opération | pour ajouter une option avec `flags` ou `setf` ne s'avère efficace que lorqu'un seul bit contrôle une option.
- Ce n'est pas le cas pour des options telles que la base utilisée pour l'impression des entiers par ex.
- On a donc une deuxième version de `setf` recevant un deuxième pseudo argument qui indiquera quelle sorte d'option

## Les manipulateurs non paramétriques

7

- `dec / hex / oct`
- `boolalpha / noboolalpha`
- `left / base / internal`
- `scientific / fixed`
- `showpoint / noshowpoint`
- `showpos / noshwopos`
- `skipws / noskipws`
- `uppercase / nouppercase`
- `ws`
- `endl`
- `ends`
- `flush`



## Les manipulateurs paramétriques

8

- Les manipulateurs suivants sont plus complexes, car ils s'utilisent avec un paramètre; ils nécessitent l'inclusion de `iomanip`.
- `setbase(int base);` fixe la base de numérotation à utiliser, qui peut être 8, 10 ou 16
- `setprecision(int nbchiffres);` fixe le nombre de chiffres après le point décimal .
- `setw(int largeur);` fixe la largeur minimale du champ pour l'écriture suivante; n'est valable que pour une seule écriture.
- `setfill(char car);` fixe le caractère de remplissage.

1

# Les exceptions standards



## Les exceptions standards

2

- La bibliothèque standard comporte quelques classes fournissant des exceptions spécifiques susceptibles d'être déclenchées par un programme.
- Certaines peuvent être déclenchées par des fonctions de la bibliothèque standard.
- Toutes les classes dérivent d'une classe de base nommée `exception` et sont organisées suivant la hiérarchie ci-après.
- La déclaration de ces exceptions se trouvent dans le fichier d'entête <`stdexcept`>.



## Les exceptions standards

3

```
logic_error
        invalid_argument
        domain_error
        length_error
        out_of_range
        future_error(C++11)

bad_optional_access(C++17)

runtime_error
        range_error
        overflow_error
        underflow_error
        regex_error(C++11)
        system_error(C++11)
            ios_base::failure(C++11)
            filesystem::filesystem_error(C++17)
        tx_exception(TM TS)
        nonexistent_local_time(C++20)
        ambiguous_local_time(C++20)
        format_error(C++20)

bad_typeid
    bad_cast
        bad_any_cast(C++17)
    bad_weak_ptr(C++11)
    bad_function_call(C++11)
    bad_alloc
        bad_array_new_length(C++11)
    bad_exception
    ios_base::failure(until C++11)
    bad_variant_access(C++17)
```



## Les exceptions standards

4

- La classe de base `exception` dispose d'une méthode `what()` qui fournit une chaîne de caractères. Depuis C++11, son prototype est :

```
virtual const char* what() const noexcept;
```

- Cette fonction virtuelle dans `exception` doit être redéfinie dans les classes dérivées.
- Toutes ces classes disposent d'un constructeur recevant un argument de type chaîne dont la valeur pourra être ensuite récupérée en utilisant `what()`.



## Les exceptions standards

5

- On peut créer nos propres classe exception dérivées à partir de ces exceptions standards :
  - On facilite le traitement des exceptions.
  - On est sûr d'intercepter toutes les exceptions avec la gestionnaire `catch(exception& e)`.
  - On peut s'appuyer sur la méthode `what()`.



## Exemple

6

```
// une nouvelle classe FractionException
class FractionException : public std::exception {
   string info;
public:
   FractionException(const char* s) throw():info(s){}
   const char* what() const noexcept { return info.c_str(); }
};

int main(){
   try { fraction fb(3,0); }
   catch(exception& e){ cout<<e.what()<<"\n";}
   return 0;
}
```

1

# Standard Template Library Généralités



## Standard Template Libraries (STL)

2

La **Standard Template Library** aussi appelée **STL** est une bibliothèque C++ de modèles de classes et de fonctions, normalisée par l'ISO. Cette bibliothèque fournit :

- un ensemble de classes conteneurs (vecteurs, tableaux associatifs, listes chaînées, *etc.*), qui peuvent être utilisées pour contenir des éléments de n'importe quel type de données (à condition que certaines opérations comme la copie et l'affectation soient supportées);
- une abstraction des pointeurs (itérateurs) pour parcourir séquentiellement les éléments d'un conteneur;
- des algorithmes génériques ( indépendants des structures de données et utilisant les itérateurs) : algorithmes d'insertion, de suppression, de recherche, de tri….



## L'architecture de la STL

3

- L'architecture de la STL prône une séparation très forte entre la notion de conteneur et celle d'algorithme.
- A l'opposé des concepts habituellement retenus en POO, les algorithmes classiques (tri, échange de données, recopie de séquences) ne sont pas des méthodes de conteneurs.
- Ce sont des fonctions externes ou des objets foncteurs qui interagissent avec les conteneurs via les itérateurs.
- Les concepteurs de la STL ont fait un gros effort d'homogénéisation.
- Beaucoup de méthodes sont communes à différents conteneurs.
- En pratique, dès qu'une action donnée est réalisée avec un conteneur, on peut souvent lui interchanger un autre conteneur.
- Bien que la norme n'impose pas l'implémentation des conteneurs, elle introduit des contraintes d'efficacité qui les conditionnent.



## Catégories de conteneur

4

- Conteneurs séquentiels :
  - Les éléments ont un rang.
  - On peut parcourir le conteneur suivant ce rangement.
  - Quand on insère ou supprime un élément, on le fait en un endroit que l'on a explicitement choisi.
    - `array<T,size_t N>` : tableau statique de `N` cellules de `T` à une dimension
    - `vector<T>` : tableau dynamique de `T` à une dimension
    - `list<T>` , `forward_list<T>` : liste doublement et simplement chaînée de `T`
    - `deque<T>` : file à double accès de `T`
- Conteneurs adaptateurs : il propose une interface différente en adaptant des conteneurs séquentiels.
  - `queue<T>, stack<T>` : file et pile de `T`
  - `priority_queue<T>` : tas de `T`



## Catégories de conteneur

5

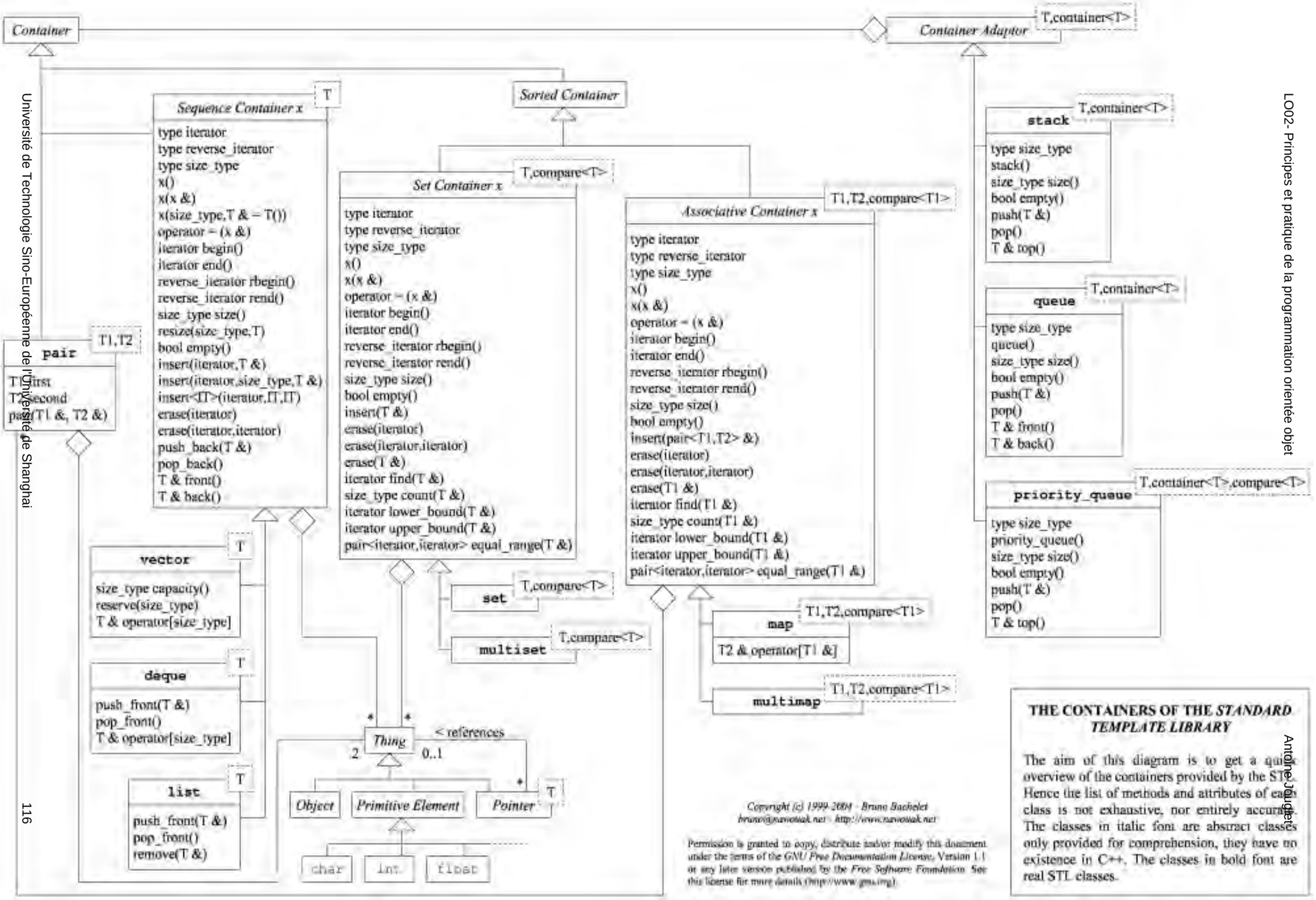
- Conteneurs associatifs :
  - Une valeur est associée à une clé.
  - Une valeur peut être accéder à partir de sa clé associée.
  - Pour insérer un nouvel élément dans ce conteneur, il ne sera théoriquement plus utile/possible de préciser un emplacement.
  - Il est toujours possible de parcourir séquentiellement un tel conteneur.
  - Les opérations sont en O(ln n) sur un conteneur de n éléments.
    - `map<T1, T2>` , `multimap<T1, T2>` : tableau associatif où des données de type `T2` sont associées à des clés de type `T1`;
    - `set<T>, multiset<T>` : ensemble de `T` (clés et éléments confondus)



## Les conteneurs

6

- Conteneurs associatifs non ordonnées:
  - Une fonction de hachage est utilisée pour associer un emplacement à chaque élément.
  - Les éléments peuvent être accédés en 0(1) amorti et O(n) dans le pire cas pour un conteneur de n éléments.
    - `unordered_map<T1, T2>, unordered_multimap<T1, T2>` : tableau associatif où des données de type `T2` sont associées à des clés de type `T1`;
    - `unordered_set<T>, unordered_multiset<T>` : ensemble de `T` (clés et éléments confondus)
- Conteneur d'étendue : une vue (une abstraction) « non propriétaire » sur une séquence continue d'objets, la propriété de ces objets étant détenue par d'autres agrégats.
  - `span<T>` : représente une vue sur une séquence d'éléments de `T` d'une étendue connue sans que ces éléments appartiennent au conteneur `span`.






**THE CONTAINERS OF THE *STANDARD TEMPLATE LIBRARY***

The aim of this diagram is to get a quick overview of the containers provided by the STL. Hence the list of methods and attributes of each class is not exhaustive, nor entirely accurate. The classes in italic font are abstract classes only provided for comprehension, they have no existence in C++. The classes in bold font are real STL classes.

Antoine Jouglet

1

# Standard Template Library
# Les itérateurs



## Les itérateurs

2

- On peut toujours parcourir un conteneur de manière séquentielle (d'un début jusqu'à une fin donnés) en utilisant une implémentation du design pattern « iterator ».



## Les itérateurs en C++

3

- Chaque conteneur **`CONT`** de la STL définit :
  - le type **`CONT::iterator`**.
  - à ce type est associé **une séquence de parcours** (qui dépend du conteneur).
  - une méthode **`CONT::begin()`** qui renvoie un objet de la classe `CONT::iterator` « pointant » sur le premier élément du conteneur dans la séquence.
  - une méthode **`CONT::end()`** qui renvoie un objet de la classe `CONT::iterator` « pointant » sur un élément fictif considéré comme étant situé juste après le dernier élément du conteneur dans la séquence.



## Les itérateurs en C++

4

Une classe **`iterator`** définit au moins les méthodes suivantes :

- **`operator*() const`** qui renvoie une référence sur l'élément pointé;
- **`operator++()`** qui permet de faire en sorte que l'itérateur pointe sur l'élément suivant (déplacement séquentiel).
- **`operator!=(iterator) const`** et **`operator==(iterator) const`** qui permettent de savoir si deux itérateurs pointent ou non sur le même élément.
- Lorsqu'un conteneur c est vide alors (`c.begin()==c.end()`) est vrai.



## Catégories d'itérateurs

5

- Les itérateurs sont classés par catégories définissant leur possibilités :
  - itérateur unidirectionnel : seulement l'opération ++;
  - itérateur bidirectionnel : opérations ++ et `--`;
  - itérateur à accès direct (random access) :
    - opérations `++` et `--`;
    - l'opération `it+i` est possible et `it[i]==*(it+i)`;
    - ces itérateurs peuvent être comparés avec un opérateur d'inégalité (`<`, `<=`, `>`, `>=`).



## Catégories d'itérateurs

6

- Suivant la structure de données qu'ils représentent, les conteneurs de la STL implémentent un type iterator de l'une de ces catégories :
  - itérateur unidirectionnel : `forward_list`
  - itérateur bidirectionnel : `list`, `map`, `multimap`, `set`, `multiset`.
  - itérateur à accès direct (random access) : `vector`, `deque`, `string`.

## Catégories d'itérateurs

7

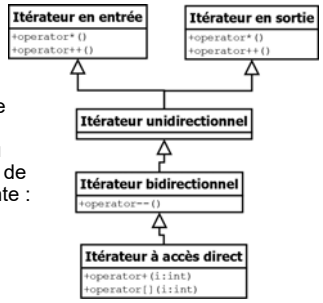
- On distingue aussi :
  - Les itérateurs en entrée (input) : ils autorisent la consultation de la valeur pointée mais pas sa modification :
    ```
    …=*it; // ok
    *it=…; // incorrect
    ```
  - Les itérateurs en sortie (output) : ils autorisent la modification de la valeur pointée mais pas sa consultation :
    ```
    …=*it; // incorrect
    *it=…; // ok
    ```



## Hiérarchie d'itérateurs

8

- Il est courant de représenter les 5 catégories d'itérateur suivant une hiérarchie selon laquelle toute catégorie possède au moins les possibilités de la catégorie précédente :






## Le type `reverse_iterator`

9

- Toutes les classes conteneur pour lesquelles `iterator` est au moins bidirectionnel disposent d'un second itérateur nommé `reverse_iterator`.
- La séquence associée à ce type est la même que celle associée au type `iterator` mais dans le sens inverse.
- `rbegin()` pointe sur le dernier élément du conteneur. `rend()` pointe juste avant le premier.
- Les opérations `++` et `--` sont inversées.



## Le type `const_iterator`

10

- Toutes les classes conteneur disposent aussi d'un type itérateur nommé **`const_iterator`**.
- Ce type d'itérateur fonctionne comme le type `iterator` sauf que la méthode `operator*() const` renvoie une référence constante de l'objet pointé.
- Les méthodes `begin() const` et `end() const` d'un conteneur renvoient un objet de type `const_iterator`.
- C'est pourquoi seuls les itérateurs `const_iterator` sont utilisables à partir d'une référence constante de conteneur ou à partir d'un conteneur constant.
- Le type `const_reverse_iterator` est aussi disponible.



## Intervalle d'itérateurs

11

- Grâce à la notion d'itérateur, il est possible de parcourir un conteneur séquentiellement du début à la fin.
- De façon générale, un intervalle d'itérateurs représenté par deux valeurs d'itérateur **`it1`** et **`it2`** sert à désigner les éléments entre `it1` et `it2` (`*it1` compris, `*it2` non compris).

```
vector<int> v(10,0);
vector<int>::iterator it1=v.begin(), it2=v.end();
/* [it1; it2) est un intervalle désignant l'ensemble
   des éléments de v */
it1=v.begin()+2, it2=v.begin()+6;
/* [it1; it2) désigne maintenant les éléments v[2],
   v[3], v[4], v[5] */
```

1

# Standard Template Library
# Les conteneurs séquentiels



## Les conteneurs séquentiels

2

- `array` : tableau statique de `T` à une dimension
- `vector` : tableau dynamique de `T` à une dimension
- `list, forward_list` : liste doublement ou simplement chaînée de `T`
- `deque` : file à double accès de `T`
- Les éléments sont ordonnés. On peut parcourir le conteneur suivant cet ordre. Quand on insère ou supprime un élément, on le fait en un endroit que l'on a explicitement choisit.
- Il existe des adaptateurs de ces classes (`queue` : file de `T`, `stack` : pile de `T`, `priority_queue` : tas de `T`) ayant une interface restreinte.



## Construction d'un conteneur séquentiel

3

- Construction d'un conteneur vide : appel du constructeur sans argument.
- Construction avec un nombre donné d'éléments : l'appel d'un constructeur avec un seul argument `int n` construit un conteneur initialisé avec n éléments : les éléments objets sont initialisés avec un constructeur sans argument.
- Construction avec un nombre donné d'élément initialisés à une valeur : le premier argument fournit le nombre d'élément, le deuxième fournit la valeur. Les éléments sont alors construits par recopie de la valeur fournie.
- Construction avec recopie d'un autre conteneur (de même type).



## Construction d'un conteneur séquentiel

4

- Toute construction d'un conteneur non vide dont les éléments sont des objets entraîne :
  - soit l'appel d'un constructeur : il peut s'agir d'un constructeur sans argument lorsqu'aucun argument n'est nécessaire;
  - soit l'appel d'un constructeur de recopie.
- Il est possible de construire un conteneur à partir d'une séquence d'éléments de même type désignée par un intervalle d'itérateurs.

```
template<class Iterator>
vector(Iterator first, Iterator last);
```

- Permet notamment de construire un conteneur à partir d'un autre conteneur d'un autre type.



## Affectation entre conteneurs

5

- Affectation : on peut affecter un conteneur d'un type donné à un autre conteneur de même type (même nom de patron, même type d'éléments).
- L'utilisation de l'opérateur affectation n'est possible qu'entre conteneurs de même type.
- Les tailles des deux conteneurs ne sont pas forcément égales.



## Affectation entre conteneurs contenant des éléments de même type

6

- `assign` permet d'affecter, à un conteneur existant, les éléments d'une séquence définie par intervalle `[debut, fin)`, à condition que les éléments désignés soient du même type.

```
c.assign(it1, it2);
```

- Une surcharge de `assign` permet d'affecter à un conteneur, un nombre donné d'éléments avec une valeur donnée (à partir du début).

```
c.assign(20,'a');
```

- Dans les deux cas, les éléments existants seront remplacés par les éléments voulus, comme si il y avait eu affectation.

## Quelques méthodes utiles

7

- La méthode `clear()` vide le conteneur de son contenu.
- La méthode `size()` permet de connaître le nombre d'éléments contenus dans le conteneur.
- La méthode `resize(size_type taille, T c=T())` permet de redimensionner un conteneur avec une taille donnée.
- La méthode `max_size()` permet de connaître le nombre maximum d'éléments que peut contenir un conteneur à un instant donné.
- La méthode `empty()` renvoie `true` si le conteneur ne contient pas d'éléments.
- La méthode `front()` renvoie une référence sur le premier élément (`c.front()` équivaut à `*(c.begin())`).
- La méthode `back()` renvoie une référence sur le dernier élément (`c.back()` équivaut à `*(c.rbegin())`).
- La méthode `swap(CONT&)` permet d'échanger le contenu de deux conteneurs de même type : `c1.swap(c2);`



## Insertion / suppression dans un conteneur séquentiel

8

- `c.insert(position, valeur);` insère dans c une valeur avant l'élément pointé par l'itérateur `position` et fournit un itérateur sur l'élément inséré.
- `c.insert(position, nb, valeur);` permet d'insérer dans c une valeur `nb` fois (ne renvoie rien).
- `c.insert(debut, fin, position);` insère dans c les valeurs de l'intervalle d'itérateurs `[debut,fin)` avant l'élément pointé par l'itérateur `position` (ne renvoie rien).
- `push_back(valeur);` insère dans c une valeur après le dernier élément.
- `emplace_back(…);` construit un nouvel objets (en utilisant les arguments fournis) après le dernier élément.
- `c.erase(position);` supprime de c l'élément désigné par position, fournit un itérateur sur l'élément suivant ou sur la fin de séquence `end()` s'il n'existe pas.
- `erase(début,fin);` supprime de c les éléments de l'intervalle `[début, fin)`, fournit un itérateur sur l'élément suivant ou sur la fin de séquence `end()` s'il n'existe pas.
- `c.pop_back();` supprime le dernier élément de c.



## vector

9

- Un `vector` est semblable à un tableau de type C mais encapsulé dans une classe pour fournir des capacités supplémentaires, en particulier, la croissance dynamique. Si n est la taille du vecteur, ses principales caractéristiques sont :
  - Accès indexé aux éléments en $O(1)$.
  - Ajout ou suppression d'un élément à la fin du vecteur sans redimensionnement en $O(1)$.
  - Ajout ou suppression d'un élément à la fin du vecteur avec redimensionnement en $O(n)$. ($O(1)$ amorti).
  - Ajout ou suppression d'un élément au milieu du vecteur en $O(n)$.



## vector

10

- La méthode `operator[](size_type i)` permet d'accéder à l'élément situé à l'indice `i`.
- `v[i]` est équivalent à `*(v.begin()+i)`.
- La méthode `at(size_type i)` fait comme `[]` mais génère une exception de type `out_of_range` en cas d'indice incorrect, ce que l'opérateur `[]` ne fait théoriquement pas.
- La méthode `capacity()` fournit la taille potentielle du vecteur (le nombre d'éléments qu'il pourra accepter sans avoir à effectuer une nouvelle allocation).
- La méthode `reserve(size_type taille)` permet de modifier la capacité.
- La méthode `resize(size_type taille)` modifie la taille effective du vecteur.
- La méthode `shrink_to_fit()` est une requête (dont la réponse dépend de l'implémentation) pour réduire la capacité à la taille courante.



## Invalidation des itérateurs ou des références

11

- Certaines opérations entraînent l'invalidation des itérateurs ou des références sur certains éléments de ce vecteur :
  - tous les éléments en cas d'augmentation de la capacité du vecteur;
  - tous les éléments à la suite d'un élément inséré;
  - tous les éléments à la suite d'un élément supprimé.



## La spécialisation `vector<bool>`

12

- La classe `vector` est spécialisée pour le type `bool` afin d'optimiser l'encombrement mémoire.
- Dans un tel vecteur, chaque booléen prend un bit dans la mémoire.
- La méthode `flip()` permet d'inverser les bits d'un tel vector.
- Les références renvoyées par les méthodes de tels vecteurs ne sont pas des références classiques mais des objets de la classe `vector<bool>::reference()`.

## list

13

- La classe `list` modélise le concept de liste doublement chaînée.
- Contrairement aux vecteurs, elle ne possèdent pas d'opérateur d'indexation mais permettent d'ajouter ou de supprimer un élément à n'importe quelle position (spécifiée par un itérateur) en $O(1)$.
- Du fait de la structure de chaîne, l'occupation en mémoire d'une liste est le plus souvent supérieure à celle d'un vecteur pour le même nombre d'éléments stockés.
- Les itérateurs sont seulement bidirectionnels.



## List : suppressions conditionnelles

14

- La méthode `remove(valeur)` supprime tous les éléments égaux à `valeur`.
- La méthode `remove_if(predicat)` supprime tous les éléments `x` pour lesquels `predicat(x)` renvoie `true`.
- Les méthodes `remove` ne renvoient aucune valeur.



## List : tris

15

- La classe `list` dispose de son propre tri en $O(n \log n)$.
- Ce tri tire parti de la structure des listes chaînées.
- La méthode `sort()` trie la liste en s'appuyant sur l'opérateur `operator<` des éléments agrégés.
- La méthode `sort(cmp)` trie la liste en s'appuyant sur le prédicat binaire `cmp` : `cmp(c1,c2)` doit renvoyer `true` si `c1` doit être avant `c2` dans la liste.



## List : opérations globales

16

- La méthode `unique()` élimine les doublons sur une liste préalablement triée. Si la liste est non triée, elle remplace par un seul élément les suites de valeurs égales.
- On utilise `unique(predicat)` si on veut utiliser autre chose que `operator==` pour les éléments agrégés.
- `L1.merge(L2)` (ou `merge(L2,predicat)`) permet de fusionner deux listes (en s'appuyant sur < ou sur `predicat`).
- La méthode `splice` déplace les éléments d'une liste dans une autre liste. Les éléments déplacés sont supprimés de la liste d'origine.



## Invalidation des itérateurs ou des références

17

- En cas d'insertion ou de suppression, seuls les itérateurs et les références des éléments insérés ou supprimés deviennent invalides.



## deque

18

- Le conteneur `deque` est comme `vector` au niveau des complexités mais il offre en plus l'insertion et la suppression en début en $O(1)$ (si pas de redimensionnement) : `push_front(x)`, `pop_front()`.
- Il est donc très efficace dès lors qu'il s'agit d'ajouter ou de retirer une valeur à l'une de ses extrémités.
- Il possède un opérateur d'indexation `operator[](size_type)` de complexité $O(\log n)$ amortie
- Les insertions en milieu de séquence sont plus rapides que sur un vecteur sans pour autant atteindre le niveau de performances des listes.

## Adaptateurs

19

- Les utilisations spécialisées des conteneurs élémentaires permettent de modéliser 3 structures de données classiques : les piles, les files et les tas.
- Ce sont des classes qui adaptent un conteneur élémentaire à une structure de donnée en lui fournissant une nouvelle interface : on parle alors d'adaptateur.



## Adaptateur `stak` (pile)

20

- `empty()`, `size()`, `top()`, `push(valeur)`, `pop()`.
- `template <class T, class Container = deque<T> > class stack;`
- La classe `Container` peut être n'importe quelle classe implémentant les méthodes `back()`, `push_back()`, `pop_back()`.



## Adaptateur queue (file)

21

- `empty()`, `size()`, `front()`, `back()`, `push(valeur)`, `pop()`.
- `template <class T, class Container = deque<T> > class queue;`
- La classe `Container` peut être n'importe quelle classe implémentant les méthodes `front()`, `back()`, `push_back()`, `pop_front()`.



## Adaptateur `priority_queue` (tas)

22

- `empty()`, `size()`, `top()`, `push(valeur)`, `pop()`.
- `template<class T, class Container = vector<T>, class Compare=less<typename Container::value_type> > class priority_queue;`
- La classe `Container` peut être n'importe quelle classe implémentant les méthodes `front()`, `push_back()`, `pop_back()` et dont les éléments sont accessibles avec des itérateurs à accès direct.



## Comparaison de conteneurs

23

- Pour chaque classe conteneur C sont définies les fonctions
  `bool operator==(const C<T>& L, const C<T>& R);`
  `bool operator<(const C<T>& L, const C<T>& R);`
  qui permettent la comparaison de 2 conteneurs de même type.
- Si `c1` et `c2` sont deux conteneurs de même type, alors `c1==c2` renvoie `true` si ils ont la même taille (`c1.size()==c2.size()`) et si les éléments de même rang sont égaux.
- Il est nécessaire que l'opérateur `operator==` soit défini pour le type `T` d'élément agrégé.



## Ordre entre conteneurs

24

- Si `c1` et `c2` sont deux conteneurs de même type, alors `c1<c2` renvoie `true` si `c1` et plus petit dans l'ordre lexicographique que `c2` :
  - Les éléments de même rang des conteneurs c1 et c2 sont comparés dans l'ordre séquentiel.
  - La comparaison se termine quand :
    - La fin de l'un des conteneurs est atteinte;
    - L'égalité entre deux éléments de même rang est fausse.
- Il est nécessaire que les opérateur `operator==` et `operator<` soient défini pour le type `T` d'élément agrégé.
- Les opérateurs de comparaison entre conteneurs de même type <=, >=, < sont également disponibles (par génération à partir de == et <).

1

# Standard Template Library
# Les conteneurs associatifs ordonnés ou non-ordonnés



## Les conteneurs associatifs

2

- `map<T1,T2>, unordered_map<T1,T2>` : tableau qui associe des valeurs de type `T2` à des clés de type `T1` (unicité des entrées de type `T1`).
- `multimap<T1,T2>, unordered_multimap<T1,T2>` : idem que `map` mais il peut y avoir plusieurs entrées avec la même clé.
- `set<T>, unordered_set<T>` : ensemble de `T` (unicité des éléments)
- `multiset<T>, unordered_multiset<T>` : ensemble de `T` (il peut y avoir plusieurs fois le même élément).
- Une valeur est associée à une clé.
- Une valeur peut être accéder à partir de sa clé associée.
- Pour insérer un nouvel élément dans ce conteneur, il ne sera théoriquement plus utile de préciser un emplacement.
- Il est toujours possible de parcourir séquentiellement un tel conteneur.



## Conteneurs associatifs

3

- Les conteneurs associatifs ont pour principale vocation de retrouver une information, non plus en fonction de la place dans le conteneur, mais en fonction d'une partie de sa valeur nommée clé.
- L'insertion, la suppression et la recherche d'un élément dans ces conteneurs se font en temps logarithmique ($O(\log n)$).



## Associations simples/multiples

4

- **`map`** et **`unordered_map`** imposent l'unicité des clés alors que **`multimap`** et **`unordered_multimap`** ne l'imposent pas.
- Dans le cas où la valeur associée à la clé n'existe pas, ce qui veut dire que les éléments se limitent à leur seule clé, on utilise les **`set`**, **`multiset`** , **`unordered_set`** , **`unordered_multiset`** : ils représentent des ensembles.



## Conteneurs associatifs ordonnés et relation d'ordre

5

- Dans les conteneurs associatifs ordonnés, une relation d'ordre est donc nécessaire pour ordonner ces éléments.
- Par défaut, cette relation d'ordre est `less<type_clef>` qui utilise `operator<`.
- Il est possible de choisir la relation d'ordre qui sera utilisée pour ordonner intrinsèquement le conteneur (classe fonction fournie en 3ème argument du template).
- La méthode `key_comp()` fournit la fonction utilisée pour ordonner les clés.
- La méthode `value_comp()` donne le résultat de la comparaison de 2 éléments désignés par 2 itérateurs.
- Dans ce type de conteneur ordonné, on dit que 2 éléments sont équivalents s'ils ne sont pas comparables par rapport à la relation d'ordre.
- Ainsi, si cette relation est `operator<`, 2 éléments a et b sont équivalents si `(!a<b && !b<a)` renvoie `true`.



## Conteneurs associatif non-ordonnés

6

- Les conteneurs associatifs non-ordonnés se base sur une fonction de hachage calculé à partir des clés afin de déterminer l'emplacement des éléments.
- Par défaut les conteneurs utilisent une fonction de hachage construite à partir de std::hash.
- L'utilisateur peut fournir ses propres fonctions de hachage.

## map et unordered_map

7

- Un conteneur **map** est formé d'éléments composés de deux parties : une clé et une valeur.
- Pour représenter de tels éléments, on utilise un patron nommé **pair** paramétré par le type de la clé et par celui de la valeur.



## Construction d'un map

8

- Possibilités de construction :
  - construction d'un conteneur vide ;
  - construction à partir d'un conteneur de même type;
  - construction à partir d'un intervalle d'itérateurs;



## pair (header <utility>)

9

```
template<class T1, class T2> struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first; T2 second;
  pair(): first(T1()), second(T2()){}
  pair(const T1& x, const T2& y):first(x),second(y){}
  template<class U, class V> pair(const pair<U,V>& p)
          :first(p.first),second(p.second){}
}
template <class T1, class T2>
pair<T1,T2> make_pair(const T1 &a, const T2 &b){
  return pair<T1,T2>(a,b);
}
```

Pour un élément d'un map, first désigne la clé, second désigne la valeur.



## Insertion et accès

10

- L'operateur `value_type& operator[](const key_type& x );` permet à la fois d'insérer des éléments et d'y accéder.
- Si m est un map, l'instruction **m[k]=v**; permet d'insérer l'élément de clé k et de valeur v dans m.
- Un élément **v** n'est introduit dans m que s'il n'existe pas d'autres élément possédant la clé **k**.
- Une fois cette valeur insérée, l'opération **m[k]** renvoie une référence sur **v**.
- Une tentative d'accès **m[k]** avec une clé k inexistante amène à la création d'un nouvel élément créé avec le constructeur sans argument `value_type()`.



## map et itérateurs

11

- L'indirection d'un itérateur `it` renvoie une référence sur un objet de type `pair<key_type,value_type>`.
- `(*it).first` désigne la valeur de la clé;
- `(*it).second` désigne la valeur associée.



## Chercher un élément

12

- La méthode **find(k)** fournit un itérateur sur un élément ayant de clé k.
- Si aucun élément ayant cette clé n'est trouvée, cette méthode renvoie `end()`.

## Insertion et suppression dans un map

13

- Insertion :
  - `insert(element);`
  - `emplace(element);`
  - `insert(debut,fin);` insère les paires de la séquence `[debut,fin)`
- Suppression / extraction :
  - `erase(position); extract(position);`
  - `erase(debut,fin);`
  - `erase(cle); extract(cle);`
- Les opérations sur les conteneurs associatifs n'entraînent jamais d'invalidation des références et des itérateurs, sauf les éléments supprimés.



## `multimap, unored_multipmap`

14

- Une même clé peut apparaître plusieurs fois.
- Les éléments correspondant apparaissent consécutivement.
- `[]` n'est plus applicable.
- La méthode `erase` peut supprimer plusieurs éléments ayant la même clé.
- Si il existe plusieurs clés équivalentes, `find` fournit un itérateur sur un des éléments ayant la clé voulue. Attention on ne précise pas si il s'agit du premier.



## Chercher dans un `(unordered_)multimap`

15

- Si il existe plusieurs clés équivalentes, `find(k)` fournit un itérateur sur un des éléments comme clé `k`. Attention, ce n'est pas obligatoirement le premier.
- La méthode `count(k)` renvoie le nombre d'éléments ayant comme clé `k`.



## `(unordered_)set` et `(unordered_)multiset`

16

- Les conteneurs `set` sont des cas particuliers des conteneurs `map` dans lequels aucune valeur n'est associée à la clé.
- Les éléments du conteneur ne sont donc plus que des clés (et non plus des objets `pair`).
- Un élément d'un conteneur est une constante : on ne peut pas en modifier la valeur.
- Un conteneur de type `set` est obligatoirement ordonné, tandis qu'un ensemble mathématique ne l'est pas nécessairement.

1

# Standard Template Library
# Les algorithmes



## Les algorithmes standards

2

- Les algorithmes standards sont des patrons de fonctions.
- Leur code est écrit sans connaissance préalable des éléments manipulés.
- L'accès aux éléments se fait toujours indirectement par l'intermédiaire d'itérateurs à partir desquels sont déduits (par indirection) les types des éléments manipulés.



## copy

3

```
template <class InputIt, class OutputIt>
OutputIterator copy(InputIt first, InputIt last,
                    OutputIt place);
```

- Permet de copier les éléments d'un conteneurs désignés par l'intervalle d'itérateurs `[first,last)` à partir de la position désignée par l'itérateur `place`. Les éléments du conteneur destination sont alors modifiés.
- S'il s'agit d'une copie dans un même conteneur, il ne doit pas y avoir intersection entre les éléments copiés et les éléments affectés.
- Retourne un itérateur sur le dernier élément copié dans le conteneur destination.



## generate

4

```
template <class ForwardIt, class Generator>
void generate(ForwardIt first, ForwardI last,
              Generator gen );
```

- Permet d'affecter les éléments désignés par l'intervalle `[first; last)` avec des appels successifs de l'objet fonction `gen`.
- A le même comportement que :

```
template <class ForwardIt, class Generator>
void generate(ForwardIt first, ForwardIt last,
              Generator gen) {
  while (first != last) *first++ = gen();
}
```



## Algorithmes de recherche

5

- Retourne un itérateur sur la 1ère occurrence dans `[first,last)` égal à `value` :

```
template <class InputIt, class T>
InputIt find(InputIt first, InputIt last, const T& value );
```

- Retourne un itérateur sur la première occurrence dans `[first1,last1)` de n'importe quel élément dans `[first2,last2)` :

```
template<class ForwardIt1, class ForwardIt2, class BinaryPred>
ForwardIt1 find_first_of(
   ForwardIt1 first1, ForwardIt1 last1,
   ForwardIt2 first2, ForwardIt2 last2);

template<class ForwardIt1, class ForwardIt2>
ForwardIt1 find_first_of(
   ForwardIt1 first1, ForwardIt1 last1,
   ForwardIt2 first2, ForwardIt2 last2, BinaryPred pred );
```



## Algorithmes de recherche

6

- Retourne un itérateur sur la première occurrence dans `[first1,last1)` de la séquence d'éléments désignée par`[first2,last2)` :

```
template <class ForwardIt1, class ForwardIt2>
ForwardIt1 search(ForwardIt1 first1, ForwardIt1 last1,
                  ForwardIt2 first2, ForwardIt2 last2);

template <class ForwardIt1, class ForwardIt2,
          class BinaryPred>
   ForwardIt1 search(ForwardIt1 first1, ForwardIt1 last1,
                     ForwardIt2 first2, ForwardIt2 last2,
                     BinaryPred comp);
```

## Algorithmes de recherche

7

- Retourne un itérateur sur la première occurrence dans [first1, last1) de la séquence de count fois l'élément value :

```
template <class ForwardIt, class Size, class T>
   ForwardIt search_n(ForwardIt first, ForwardIt last,
                      Size count, const T& value);

Template <class ForwardIt, class Size, class T,
                                       class BinaryPred>
  ForwardIt search_n(ForwardIt first, ForwardIt last,
                     Size count, const T& value,
                     BinaryPred pred);
```



## Algorithmes de recherche de maximum ou minimum

8

```
template <class ForwardIt>
ForwardIt max_element(ForwardIt first, ForwardIt last);

template <class ForwardIt, class Compare>
ForwardIt max_element(ForwardIt first, ForwardIt last,
                                       Compare comp);
template <class ForwardIt>
ForwardIt min_element(ForwardIt first, ForwardIt last);

template <class ForwardIt, class Compare>
ForwardIt min_element(ForwardIt first, ForwardIt last,
                                       Compare comp);
```



## Algorithmes de transformation d'une séquence

9

Algorithmes qui modifient les valeurs d'une séquence ou leur ordre :

- Remplacement de valeurs : `replace`, `replace_if`
- Rotation : `rotate`
- Permutations : `next_permutation`, `prev_permutation`
- Permutations aléatoires : `random_shuffle`
- Partitions : `partition`, `stable_partition`



## Algorithmes de suppression

10

- `remove`
- `remove_copy`
- `remove_copy_if`
- `unique`
- `unique_copy`



## Algorithmes de tris

11

```
template <class RandomAccessIt>
void sort(RandomAccessIt first, RandomAccessIt last);

template <class RandomAccessIt, class Compare>
void sort(RandomAccessIt first, RandomAccessIt last,
          Compare comp);

template <class RandomAccessIt>
void stable_sort(RandomAccessIt first,
                 RandomAccessIt last);

template <class RandomAccessIt, class Compare>
void stable_sort(RandomAccessIt first,
                 RandomAccessIt last,
                     Compare comp );
```



## Et autres…

12

- Algorithmes à caractère numérique : somme des éléments d'une séquence, produit scalaire, …
- Algorithmes à caractère ensembliste : union, intersection , différence, différence symétrique, inclusion, …
- …
- De nombreux algorithmes ont été ajoutés en C++11, C++17 et C++20.

# BIBLIOGRAPHIE


- [Bersini, 2011] Hugues Bersini, *L'orienté objet*, ed. Eyrolles, 4ème ed, 2011.
- [Blanchette et Summerfield, 2007] Jasmin Blanchette et Mark Summerfield, *Qt4 et C++ : Programmation d'interfaces GUI*, ed. Campus Press, 2007.
- [Clavel et al., 2000] Gilles Clavel, Nicolas Fagart, David Grenet, Jorge Miguéis,*C++ La synthèse - Concepts objet, standard ISO et modélisation UML*, ed. DUNOD, 2000.
- [Delannoy, 2017] Claude Delannoy, *Programmer en langage C++*, ed. Eyrolles, 9e ed., 2017.
- [Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johson, John Vlissides, *Design Patterns. Elements of reusable object-oriented software*, ed. Addison-Wesley, 1995.
- [Géron et Tawbi, 1999] Aurélien Géron, Fatmé Tawbi, *Pour mieux développer avec C++ - Design patterns, STL, RTTI et smart pointers*, ed. DUNOD, 1999.
- [Meyer, 2008] Bertrand Meyer , *Conception et programmation orientées objet*, ed. Eyrolles, 2ème ed, 2008.
- [Meyers, 2010] Scott Meyers, *Effective STL*, ed. Addison-Wesley, 2010.
- [Meyers, 2011] Scott Meyers, *Effective C++*, ed. Addison-Wesley, 3ème ed, 2011.
- [Meyers, 2014] Scott Meyers, *Effective Modern C++*, ed. O'Reilly Media, 2014.
- [Muller et Gaertner, 2003] Pierre-Alain Muller, Nathalie Gaertner, *Modélisation objet avec UML*, ed. Eyrolles, 2ème ed, 2003.
- 21天学通C++(第6版) 利伯蒂(Liberty.J.)、拉奥(Rao.S.)、琼斯(Jones.B.), 2007.
- http://www.cplusplus.com
- http://www.cppreference.com

# LEXIQUE

| **français** | **anglais** | **chinois** |
|---|---|---|
| parenthèse ouvrante ( | opening parenthesis | 左括号 |
| parenthèse fermante ) | closing parenthesis | 右括号 |
| parenthèses ( ) | parenthesis | 括号 |
| parentèse-parenthèse ( ) ou (( ou )) | parenthesis- parenthesis | 一对括号，两个括号 |
| crochet ouvrant [ | opening bracket | 左方括号 |
| crochet fermant ] | closing bracket | 右方括号 |
| crochets [ ] | brackets | 方括号 |
| crochet-crochet [ ] ou [[ ou ]] | bracket-bracket | 一对方括号，两个方括号 |
| accolade ouvrante { | opening brace | 左大括号 |
| accolade fermante } | closing brace | 右大括号 |
| accolades { } | braces | 大括号 |
| chevron ouvrant < | opening chevron | 左尖括号 |
| chevron fermant > | closing chevron | 右尖括号 |
| chevrons << ou >> | chevron-chevron | 两个尖括号 |
| point **.** | dot | 点，点号 |
| point virgule **;** | semi-colon | 分号 |
| virgule **,** | colon | 逗号 |
| flèche **->** | arrow | 箭头，右箭头 |
| plus **+** | plus | 加号 |
| plus-plus **++** | plus-plus | 两个加号 |
| moins **-** | minus | 减号 |
| moins-moins **--** | minus-minus | 两个减号 |
| étoile ***** | star | 星号 |
| arobas **@** | at | 艾特符号，at 号 |
| esperluette **&** | ampersand | 取址符 |
| dièse **#** | sharp | 井号 |
| | | |
| accesseur | accessor | 访问器 |
| affectation | assignment | 赋值 |
| allocation | allocation | 地址分配 |
| ambiguïté | ambiguity | 歧义 |
| argument | argument | 参数 |
| argument par défaut | default argument | 默认参数 |
| attribut | attribute | 属性 |
| donnée membre | member variable | 成员变量 |
| bogue | bug | 程序错误 |
| chaîne de caractère | character string | 字符串 |
| classe | class | 类 |
| classe ascendante | ancestor class, super-class | 祖先类 |
| classe dérivée, classe descendante | derived class, sub-class | 派生类 |

| | | |
|---|---|---|
| classe abstraite | abstract class | 抽象类 |
| compilateur | compiler | 编译器 |
| compilation | compilation | 编译 |
| compiler | to compile | 编译 |
| conversion | conversion | 转换 |
| conversion explicite | casting, explicit conversion | 显式转换 |
| copie | copy | 复制 |
| constructeur | constructor | 构造器 |
| contrainte | constraint | 约束 |
| conteneur | container | 容器 |
| convention | convention | 惯例 |
| déclaration | declaration | 声明 |
| déclencher | to trigger | 触发 |
| définir | to define | 定义 |
| destructeur | destructor | 析构器 |
| donnée | data | 数据 |
| encapsulation | encapsulation | 封装性 |
| ensemble | set | 集合 |
| entête | header | 头文件 |
| énumération | enumeration | 枚举 |
| exception | exception | 异常 |
| explicite | explicit | 明确的 |
| extensibilité | extensability, inheritability | 可扩展性 |
| fiabilité | reliability | 可靠性 |
| flot | stream | 流 |
| fonction | function | 函数 |
| fonction en ligne | inline function | 内联函数 |
| généralisation | generalization | 普遍化 |
| généraliser | to generalize | 普及 |
| générique | generic | 通用的 |
| héritage | inheritance | 继承性 |
| hiérarchie | hierarchy | 层次 |
| implémenter | to implement | 实现接口 |
| implicite | implicit | 隐式 |
| infixe | infix | 中缀 |
| initialiser | initialize | 初始化 |
| instance | instance | 实例 |
| interface | interface | 接口 |
| interpréteur | interpreter | 解释器 |
| liste chaîné | linked list | 链表 |
| message | message | 消息 |
| modèle | model, pattern | 模式 |
| modèle de conception | design pattern | 设计模式 |

| | | |
|---|---|---|
| modularité | modularity | 模块性 |
| objet | object | 对象 |
| opérateur | operator | 操作符 |
| paradigme de programmation | programming paradigm | 编程范式 |
| patron | template | 模板 |
| planter | to crash | 停止 |
| pluralité | plurality | 多元性 |
| pointeur | pointeur | 指针 |
| polymorphisme | polymorphism | 多态性 |
| portabilité | portability | 可移植性，平台无关 |
| postfix | postfix | 后缀 |
| preprocesseur | preprocessor | 预处理 |
| principe de substitution | substitution principle | 替换原则 |
| priorité | priority | 优先级 |
| procédural | procedural | 过程性的 |
| prototype | prototype | 原型 |
| récursif | recursive | 递归 |
| redéfinir | to override | 重定义 |
| redéfinition | overriding | 重定义 |
| référence | reference | 引用 |
| réutilisable | reusable | 可重用性 |
| statique | static | 静态的 |
| spécialisation | specialization | 专门化 |
| surcharge | to overload | 重载 |
| transtypage | transtypage, dynamic casting | 动态类型转换 |
| validité | validity | 有效性 |