Programmation Impérative Avancée

HLIN302

Pascal Giorgi & Alban Mancheron

Université de Montpellier

2018-2019

Présentation du module

- Volume horaire :
  - 13h cours (9 séances)
  - 16,5h TD (11 séances)
  - 16,5h TP (11 séances)

  Les TD/TP commencent dans 2 semaines ! ! !

- Évaluation :
  - CC = 1 contrôle sur machine
  - TP = projet noté avec rapport (en groupe)
  - EX = 1 examen final (avec 1 session de rattrapage)
  - note finale = 0.25 TP + max(0.75 EX, 0.5 EX+0.25 CC)

Présentation du module

Objectifs du module :

- Aller plus loin dans la connaissance d'un langage impératif
- Initiation à un langage évolué (multi-paradigme) : *le C++*
- À l'issue de ce cours, vous devez être capable de :
  - maîtriser complètement les principes d'un langage impératif
  - utiliser et définir des structures de données avancées
  - écrire des programmes complexes

Plan du cours

Plan du cours

Plan du cours

Programmation impérative

La programmation impérative procédurale correspond à la notion de changement d'état d'un programme à partir :

- d'instructions simples,
- d'appel d'algorithmes (i.e. procédure/fonction).

Changement d'état d'un programme

Cela correspond à la modification d'une donnée utilisée par le programme (une variable).

Remarque

Ce modèle de programmation est au plus proche des architectures de nos ordinateurs actuels.

Historique

Le langage C

- est un langage purement impératif
- est créé en 1972 par D. Ritchie et K. Thompson pour développer UNIX
- normalisation ANSI (1989) et ISO (1990,1991,2011)

Le langage C++

- est une évolution de C ajoutant le paradigme objet
- est inventé en 1983 par B. Stroustrup
- normalisation ISO (1998,2003,2011)

Caractéristiques importantes du C++

Noyau de langage impératif procédural

- faiblement typé (conversion implicite des types)
- langage compilé
- arithmétique sur les adresses mémoires et sur les bits

Remarque

- permet de rester très proche de l'architecture des ordinateurs,
- utile pour avoir les meilleures performances possibles

Concevoir un programme en C++

Trois étapes ordonnées :

1. écrire dans un fichier texte le code du programme en C++
2. compiler le programme
3. exécuter le programme

Écrire un programme C++

Il suffit d'écrire avec un éditeur de texte simple (sans formatage) quelques lignes de code.

exemple.cpp

```
1  #include <iostream>
2
3  int main(){
4     int a,b,c;
5     std::cin>>a>>b;
6     c=a+b;
7     std::cout<<a<<"+"<<b<<"="<<c<<std::endl;
8
9     return 0;
10 }
```

Compilation d'un programme C++

Objectif

Traduire un programme décrit en C++ dans un programme décrit en langage machine.

Attention

Les programmes écrits en C++ ne sont pas exécutables, seules leurs traductions en langage machine le sont.

L'outil permettant la traduction d'un texte écrit en C++ en langage machine s'appelle un compilateur.

Compilation d'un programme C++

Il existe plusieurs compilateurs C++ en fonction du système d'exploitation et de l'architecture.

Dans un environnement GNU/Linux,

- la suite de compilation standard est appelé GCC (the GNU Compiler Collection).
- le compilateur C++ de GCC est : g++

Un exemple de compilation

Pour compiler l'exemple (exemple.cpp), on tape dans un terminal :

```
g++ -Wall exemple.cpp -o Exemple
```

ce qui permet de créer un programme exécutable nommé Exemple.

Syntaxe générale

```
g++ -Wall source.cpp -o execu
```

- `source.cpp` est le fichier du programme en C++.
- `execu` est le programme traduit exécutable sur la machine.
- `-o ...` indique que `...` est le nom du programme.

Notion de programme en C++

Un programme est constituée de plusieurs fonctions/procédures qui peuvent s'appeler les unes les autres.

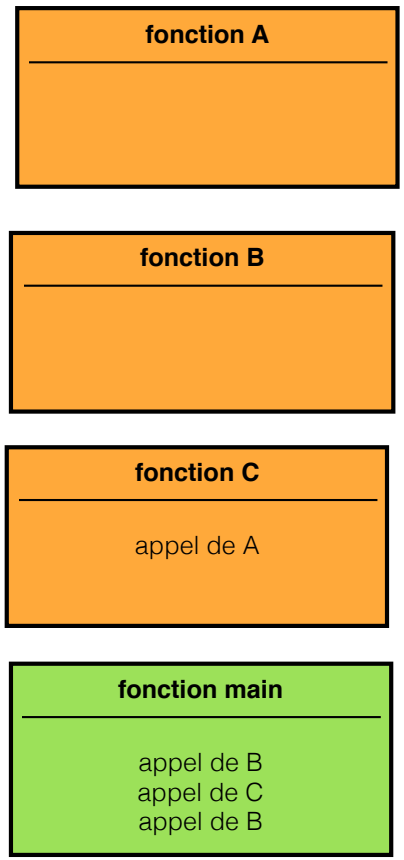
La première fonction appelée par un programme est :
le point d'entrée du programme

Point d'entrée d'un programme C++

Il correspond à une fonction appelée `main` qui est spécifique :

```
1 int main(){
2   // debut du programme
3   ...
4   // fin du programme
5   return 0;
6 }
```

Principe d'un programme C++



Écriture du programme :

1. On définit un ensemble de fonctions A,B,C.
2. On définit un programme avec main

Quand on lance le programme :

- il exécute main
  - qui exécute B
  - qui exécute C
    - qui lui même exécute A
  - qui exécute encore B

Attention

la fonction main n'est définie qu'une seule fois par programme.

Le langage C++ : modularité des fichiers

Les fichiers contenant du codes sources C++ sont :

- des "Headers" : structures et/ou signature de fonctions
  ↪ ne sont pas compilés, on les inclut par `#include`
- des "Sources" : corps des fonctions et/ou programme principal
  ↪ ne sont pas inclus, on les compile par `g++ -c`

Attention

En C++, les fichiers

- "Headers" ont une extension du type : `.h`, `.H`, `.hpp`, `hxx`
- "Sources" ont une extension du type : `.cc`, `.C`, `.cpp`, `.cxx`

Plan du cours

Programmation impérative = Manipulation mémoire

Rappel :

- la mémoire est binaire (un grand tableau de 0 et de 1)
- un langage impératif manipule les données via cette mémoire

Comment savoir ou se trouve les données en mémoire ?

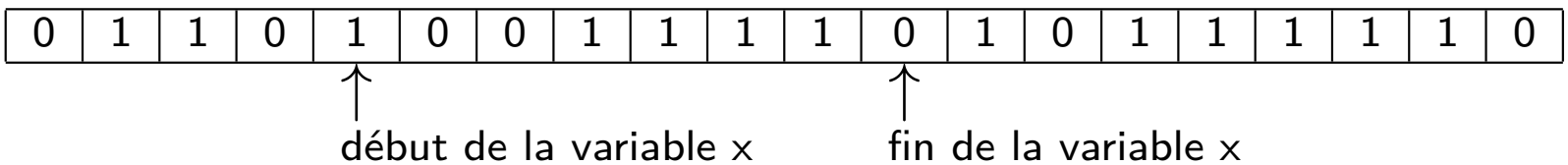
0	1	1	0	1	0	0	1	1	1	1	0	1	0	1	1	1	1	1	0

Programmation impérative = Manipulation mémoire

Rappel :

- la mémoire est binaire (un grand tableau de 0 et de 1)
- un langage impératif manipule les données via cette mémoire

Comment savoir ou se trouve les données en mémoire ? les variables

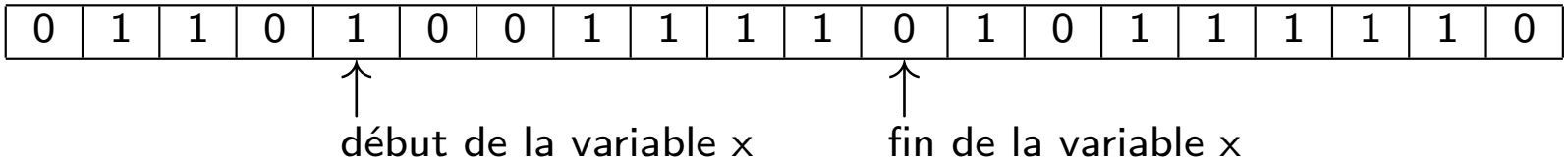


Programmation impérative = Manipulation mémoire

Rappel :

- la mémoire est binaire (un grand tableau de 0 et de 1)
- un langage impératif manipule les données via cette mémoire

Comment savoir ou se trouve les données en mémoire ? les variables



On identifie une variable par :

- une adresse de début dans l'espace mémoire
- une taille indiquant l'espace mémoire occupé par la variable

Les variables

Definition

On appelle variable une zone mémoire de l'ordinateur à laquelle on a donné un nom, ainsi qu'un type de données.

- le nom de la variable, appelé *identificateur*[1], permet de manipuler facilement les données stockées en mémoire.
- le type de données permet au compilateur de réserver l'espace mémoire suffisant pour stocker les valeurs.

(1) on utilise un identificateur plutôt que l'adresse mémoire mais l'on peut facilement récupérer l'adresse d'une variable à partir de son identificateur.

Les variables

Exemple

```
int a;
```

a est une variable de type entier :

- le compilateur réservera 4 octets en mémoire pour stocker ses valeurs
- on utilisera le nom « a » pour travailler avec l'espace mémoire attribué à la variable a.

Les variables de type tableau

La notion de tableau permet de manipuler plusieurs zones mémoires contigus au sein d'une même variable.

Exemple

```
int T[10];
```

T est une variable tableau de 10 entiers.

- la taille du tableau doit être connue à la compilation.
- on accède aux éléments du tableau par l'opérateur `[ ]`,
  ex : `T[i]` où i est un indice valide du tableau (commence à 0).

L'adresse mémoire d'une variable

Chaque variable est stockée en zone mémoire à une adresse précise. L'opérateur d'adresse `&` permet de récupérer l'adresse associée à une variable

- si `a` est une variable définie :
  `&a` renvoie la valeur de l'adresse de `a`

Attention

- l'adresse des variables n'est pas choisie par le programmeur :
  `&a=...` est interdit ! ! !
- l'adresse des variables peut être stocker dans les variables de type pointeur

Les variables de type pointeurs

La notion de pointeur permet de manipuler une zone mémoire non pas par son identificateur mais par son adresse dans la mémoire.

Definition

`type* var;`

`var` est un pointeur sur une zone mémoire de type `type`

Attention : `var` contient une adresse mémoire pas une valeur

```
1 int a;
2 int *ptr;
3 ptr = &a; // affectation valide
4 ptr = a;  // affectation non valide
```

Les variables de type pointeurs

Pour manipuler une zone mémoire par un pointeur, il faut utiliser l'opérateur de déréférencement `*`

Definition

```
type* var;
```

le déréférencement `*var` permet de manipuler la zone mémoire d'adresse mémoire `var`.

```
1 int a;
2 int* ptr;
3 ptr = &a; // affecte ptr avec l'adresse de a
4 *ptr = 3; // affecte la variable a avec la valeur 3
```

Plan du cours

Les types de données en C++

- **des nombres** : entiers (naturels ou relatifs), approximations flottantes des réels
- **des lettres** : caractères
- **des booléens**

type	**type en C++**	**valeurs**
entiers naturels	`unsigned int`	$[0, 2^{16} - 1]$ ou $[0, 2^{32} - 1]$
entiers relatifs	`int`	$[-2^{15}, 2^{15} - 1]$ ou $[-2^{31}, 2^{31} - 1]$
réels (simple précision)	`float`	$\approx$ 7 chiffres significatifs
réels (double précision)	`double`	$\approx$ 15 chiffres significatifs
caractère	`char`	
booléen	`bool`	{true,false}

Les entiers en C++

Il existe différentes variantes pour les entiers qui utilisent plus ou moins d'octets (norme c++11) :

type entier	**type en C++**	**nbr octets**	**valeurs**
court	`short int`	≥ 2	$[-2^{15}, 2^{15}-1]$
standard	`int`	≥ 2	$[-2^{15}, 2^{15}-1]$
long	`long int`	≥ 4	$[-2^{31}, 2^{31}-1]$
très long	`long long int`	≥ 8	$[-2^{63}, 2^{63}-1]$

En précédant les types entiers du C++ par `unsigned` on obtient les versions des types non signés (positif).

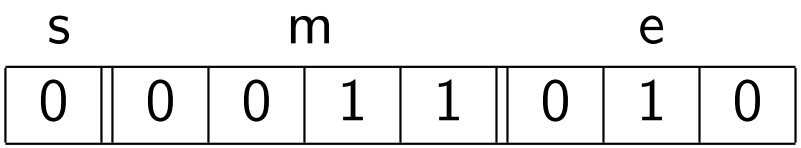
Les réels en C++

Les réels ne sont pas représentables en codage binaire. On représente un réel x par une approximation rationnelle particulière :

$$x \approx (-1)^s \times \frac{m}{2^e}$$

exemple : $0.75 = (-1)^0 \times \frac{3}{2^2}$

Le codage de x correspond au codage binaire de s,m et e.



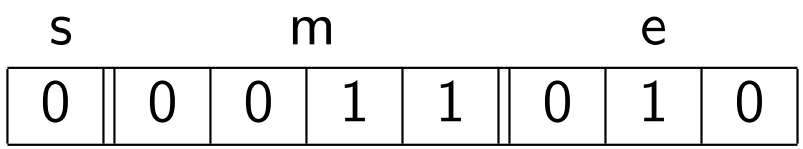
Les réels en C++

Les réels ne sont pas représentables en codage binaire. On représente un réel x par une approximation rationnelle particulière :

$$x \approx (-1)^s \times \frac{m}{2^e}$$

exemple : $0.75 = (-1)^0 \times \frac{3}{2^2}$

Le codage de x correspond au codage binaire de s,m et e.



Norme IEEE754 :

- `float` (32 bits) : 1 bit pour s, 23 bits pour m, 8 bits pour e
- `double` (64 bits) : 1 bit pour s, 52 bits pour m, 11 bits pour e

Les booléens en C++

Les booléens : `bool`

- valeurs : `true` ou `false`
- par définition `true` correspond à l'entier 1 et `false` à 0
- tout entier non nul sera interprété comme `true`, alors que 0 comme `false`

```
1 bool a, b;
2 a = true;
3 b = a || false;
4 a = b && 1;
```

Les types entiers adaptatifs

Pour garantir la portabilité des codes, on peut utiliser un alias pour les types d'entiers.

- `size_t` : plus grand entier non-signé représentable
- `ptrdiff_t` :plus grand entier signé représentable

On utilise souvent `size_t` pour les tailles de tableau, de chaînes de caractères, ...

Les opérateurs

Attention

si les opérandes ne sont pas du même type, il y aura une conversion implicite ! ! !

Arithmétiques :

- addition : +
- soustraction : -
- division : /
- multiplication : *
- négation : -
- modulo : %

Comparaisons :

- égalité : ==
- différent : !=
- supérieur : >
- inférieur : <
- supérieur ou égal : >=
- inférieur ou égal : <=

Les opérateurs

Attention

si les opérandes ne sont pas du même type, il y aura une conversion implicite ! ! !

Logiques :

- et : &&
- ou : ||
- non : !

- et binaire : &
- ou binaire : |
- non binaire : ~
- décalages de bits : << et >>

Les opérandes sont soit des booléens soit des types numériques.

L'opérateur d'affectation =

- `var = exp`
  - `exp` : expression du même type que `var`
  - `var` : identifiant d'une variable déclarée
  - la variable `var` prend la valeur de l'expression `exp`
  - ex : `a = 2 + 3;` `a` prend la valeur de l'expression `2 + 3 = 5`

Attention

- `var` doit correspondre à une zone mémoire
  - l'affectation `a+b=3;` n'est pas correcte.
  - l'affectation `*ptr=3` est correcte, si `ptr` est un pointeur.
- si `var` et `exp` ne sont pas du même type, il y une conversion implicite.

Autres opérateurs

- incrémentation/décrémentation d'une variable entière `a` :
  - `a++` incrémente la valeur de `a` par 1
  - `a--` décrémente la valeur de `a` par 1
- affectation élargies : `+=`, `-=`, `*=`, `/=`
  - `a += 3;` correspond à l'expression `a = a + 3;`
- taille mémoire des variables : `sizeof`
  - `sizeof(a)` renvoie la taille en octet de la variable `a`.
- et beaucoup d'autres...

L'instruction `if ··· else`

Definition

```
if (exp) {
   instr1
} else {
   instr2
}
```

- `exp` est une expression booléenne
- `instr1` et `instr2` sont une ou plusieurs instructions

L'instruction `for`

Definition

```
for (exp1; exp2; exp3) {
     instr;
}
```

- `exp1` est une **expression quelconque** évalué une seule fois au début de la boucle
- `exp2` est une **expression booléenne** qui permet d'arrêter la boucle
- `exp3` est une **expression quelconque** évaluée à chaque tour de boucle (en dernier).

- `instr` est une instruction ou un bloc d'instructions

L'instruction `for` : schéma d'exécution

Definition

```
for (exp1; exp2; exp3) {
     instr;
}
```

1. exp1

   ...
2. exp2 → sort de la boucle si exp2=false
3. instr
4. exp3

   ...
5. exp2 → sort de la boucle si exp2=false
6. instr
7. exp3

L'instruction `for` : ce qui ne faut pas faire

Attention au boucle qui ne se terminent jamais ! ! !
les boucles infinies. . .

```
1 int i, s;
2 for (i = 1 ; i < 11 ; s = s + i)
3    ...
```

→ la variable de boucle n'est pas incrémentée

```
1 int i, s;
2 for (i = 1 ; i != 10 ; i += 2)
3    ...
```

→ la condition d'arrêt de la boucle n'est jamais atteinte

L'instruction `while`

Definition

```
while (exp){
  instr;
}
```

- `exp` est une expression booléenne permettant de contrôler la boucle
- `instr` est une instruction ou un bloc d'instructions

L'instruction `while` : schéma d'exécution

Definition

```
while (exp){
   instr;
}
```

1. `exp` → sort de la boucle si `exp`=false
2. `instr`

   ...
3. `exp` → sort de la boucle si `exp`=false
4. `instr`

   ...
5. `exp` → sort de la boucle si `exp`=false
6. `instr`

   ...

L'instruction `do {} while`

Parfois, il est souhaitable d'exécuter le corps de boucle avant la condition de boucle (`instr` avant `exp`).

Dans ce cas, on peut utiliser l'instruction `do {} while;`

Definition

```
do {
   instr;
} while (exp);
```

- `instr` et `exp` sont identiques à ceux utilisés dans la boucle `while` classique

L'instruction `do {} while` : schéma d'exécution

Definition

```
do {
   instr;
}
while (exp);
```

1. `instr`
   ...
2. `exp` → sort de la boucle si `exp`=false
3. `instr`
   ...
4. `exp` → sort de la boucle si `exp`=false
5. `instr`
   ...

Plan du cours

Les espace de noms : `namespace`

Il est possible en C++ de rattacher des fonctions et des variables à un espace de nom particulier.

Objectif : permettre de partager le même nom de fonction entre plusieurs bibliothèques (ou bouts de programme)

Exemple

considérons que les fichiers titi.h et toto.h définissent tout les deux la fonction `void affiche();` et que j'ai besoin de ces 2 fichiers dans mon programme.

Les espace de noms : namespace

Quelle fonction va être utilisée par mon programme ?

```
1 #include "titi.h"
2 #include "toto.h"
3 int main() {
4    affiche();
5    return 0;
6 }
```

Les espace de noms : namespace

Quelle fonction va être utilisée par mon programme ?

```
1 #include "titi.h"
2 #include "toto.h"
3 int main() {
4   affiche();
5   return 0;
6 }
```

aucune des 2 car il y aura une erreur de compilation :

```
error: redefinition of ‘void affiche()‘
```

Cela signifie que des codes peuvent être exclusif entre eux, ce qui n'est pas souhaitable.

Les espace de noms : namespace

L'utilisation de `namespace` permet d'enlever l'exclusivité de nom dans les programmes.

Definition

```
namespace ident{
   fonction1
   ...
}
```

- `fonction1` et `...` sont attachées à l'espace de nom `ident`
- pour les utiliser il faut précéder leur nom par `ident::` (ex : `ident::fonction1`)
- on peut utiliser une directive plus globale en début de fichier `using namespace ident;`
  ↪ uniquement dans les fichiers sources (.cpp)

Les entrées-sortie

Comme pour tout langage de programmation il est souhaitable de pouvoir interagir avec le programme :

- saisir des valeurs au clavier
- afficher à l'écran des variables

En C++, les fonctionnalités d'entrée-sortie standards sont définies dans le fichier `iostream` et appartiennent à l'espace de nom `std`.

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4    ...
5    return 0;
6 }
```

Instruction d'affichage

Definition

```
std::cout << exp;
```

- exp est une expression quelconque
- std::cout est le nom de la sortie standard (l'écran par défaut)
- l'opérateur d'écriture << indique ici d'envoyer la valeur de l'expression exp sur le flux de sortie standard cout

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4   cout << 1;
5   return 0;
6 }
```

Instruction d'affichage

Definition

```
std::cout << exp;
```

exp peut être :

- expression arithmétique
- expression booléenne (affiche 1 pour vrai 0 pour faux)
- une constante ou une variable de type standard
- `std::endl` : instruction de retour à la ligne

On peut enchaîner les affichages : `std::cout << exp1 << exp2 << exp3 << ··· << expn;`

Instruction d'affichage : exemple

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4    int a = 18;
5    float b = 2.3;
6    cout << "l'entier a = " << a << endl;
7    cout << "le flottant b = " << b << endl;
8    return 0;
9 }
```

ce programme affichera à l'écran :

```
l'entier a = 18
le flottant b = 2.3
```

Instruction de saisie clavier

Definition

```
std::cin >> var;
```

- `var` est un identificateur de variable valide
- `std::cin` est le nom de l'entrée standard (le clavier)
- l'opérateur de lecture >> indique ici d'affecter la valeur de l'entrée standard dans la variable `var`

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4   int a;
5   cin >> a;
6   return 0;
7 }
```

Instruction de saisie clavier

Definition

```
cin >> var;
```

- `var` est un identificateur de variable valide

`var` peut être :

- le nom d'une variable déjà déclaré
- un élément de tableau déjà alloué

Remarque

On peut enchaîner les saisies clavier :

```
std::cin >> var1 >> var2 >> ··· >> varn;
```

↪ séparation des valeurs par {espace, retour ligne, tabulation}

Instruction de saisie clavier : exemple

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4    int a;
5    float b;
6    cout << "Entrez un entier puis un flottant: ";
7    cin >> a >> b;
8    return 0;
9 }
```

ce programme affichera à l'écran :

```
Entrez un entier puis un flottant
```

et attendra la saisie au clavier d'un entier et d'un réel qu'il affectera respectivement à la variable `a` et `b`.

Le type chaîne de caractère

Le type chaîne de caractère : `std::string`

- pas un type natif du langage : besoin du `#include<string>`
- basé sur un tableau dynamique de caractères

La déclaration `std::string t;` définit la variable `t` comme une chaîne de caractère (vide par défaut).

Construction de chaînes non vides :

- `std::string s('a',10);`
- `std::string mot("bonjour");`
- `std::string mot = "bonjour";`

Le types chaîne de caractère

Fonctionnalité sur les variables de type `string` :

- `s.length()` et `s.size()` donne la longueur de `s`
- `s[i]` est le i-ème caractère de s
- `s+t` retourne une nouvelle chaîne concaténant `s` et `t`

Entrée-Sortie avec `string`

- `std::cout << s` affiche la chaîne `s`
- `std::cin >> s` affecte `s` avec la chaîne de caractère saisie

string : exemple

```
1  #include <iostream>
2  #include <string>
3
4  int main() {
5     std::string mot, phrase;
6     do {
7        std::cin >> mot;
8        phrase = phrase + ".." + mot;
9     } while (mot != "fin");
10
11    std::cout << phrase << std::endl;
12
13    return 0;
14 }
```

Plan du cours

Définition de fonctions

Definition

```
type_retour nom(liste des paramètres formels){
   corps
}
```

- `nom` correspond au nom donné à la fonction
- `type_retour` est le type du résultat de la fonction (`void` si la fonction ne renvoie rien)
- `liste des paramètres formels` est la liste des variables d'entrée de la fonction
- `corps` correspondant aux instructions effectuées par la fonction en fonction des paramètres formels.

Fonctions : paramètres formels

La liste des paramètres formels d'une fonction est :

- vide si la fonction n'a aucun paramètre
- de la forme : $type_1$ `p1, ..., ` $type_n$ `pn`

Les couples $type_i$ `pi` sont de la forme :

- `type pi` pour une variable normale
- `type pi[k]` pour un tableau à k éléments
- `type* pi` pour une pointeur ou un tableau

où `type` est un type connu (type de base, structure ou classe) et `pi` est le nom de la variable formelle.

Appel de fonction

Definition

```
var = nom(liste des paramètres effectifs);
ou
nom(liste des paramètres effectifs);
```

- la liste des paramètres effectifs correspond à l'ensemble des variables et des constantes que l'on souhaite donner comme argument à la fonction. ex : `max(2, 3)`.

Attention : le passage des arguments se fait par copie
la valeur des paramètres effectifs est copiée dans les variables formelles correspondantes.

Appel de fonction : exemple

```
1  #include <iostream>
2
3  int max(int a, int b){
4     if (a > b) return a; else return b;
5  }
6  int main() {
7     int x, y;
8     x = 3;
9     y = 5;
10    cout << "le max est : " << max(x, y) << endl;
11    return 0;
12 }
```

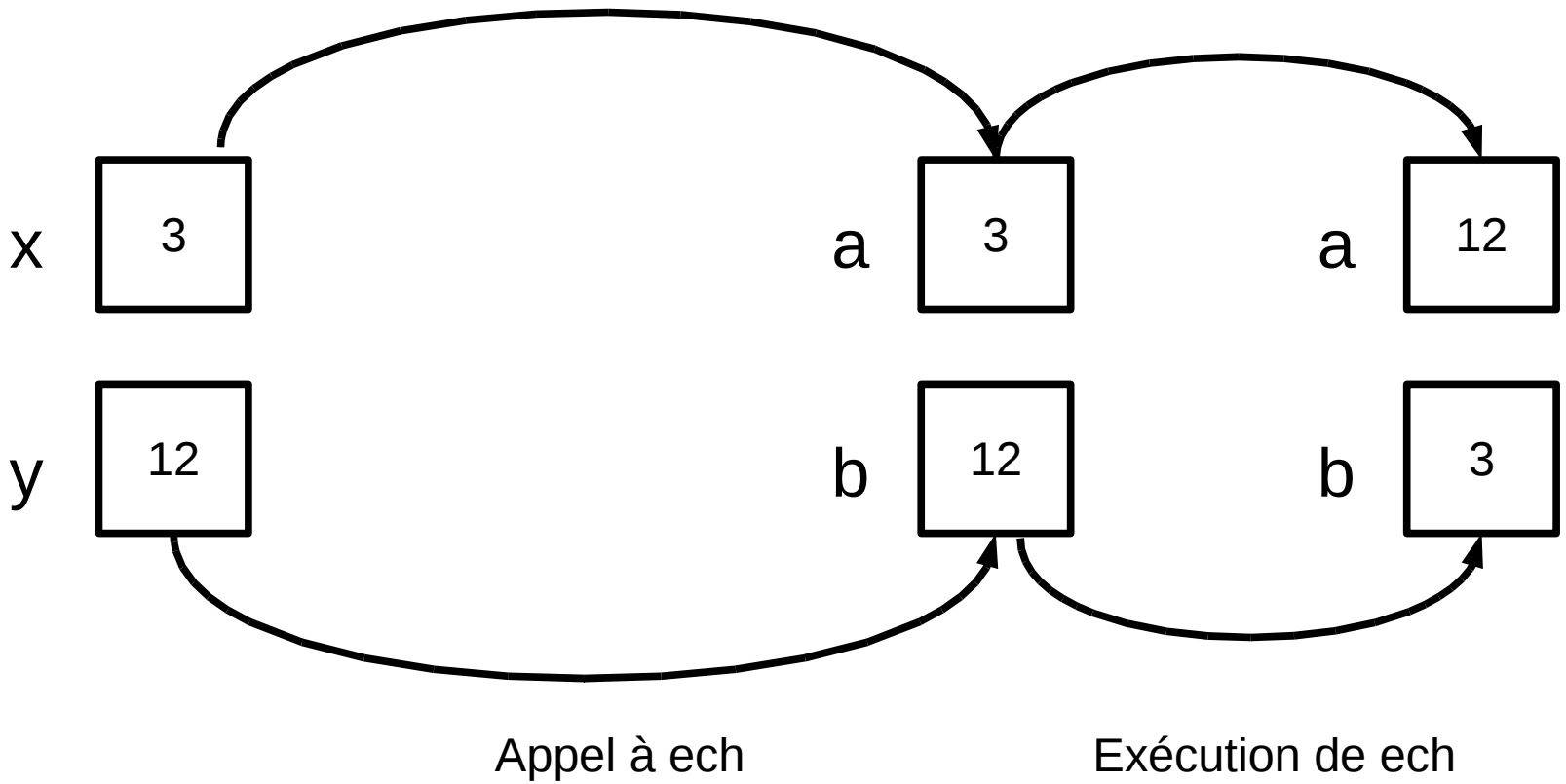
Fonction : passage de paramètres par copie

Par défaut, les paramètres d'une fonction sont initialisés par une copie des valeurs des paramètres réels.

Modifier la valeur des paramètres formels dans le corps de la fonction ne change pas la valeur des paramètres réels.

```
1 void ech(int a, int b){
2   int r;
3   r = a; a = b; b = r;
4 }
5 int main() {
6   int x, y;
7   x = 3; y = 12;
8   ech(x, y); // ne change pas la valeur de x et de y
9 }
```

Fonction : passage de paramètres par copie

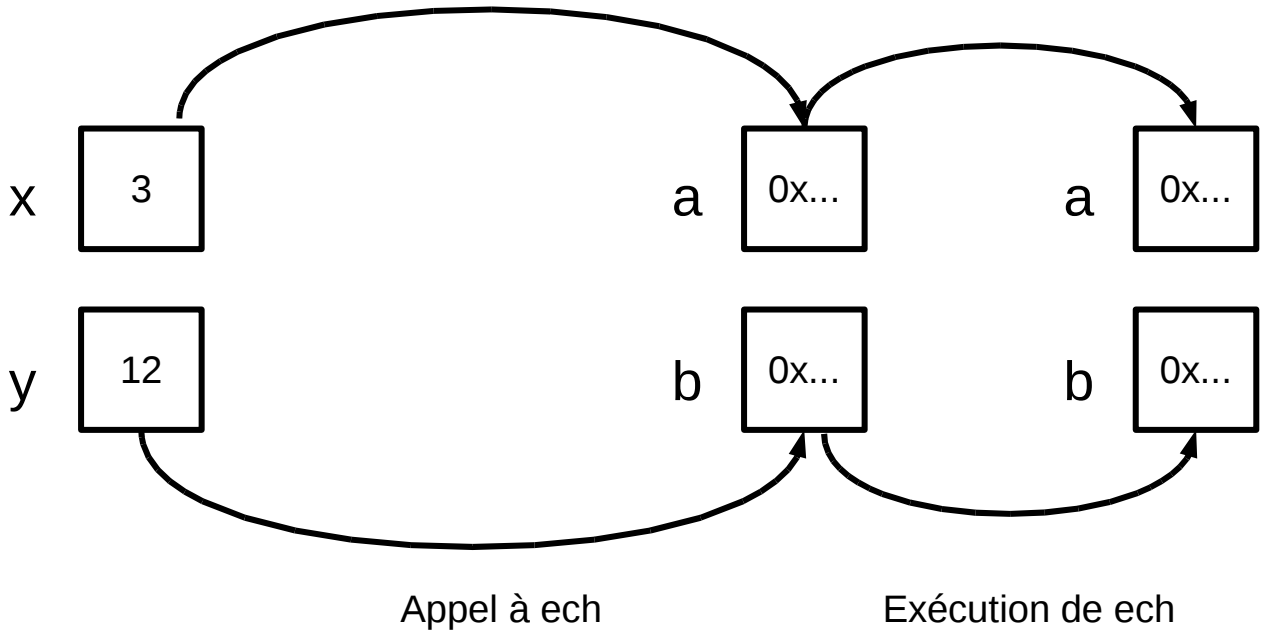


Fonction : paramètre modifiable

Une solution : utiliser les pointeurs

```
1 void ech(int* a, int *b){
2   int r;
3   r = *a; *a = *b; *b = r;
4 }
5 int main() {
6   int x, y;
7   x = 3; y = 12;
8   ech(&x, &y); // la valeur de x et de y sont echangees
9 }
```

Fonction : paramètre modifiable



Avec les pointeurs, c'est pareil on copie :
a et b sont des copies de &x et &y

La seule différence est que la variable x et *a représente la même zone mémoire : donc modifier l'un modifie l'autre.

Fonction : passage de paramètres par référence

En C++, on peut choisir de ne pas passer les paramètres par copie :

$$\Rightarrow \text{passage par référence}$$

Il suffit de précéder le nom d'un paramètre formel par & pour indiquer le passage par référence :

- `type& var` comme paramètre formel dans une fonction indique que l'on substituera la variable `var` par la variable effective lors de l'appel.
- le passage par référence n'est pas indiqué lors de l'appel

Les paramètres passés par référence sont donc modifiables ! ! !

Fonction : passage de paramètres par référence

```
1 void ech(int& a, int &b){
2   int r;
3   r = a; a = b; b = r;
4 }
5 int main() {
6   int x, y;
7   x = 3; y = 12;
8   ech(x, y); // la valeur de x et de y sont echangees
9 }
```

Fonction : passage de paramètres par référence

- Il faut que les paramètres effectifs soient compatibles avec un passage par référence...

  `ech(x, 4)` n'est pas possible !

- Il y a possibilité de forcer la non-modification de variables passées par référence : `const type& var`.

  quel intérêt ? ? ?

Fonction : passage de paramètres par référence

- Il faut que les paramètres effectifs soient compatibles avec un passage par référence...

  `ech(x, 4)` n'est pas possible !

- Il y a possibilité de forcer la non-modification de variables passées par référence : `const type& var`.

  quel intérêt ? ? ?
  $\rightarrow$ éviter la copie des variables, ce qui pourrait être coûteux...

Plan du cours

Plan du cours

Plan du cours

La modélisation d'un problème en algorithmique nécessite l'introduction de structure de données :
$\rightarrow$ introduction de niveau d'abstraction.

Intérêts en programmation :

- introduit de la modularité dans les programmes
- facilite l'écriture et la maintenance des programmes
- les codes sont plus expressifs

Besoin de structure : exemple

On veut écrire un programme qui gère des comptes bancaires.
Ce dont on a besoin :

- stocker un compte : numéro, intitulé, solde, ...
- effectuer des opérations sur un compte : créditer, débiter

Nécessité de regrouper les données d'un compte :
comme en C, on peut utiliser les `struct`

```
1 struct compte {
2   int      numero;
3   string   intitule;
4   float    solde;
5 };
```

Besoin de structure : exemple

En plus de la structure, il faut définir les fonctions associées :

```
1 struct compte {
2   int      numero;
3   string   intitule;
4   float    solde;
5 };
6
7 void crediterCompte(struct compte &c, float f){
8   c.solde = c.solde + f;
9 }
```

Inconvénients des `struct`

- Les types de données et leurs fonctions associées sont totalement décorrélés dans le programme.

- Utiliser les types de donnée nécessite dans la plupart des cas la connaissance de la structure sous-jacente.

Plan du cours

Qu'est-ce qu'une classe ?

Une classe est **un nouveau type du langage** ajouté par le programmeur.

Les classes permettent :

- regrouper des données au sein d'une même entité
- associer des fonctions à cette entité
- restreindre l'accès à cette entité

Les entités représentées par une classe sont des objets.

Notions d'objets et de classe

Definition

Un objet est une entité qui regroupe :

- des données membre : les attributs
- des fonctions membre : les méthodes

Les méthodes :

- sont des fonctions qui s'appliquent à l'objet et à ses attributs.
- représentent les fonctionnalités de l'objet : son comportement.

Notions d'objets et de classe

Definition

Une classe est un moule d'objet :

- décrit les attributs et les méthodes d'une famille d'objets.
- définit en particulier un type d'objet

Les objets d'une même classe auront des attributs de même type, mais avec des valeurs différentes.

Notions d'objets et de classe

- La déclaration d'une classe correspond à la description générale d'une famille d'objet : attributs et méthodes.
- La déclaration d'un objet correspond à déclarer une instance particulière de la classe (au travers d'une variable).

Définition d'une classe

Definition

Une classe est définie par :

- un identificateur (son nom)
- des attributs
- des méthodes

En C++ la syntaxe est :

```
class identificateur
{
   attributs
   méthodes
};
```

Les attributs d'une classe

Chaque attribut représente une donnée précise de l'objet que l'on souhaite représenter : identifié par un nom et un type.

Exemple

pour un compte bancaire on aura 3 attributs :

- `numero` : un entier
- `intitule` : une chaîne de caractère
- `solde` : un nombre à virgule flottante (ou un entier ?)

NB : L'ensemble des attributs constitue l'espace mémoire de l'objet (contigu).

Les méthodes d'une classe

Chaque méthode définit une fonctionnalité des objets de la classe.

Exemple

pour un compte bancaire on aura 2 méthodes :

- créditer le compte
- débiter le compte

Les méthodes ont la même syntaxe que les fonctions.

Attention : *l'objet sur lequel agit la méthode n'est pas passez en paramètre, il est implicite.*

Une première classe en C++

La classe compte bancaire : `compte.h`

```
1  #include <string>
2  using namespace std;
3
4  class CompteB {
5    public:
6      // attributs
7      int      numero;
8      string   intitule;
9      float    solde;
10
11     // methodes
12     void crediter(float);
13     void debiter(float);
14 };
```

Comment utiliser un objet d'une classe

Comme pour les `struct`, on utilise la notation pointée.

Definition

Si `var` est une instance d'une classe

- `var.nomAttribut` accède à l'attribut correspondant
- `var.nomMéthode(...)`appelle la méthode correspondante

on dit que `var` est l'instance appelante.

Exemple :

```
1 CompteB cmpt;
2 cmpt.numero = 10013;
3 cmpt.crediter(1000);
```

Comment écrire les méthodes d'une classe

Un nom d'attribut dans le corps d'une méthode désigne un attribut de l'instance appelante.

```
1 class CompteB {
2   public:
3     int      numero;
4     string   intitule;
5     float    solde;
6
7     void crediter(float f)
8     { solde += f;}
9 };
```

```
CompteB cmpt;
cmpt.crediter(1000);
```

L'appel à la méthode `crediter` par l'objet `cmpt` implique que l'on peut remplacer `solde` dans la ligne 8 par `cmpt.solde`.

NB : *cela marche pareil avec l'appel de méthode dans une autre méthode.*

Un premier programme avec des classes

fichier : compte.h

```
1  #include <string>
2  using namespace std;
3
4  class CompteB {
5    public:
6      int      numero;
7      string   intitule;
8      float    solde;
9
10     void crediter(float f)
11     { solde+=f;}
12     void debiter(float f)
13     { solde-=f;}
14 };
```

fichier : main.cpp

```
1  #include <string>
2  using namespace std;
3  #include "compte.h"
4  int main(){
5    CompteB cmpt;
6
7    cmpt.numero= 1;
8    cmpt.intitule="Dupond";
9    cmpt.solde=500;
10
11   cmpt.crediter(200);
12   cmpt.crediter(650);
13   cmpt.debiter(425);
14   ...
15   return 0;
16 }
```

L'instance appelante dans une classe

L'instance courante d'une classe est implicite dans la définition des méthodes. Est-il possible de la manipuler explicitement ?

L'instance appelante dans une classe

L'instance courante d'une classe est implicite dans la définition des méthodes. Est-il possible de la manipuler explicitement ?

OUI : il existe dans les classes un pointeur appelé `this` qui est initialisé avec l'adresse mémoire de l'instance courante.

On peut donc écrire dans la classe `CompteB` :

```
1 void crediter(float f)
2 { this->solde += f;}
```

Plan du cours

Constructeurs

L'initialisation des attributs d'un objet est effectuée par une méthode particulière appelée : le constructeur

Definition

Le constructeur est une méthode appelée automatiquement lors de la construction de l'objet.

- il a le même nom que la classe
- il n'a pas de type de retour
- il peut avoir des paramètres

Une classe peut définir plusieurs constructeurs avec des paramètres différents (il y a surcharge de méthode).

Constructeurs

```
1  class CompteB {
2    public:
3      int      numero;
4      string   intitule;
5      float    solde;
6      CompteB(){
7        numero   = 0;
8        intitule = "vide";
9        solde    = 0;
10     }
11     CompteB(int n, string i, float s){
12       numero   = n;
13       intitule = i;
14       solde    = s;
15     }
16     ...
17 };
```

appel au constructeur : `CompteB cmpt1;`
`CompteB cmpt2(1, "Mr Durand", 15.3);`

Constructeurs

Si aucun constructeur n'est présent dans la définition de la classe, le compilateur en génère un par défaut :

- qui n'a pas d'argument
- et qui appelle les constructeurs sans argument des attributs

Attention : si il y a au moins un constructeur dans la classe, le compilateur ne génère pas le constructeur par défaut. C'est au programmeur de l'expliciter si besoin.

Destructeur

La destruction des attributs d'un objet est effectuée par une méthode particulière appelée : le destructeur

Definition

Le destructeur est une méthode appelée automatiquement lorsque l'objet doit être détruit.

- le destructeur est désigné par le nom de la classe précédé de ~
- il n'a pas de type de retour
- il n'a pas de paramètres

Une classe ne possède qu'un seul destructeur.

Destructeur

```
1 class CompteB {
2   public:
3     int      numero;
4     string   intitule;
5     float    solde;
6
7     ~CompteB() {}
8     ...
9 };
```

Le destructeur est appelé automatiquement lorsque l'objet est détruit.
Quand un objet est-il détruit ?

Durée de vie d'un objet

Un objet (non dynamique) est détruit à la fin du bloc dans lequel il a été déclaré

```
1  int main() {
2    CompteB cmpt1(11, "toto", 1500);
3
4    for (int i = 0 ; i < 10 ; i++) {
5      CompteB cmpt2(i, "titi", i * 100);
6      cmpt2.afficher();
7    } // destruction de cmpt2
8
9    cmpt1.afficher();
10   return 0;
11 } // destruction de cmpt1
```

Plan du cours

Déclaration *vs* Définition de fonctions

En C++, comme en C, on fait la distinction entre la déclaration et la définition de fonctions :

- la déclaration consiste à déclarer l'existence d'une fonction au travers de sa signature.
- la définition consiste à spécifier le code du corps de la fonction.

Afin de fournir une interface claire aux utilisateurs, on sépare la déclaration d'une fonction de sa définition.
C'est la même chose avec les méthodes dans les classes

Les fichiers *headers* et *sources*

Dans une classe, on sépare la définition des méthodes et leurs déclaration dans 2 fichiers distincts :

- `NomClasse.h` déclarations des attributs et des méthodes
- `NomClasse.cpp` définitions des méthodes

Le fichier *header* : NomClasse.h

Definition

```
class NomClasse {
   attributs
   signature des méthodes
};
```

CompteB.h

```
1 class CompteB {
2   public:
3     int      numero;
4     string   intitule;
5     float    solde;
6     void crediter(float);
7     void debiter(float);
8 };
```

Le fichier source : NomClasse.cpp

Definition

```
#include "NomClasse.h"
type_retour NomClasse::nomMethode(paramètres) {
   corps de la méthode
}
```

CompteB.cpp

```
1 #include "CompteB.h"
2
3 void CompteB::crediter(float f){
4    solde+=f;
5 }
6 void CompteB::debiter(float f){
7    solde-=f;
8 }
```

Compilation séparée

On compile uniquement le fichier source d'une classe :

```
g++ -c NomClasse.cpp
```

cela génère le fichier objet `NomClasse.o`.

Quand on souhaite utiliser la classe `NomClasse` :

- on inclut le header `NomClasse.h`
- et pour un programme `prog.cpp` :
  - on compile le programme seul : `g++ -c prog.cpp`
  - on fait l'édition de lien : `g++ prog.o NomClasse.o -o prog`

Compilation séparée

On compile uniquement le fichier source d'une classe :

```
g++ -c -Wall NomClasse.cpp
```

cela génère le fichier objet `NomClasse.o`.

Quand on souhaite utiliser la classe `NomClasse` :

- on inclut le header `NomClasse.h`
- et pour un programme `prog.cpp` :
  - on compile le programme seul : `g++ -c prog.cpp`
  - on fait l'édition de lien : `g++ prog.o NomClasse.o -o prog`

Compilation séparée

On compile uniquement le fichier source d'une classe :

```
g++ -c -Wall -ansi NomClasse.cpp
```

cela génère le fichier objet `NomClasse.o`.

Quand on souhaite utiliser la classe `NomClasse` :

- on inclut le header `NomClasse.h`
- et pour un programme `prog.cpp` :
  - on compile le programme seul : `g++ -c prog.cpp`
  - on fait l'édition de lien : `g++ prog.o NomClasse.o -o prog`

Compilation séparée

On compile uniquement le fichier source d'une classe :

```
g++ -c -Wall -ansi -pedantic NomClasse.cpp
```

cela génère le fichier objet `NomClasse.o`.

Quand on souhaite utiliser la classe `NomClasse` :

- on inclut le header `NomClasse.h`
- et pour un programme `prog.cpp` :
  - on compile le programme seul : `g++ -c prog.cpp`
  - on fait l'édition de lien : `g++ prog.o NomClasse.o -o prog`

Compilation séparée

On compile uniquement le fichier source d'une classe :

```
g++ -c -Wall -ansi -pedantic -g NomClasse.cpp
```

cela génère le fichier objet `NomClasse.o`.

Quand on souhaite utiliser la classe `NomClasse` :

- on inclut le header `NomClasse.h`
- et pour un programme `prog.cpp` :
  - on compile le programme seul : `g++ -c prog.cpp`
  - on fait l'édition de lien : `g++ prog.o NomClasse.o -o prog`

Compilation séparée

On compile uniquement le fichier source d'une classe :

```
g++ -c -Wall -ansi -pedantic -g NomClasse.cpp
```

cela génère le fichier objet `NomClasse.o`.

Quand on souhaite utiliser la classe `NomClasse` :

- on inclut le header `NomClasse.h`
- et pour un programme `prog.cpp` :
  - on compile le programme seul : `g++ -c prog.cpp`
  - on fait l'édition de lien : `g++ prog.o NomClasse.o -o prog`

Gestion des inclusions multiples

Lorsqu'on inclut plusieurs fois le même header dans un programme avec `#include`, cela génère une erreur de compilation :

```
In file included from prog.cpp:2:
CompteB.h:4: error: redefinition of ‘class CompteB’
CompteB.h:4: error: previous definition of ‘class CompteB’
```

Afin de résoudre ce problème, on utilise le préprocesseur :

- `#define` → définition de variable
- `#ifdef`,`#ifndef`,`#else`, `#endif` → op. conditionnels

Gestion des inclusions multiples

Pour chaque description de classe (ou *header*) on définie une variable préprocesseur quelconque correspondante (*e.g.*, NomClasse_H)

CompteB.h

```
1 #ifndef CompteB_H
2 #define CompteB_H
3
4 class CompteB {
5     ...
6 };
7 #endif
```

On teste l'existence de cette variable avant toute définition dans le *header* au cas ou le fichier aurait déjà été inclus.

Plan du cours

Plan du cours

Plan du cours

Les objets membres

Definition

On désigne par objet membre, un objet qui est attribut d'un autre objet.

```
1 class Personne { ... };
2
3 class BinomeP {
4   public:
5     Personne first;
6     Personne second;
7 };
```

Les deux classes Personne et BinomeP sont reliées par une relation de composition : les objets BinomeP sont dit « composites ».

Constructeur des objets composites

Quand un constructeur d'une classe est appelé, il effectue dans l'ordre :

1. appel des constructeurs par défaut des attributs
2. exécute ses propres instructions

Si l'on veut faire appel aux constructeurs paramétrés des attributs, il faut le spécifier explicitement dans la définition du constructeur :

Definition

```
NomClasse(param) : nomAttribut1(...), nomAttribut2(...) {
   instructions du constructeur
};
```

Constructeur des objets composites

```
1  class Personne {
2    public:
3      string nom;
4      string prenom
5
6      Personne(string n, string p) : nom(n), prenom(p) {
7      }
8  };
9
10 class BinomeP {
11   public:
12     Personne first;
13     Personne second;
14
15     BinomeP(string n1, string p1, string n2, string p2)
16     : first(n1,p1), second(n2,p2) { }
17  };
```

Destructeur des objets composites

Lors de la destruction d'un objet composite, le destructeur fait dans l'ordre :

1. exécute ses instructions
2. appel aux destructeurs des attributs

Plan du cours

Déclaration d'un tableau d'objet

Construction d'un tableau de 4 objets de type `NomClasse`.

Definition

```
NomClasse var[4];
```

Par défaut, le constructeur sans paramètre est appelé pour chaque élément du tableau.

Attention : si `NomClasse` n'a pas de constructeur sans paramètre, cela génère une erreur de compilation :

```
prog.cpp: In function 'int main(int argc, char** argv)':
prog.cpp:8: error: no matching function for call to 'CompteB::CompteB()'
CompteB.h:13: note: candidates are: CompteB::CompteB(int, std::string, float)
CompteB.h:12: note:                 CompteB::CompteB(int, std::string)
CompteB.h:6: note:                CompteB::CompteB(const CompteB&)
```

Déclaration d'un tableau d'objet

Pour faire appel aux constructeurs paramétrés, il faut déclarer et initialiser le tableau en même temps :

Definition

```
NomClasse var[4] = { NomClasse(param), ... };
```

Il faut spécifier entre chaque accolade l'appel au constructeur avec les valeurs des paramètres.

Exemple

```
1  class Point {
2    public:
3      double x;
4      double y;
5
6      Point(double a, double b);
7  };
8
9  int main(int argc, char** argv) {
10   Point T1[3]; // ERREUR
11   Point T2[2] = {Point(1,1), Point(2,2)}; // OK
12   return 0;
13 }
```

Plan du cours

Visibilité des attributs et des méthodes

Il est possible de masquer certain attributs et certaine méthodes au sein d'un objet.

Intérêt : restreindre l'accès aux données de l'objet et ainsi garantir son bon comportement tout au long de son existence.

Exemple

```
CompteB cmpt(12134, "Mr Durand", 0);
cmpt.solde = -1000000;
```

ceci n'est peut être pas un comportement souhaitable pour un programme gérant des comptes bancaires.

Protection des attributs et des méthodes

On utilise les mots clés `public` et `private` pour autoriser respectivement un accès public ou un accès privé :

Les membres d'une classe (attributs ou méthodes) en

- accès public sont accessibles par tout le monde.
- accès privé sont accessible uniquement par les méthodes de l'objet lui-même et les méthodes des objets de la même classe.

Protection des attributs et des méthodes

Definition

```
class MaClasse {
   private:
      int attribut_prive;
      void methodePrivee();
   public:
      int attribut_publique;
      void methodePublique();
};
```

Protection des attributs et des méthodes

Definition

```
class MaClasse {
   private:
      int attribut_prive;
      void methodePrivee();
   public:
      int attribut_publique;
      void methodePublique();
};
```

- `attribut_public` est accessible par une instance de la classe A
- `attribut_prive` n'est pas accessible par une instance de la classe A

```
1 MaClasse a;
2 a.attribut_public = 5; // OK
3 a.attribut_prive = 3; // erreur de compilation
```

Protection des attributs et des méthodes

Definition

```
class MaClasse {
   private:
      int attribut_prive;
      void methodePrivee();
   public:
      int attribut_publique;
      void methodePublique();
};
```

- `methodePublic` est accessible par une instance de la classe A
- `methodePrivee` n'est pas accessible par une instance de la classe A

```
1 MaClasse a;
2 a.methodePublique(); // OK
3 a.methodePrivee(); // erreur de compilation
```

Protection des attributs et des méthodes

Les attributs et les méthodes privées sont accessibles uniquement dans la définition de la classe, c'est-à-dire dans la définition des méthodes de la classe.

Ainsi, on peut écrire ceci

```
1 void MaClasse::methodePublique() {
2   attribut_prive = 10;
3   methodePrivee();
4 }
```

et donc contrôler les modifications faites sur l'objet.

Accesseurs sur les attributs

Il est toujours préférable de déclarer les attributs d'une classe de manière privée.

L'accès aux attributs sera alors effectué au travers de méthodes particulière appelée Accesseur

- Accesseur en lecture : permet de récupérer la valeur de l'attribut
- Accesseur en écriture : permet de modifier la valeur de l'attribut

Pour chaque attribut, on doit faire correspondre les bon accesseurs (si nécessaire dans la conception de l'objet).

Accesseurs sur les attributs

```
1  class Point {
2    private:
3      double x;
4      double y;
5
6    public:
7      Point(double a, double b);
8
9      // Accesseur en lecture
10     double getX() { return x; }
11     double getY() { return y; }
12
13     // Accesseur en ecriture
14     void setX(double abscisse) { x = abscisse; }
15     void setY(double ordonnee) { y = ordonnee; }
16 };
```

Plan du cours

Utilité du mot clé `const`

- Définir des variables ayant une valeur constante.
- Différencier les méthodes ne modifiant pas les objets.
- Forcer un argument à ne pas être pas modifiable dans une fonction.

Variables constantes : les constantes nommées

Le mot clé `const` peut être ajouté à la déclaration d'un objet pour en faire une constante plutôt qu'une variable.

Exemple

```
const int largeurMax = 640;                    // Entier constant
const float Pi = 3.14159265;                   // Réel constant
const int sizeMax[2] = {640, 480};             // Tableau constant
const Point origine(0, 0);    // Objet constant de la classe Point
```

Une constante ne pouvant être modifiée, elle doit donc être initialisée lors de sa définition.

Méthodes constantes d'une classe

Si un objet est déclaré constant, il est impossible de le modifier :
→ on ne peut pas utiliser les méthodes modifiants l'objet ! ! !

Exemple

```
const Point origine(0, 0);
origine.setX(10);      erreur car essaie de modifier un objet constant
```

Les méthodes ne modifiant pas l'objet doivent être déclarées en ajoutant le mot clé `const` après leur déclaration.

Méthodes constantes d'une classe

Definition

```
class MaClasse {
public:
   void methodeConstante() const;
};
```

Une méthode constante peut être appelée par des objets constants ou non-constants.

L'inverse est faux : une méthode non-constante peut être appelée uniquement par des objets non-constants.

Exemple de méthode constante

```
1  class Point {
2    private:
3      double x;
4      double y;
5
6    public:
7      Point (double a, double b);
8
9      // Accesseur en lecture (methodes constantes)
10     double getX() const { return x; }
11     double getY() const { return y; }
12
13     // Accesseur en ecriture
14     void setX(double abscisse) { x = abscisse;}
15     void setY(double ordonnee) { y = ordonnee;}
16
17 };
```

Arguments constants dans une fonction

On peut forcer un argument d'une fonction (ou méthode) à être constant, en précédant sont type par `const` dans la déclaration.

Definition

```
type_retour maFonction(const type1 arg1, type2 arg2, ...);
```

Lors de l'appel d'une fonction (ou méthode), un argument non-constant peut-être passé à la place d'un argument constant :

l'objet passé devient alors constant dans la fonction.

Attention, l'inverse est faux.

Arguments constants dans une fonction

```
1  void affiche(const Point &p){
2    cout << "abscisse : " << p.getX() << endl;
3    cout << "ordonnee : " << p.getY() << endl;
4  }
5  void saisie(Point &p, double x, double y){
6    p.setX(x); p.setY(y);
7  }
8
9  int main(int argc, char** argv) {
10   const Point origine(0, 0);
11   Point P(1, 10);
12
13   affiche(origine); // OK
14   affiche(P); // OK
15   saisie(P, 4, 5); // OK
16   saisie(origine, 1, 1); // ERREUR
17
18   return 0;
19 }
```

Plan du cours

Constructeur par copie

Le constructeur par copie est un constructeur qui construit un objet à l'identique à partir d'un autre objet de la même classe.

Ce constructeur prend en paramètre une référence constante sur un objet de la même classe :

Exemple

```
class MaClasse {
public:
   MaClasse(const MaClasse &a);
};
```

Constructeur par copie

Il est possible d'appeler explicitement le constructeur par copie :

```
1 MaClasse a1;      // constructeur par defaut
2 MaClasse a2(a1); // constructeur par copie
```

Le constructeur par copie est appelé implicitement lors de passage par valeur d'un objet :

```
1 void maFonction(MaClasse x) {...}
2
3 MaClasse a1;
4 maFonction(a1); // appel implicite au constr. par copie
```

Constructeur par copie

- Si aucun constructeur par copie est spécifié dans une classe, le compilateur en génère un par défaut.

  Attention, la copie se fait bit à bit sur les attributs (dangereux avec les attributs dynamiques).

- La déclaration d'un objet couplée avec son affectation est remplacée automatiquement par l'appel au constructeur par copie :

Exemple

```
MaClasse a1;
MaClasse a2 = a1; // remplacé par MaClasse a2(a1);
```

Constructeur par copie

```
1  class Point {
2    private:
3      double x;
4      double y;
5
6    public:
7      Point (double a, double b);
8      Point (const Point &p);
9      ...
10 };
```

```
1 Point::Point(const Point &p){
2   this->x = p.x;
3   this->y = p.y;
4 }
```

ou mieux (et moins risqué)

```
1 Point::Point(const Point &p)
2 : x(p.x), y(p.y) { }
```

Opérateur d'affectation (=)

Initialisation *vs* affectation :

- L'initialisation consiste à donner une valeur initiale à un objet lors de sa construction.
- L'affectation (via l'opérateur =) consiste à changer la valeur d'un objet déjà construit.

Exemple

```
Point p1(0.5, 0.5); // Construction d'un objet p1
Point p2 = p1;      // Construction d'un objet p2 par copie
p1 = p2;            // Affectation
```

Surcharge de l'opérateur d'affectation

L'opérateur d'affectation = est une méthode permettant de copier un objet dans un autre objet déjà construit.

Cette méthode se nomme `operator=`, elle prend en paramètre une référence constante sur un objet de la même classe, et elle renvoie une référence sur l'objet courant (`*this`).

Exemple

```
class MaClasse {
public:
   ...
   MaClasse& operator=(const MaClasse &M);
}
```

Surcharge de l'opérateur d'affectation

- Si l'opérateur d'affectation n'est spécifié dans une classe, le compilateur en génère un par défaut.

  Attention, la copie se fait bit à bit sur les attributs (dangereux avec les attributs dynamiques).

- L'opérateur d'affectation est une méthode particulière : pas besoin d'appel pointé par l'objet

Exemple

```
MaClasse a, b;
a = b;  // appel à la méthode operator= de MaClasse
        // a est l'objet appelant et b le paramètre.
```

Opérateur d'affectation : Exemple

```
1  class Point {
2    private:
3      double x;
4      double y;
5
6    public:
7      Point(double a, double b);
8      Point(const Point &p);
9      Point &operator=(const Point &p);
10 };
```

```
1  Point &Point::operator=(const Point &p) {
2    if (this != &p) {
3      this->x = p.x;
4      this->y = p.y;
5    }
6    return *this;
7  }
```

Plan du cours

Bilan sur les classes en C++

Les classes permettent de

- regrouper des données et des fonctionnalités au sein d'un objet
- protéger l'accès des objets (`public`, `private`)
- encapsuler d'autres objets (objets membres)

Les notions importantes des classes sont :

- les attributs : les données d'un objet
- les méthodes : les fonctionnalités d'un objet
- constructeur : paramétrés, par défaut, par copie.
- destructeur : un et un seul par classe

Plan du cours

Plan du cours

Entrées-Sorties dans un programme

Definition

Les entrées-sorties sont le moyen d'interagir/communiquer avec un programme.

Les interactions peuvent se faire

- via des fichiers
- via des "buffers" (tampon en français)
- via des flux système standard (l'écran, le clavier, erreur)

On connaît déjà

```
1 int x;
2 std::cin>>x;                  // flux provenant du clavier
3 std::cout<<x<<std::endl; // flux allant vers l'ecran
```

Les flux systèmes standards

Ils sont ouverts automatiquement par le système au début de l'exécution du programme.

Flux ouvert en écriture seule :

- `cin` : lié à l'entrée standard, par défaut le clavier

Flux ouvert en lecture seule :

- `cout` : lié à la sortie standard, par défaut l'écran
- `cerr` : lié à la sortie d'erreur standard (non bufferisé)
- `clog` : lié à la sortie d'erreur standard

Il faut inclure la bibliothèque C++ standard d'entrée-sortie :

```
#include <iostream>
```

Les types des flux standards

- Les objets `cout`,`cerr` et `clog` sont tous des instances de la classe `ostream` définissant les flux d'écriture seule.

- L'objet `cin` est une instance de la classe `istream` définissant les flux de lecture seule.

Attention

Les objets des classes `istream` et `ostream` ne peuvent pas être copiés.

$\hookrightarrow$ passage de paramètre toujours par référence

Les flux sur des fichiers

Au lieux d'interagir avec un programme de manière interactive (écran/clavier), il est souvent souhaitable de passer par des fichiers.

Avantages :

- enregistrement pérenne des données
- envoi de données en masse
- stockage sous différents format (binaire par exemple)

Definition

Un flux fichier est un flux qui a été lié à une ressource physique du système de fichier (un seul flux par fichier).

Les flux sur des fichiers

Trois classes de flux fichier en C++ :

- `ofstream` : flux de fichier en écriture seule
- `ifstream` : flux de fichier en lecture seule
- `fstream` : flux de fichier en lecture et écriture

Il faut inclure la bibliothèque C++ d'entrée-sortie sur fichier :

```
#include <fstream>
```

Utilisation d'un flux sur un fichier

Règle d'or :

1. On déclare une variable du type de flux désiré.
2. On relie cette variable à un fichier avec la fonction
3. On utilise le flux pour envoyer/recevoir des données
4. On détache le flux du fichier

Utilisation d'un flux sur un fichier

Règle d'or :

1. On déclare une variable du type de flux désiré.

```
1 #include <ftream>
2 using namespace std;
3 int main {
4    ofstream myout;
5    ifstream myin;
6    fstream  myinout;
7    return 0;
8 }
```

Relier un flux à un fichier

Pour lier un flux à un fichier, on appelle la méthode

`open ("nom_fichier")`

à partir d'un objet flux libre.

```
1  ofstream myout;
2  myout.open("fichier1.txt");
```

Attention

Le flux `myout` et relier au fichier `"fichier1.txt` :

- si le fichier n'existe pas, il est créé sinon il est écrasé
- les données envoyées dans le flux seront écrites de manière formattée (caractère ASCII)

Relier un flux à un fichier

Options de la fonction open :

- `ios::app` : écrit dans un fichier sans l'écraser (fin du fichier)
- `ios::binary` : flux de données au format binaire

Ces options doivent être précédées des modes d'accés du flux :

`ios::in`, `ios::out`

```
1  ofstream myout1,myout2;
2  ifstream myin;
3  fstream myinout;
4
5  myout1.open("fichier1.txt");
6  myout2.open("fichier2.txt",ios::out|ios::app);
7  myin.open("fichier3.bin",ios::in|ios::binary);
8  myinout.open("fichier4.txt",ios::in|ios::out|ios::app);
```

Relier un flux à un fichier

On peut lors de la construction d'un flux le relier directement à un fichier sans passer par la fonction `open`.
↪ construction paramétrée avec les mêmes paramètre qu' `open`

```
1 ofstream   myout2("fichier2.txt",ios::out|ios::app);
```

est équivalent à

```
1 ofstream myout2;
2 myout2.open("fichier2.txt",ios::out|ios::app);
```

Détacher un fichier d'un flux

Pour détacher un fichier d'un flux, on utilise la méthode `close()`

```
1 ofstream myout2;
2 myout2.open("fichier2.txt",ios::out|ios::app);
3 ...
4 myout2.close();
```

Attention

Il faut toujours détacher un fichier de son flux avant qu'il soit détruit.
↪ risque de corruption de fichier.

Utiliser un flux sur un fichier

Comme pour `cin` et `cout`, on utilise les opérateurs d'extraction `>>` et d'insertion `<<`.

- L'extraction `>>` se fait sur un flux en lecture
- L'insertion `<<` se fait sur un flux en écriture

```
1 ifstream myin("fichier1.txt");
2 ofstream myout("fichier2.txt");
3 int x;
4 myin>>x;    lecture d'un entier sur fichier1.txt
5 myout<<x;   ecriture de x sur fichier2.txt
6 myin.close();
7 myout.close();
```

Vérification de l'état d'un flux

Plusieurs méthodes sont disponibles pour vérifier l'état d'un flux :

- `bad()` : vérifie si l'insertion/extraction du flux à échoué.
- `fail()` : comme `bad()` mais vérifie les formats des données
- `eof()` : vérifie si un flux de lecture est en fin de fichier
- `good()` : renvoi `true` si tout est ok
- `is_open()` : vérifie si le flux est lié à un fichier

```
1  ifstream  myin("fichier1.txt");
2  ofstream myout("fichier2.txt");
3  if ( !myin.is_open() || !myout.is_open())
4      cout<<"Erreur d'ouverture de fichier"<<endl;;
5  else {
6    int x;
7    myin>>x;
8    if (myin.fail())
9       cout<<"Erreur de lecture"<<endl;;
10   myout<<x<<endl;
11 }
```

Utilisation avancée des flux

L'utilisation des opérateurs >> et << est assez restrictive :

```
1 int x,y,x;
2 cin>>x>>y>>z;
```

- les extractions sont enchainées par des séparateurs,
- les données sont formatées (vérification des saisies).

Il est parfois utile de faire des interactions non formatées ou de gérer les séparateurs : ex. une phrase dans un string.

Extraction avancée dans les flux

Extraction d'un flux dans une chaine de caractère :

- `getline(char* s, std::streamsize n, char delim);`
  ↪ récupère au plus `n char` dans le flux jusqu'a `delim`.

```
1 char Nom[256];
2 cin.getline(Nom,256,'\n');
3 cin.getline(Nom,256);
```

- `getline(istream& is, string& str, char delim);`
  ↪ récupère les caractères du flux jusqu'a `delim` (ou EOF).

```
1 string Nom;
2 std::getline(std::cin,Nom,'\n');
3 std::getline(std::cin,Nom);
```

Rq : `delim` est extrait du flux mais pas stocké dans la chaîne.
Rq : `delim` n'est pas obligatoire défaut à `\n`

Extraction avancée dans les flux

- `read(char* s, std::streamsize n)`
  ↪ récupère au plus `n char` du flux (s'arrête si EOF).

```
1 char buffer[256];
2 cin.read(buffer,256);
```

- `ignore(std::streamsize n, char delim)`
  ↪ enlève au plus `n char` du flux jusqu'a `delim` (ou EOF).

```
1 string phrase2;
2 cin.ignore(1024,'.');
3 cin.getline(phrase2,'.');
```

- `peek()` ↪ retourne le caractère suivant sans l'enlever.

```
1 char c=cin.peek();
```

Insertion avancée dans les flux

- `write(char* s, std::streamsize n)`
  ↪ insère `n char` dans le flux.

```
1 char buffer[256];
2 cin.read(buffer,256);
3 cout.write(buffer,256);
```

Positionnement dans les flux

- `tellp() / tellg()`
  ↪ donne la position en octet dans le flux insertion/extraction.
- `seekp(std::streampos pos)`
  ↪ positionne l'insertion dans le flux à pos.
- `seekg(std::streampos pos)`
  ↪ positionne l'extraction dans le flux à pos.
- `seekp(std::streamoff off, std::seekdir way)`
  ↪ positionne l'insertion dans le flux à way+off.
- `seekg(std::streamoff off, std::seekdir way)`
  ↪ positionne l'extraction dans le flux à way+off.

Remarque

way doit être : `flow.beg` ou `flow.end` avec `flow` l'object flux dans lequel on veut faire le positionnement.

Exemple avec les flux de fichier

```
1  #include <iostream>      // std::cout
2  #include <fstream>       // std::ifstream
3
4  int main () {
5    std::ifstream is ("test.txt", std::ifstream::binary);
6    if (is.is_open()) {
7      is.seekg (0, is.end);
8      int length = is.tellg();
9      char * buffer = new char [length];
10
11     is.seekg (0, is.beg);
12     is.read (buffer,length);
13     is.close();
14
15     std::cout.write (buffer,length);
16     delete[] buffer;
17   }
18
19   return 0;
20 }
```

Plan du cours

Plan du cours

Plan du cours

Les pointeurs

Definition

Un pointeur est une zone mémoire qui contient une adresse mémoire.

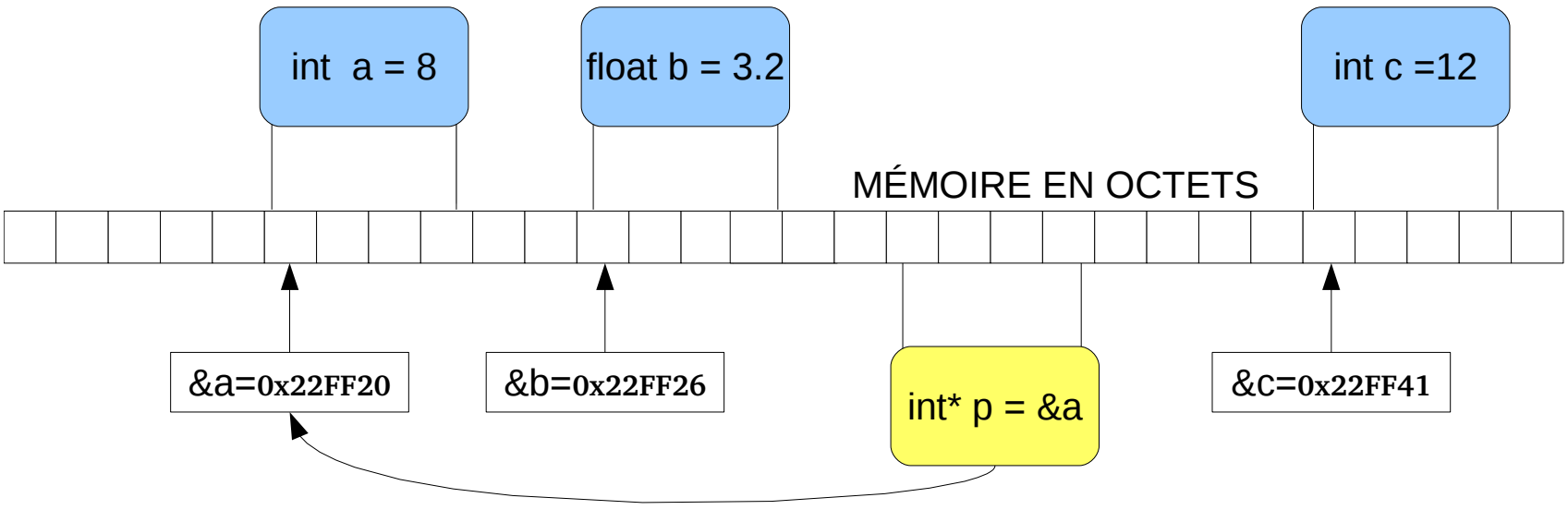
Une variable peut être définie comme un pointeur en précédant son nom par *.

```
1 int *p;
```

Attention

Comme pour les variables normales, les pointeurs sont typés.

Les pointeurs



- `a,b,c` sont des variables normales (elles stockent des valeurs).
- `p` est une variable *dite* pointeurs (elle stocke une adresse).
- `p` est de type `int*` et doit stocker l'adresse d'un entier (ici `a`).

Les pointeurs

On peut affecter une variable pointeur avec :

Exemple

```
int a, *p, *q
```

Les pointeurs

On peut affecter une variable pointeur avec :

- l'adresse d'une variable existante *compatible*,

Exemple

```
int a, *p, *q
p = &a;
```

Les pointeurs

On peut affecter une variable pointeur avec :

- l'adresse d'une variable existante *compatible*,
- avec la valeur `NULL` (pointeur vide),

Exemple

```
int a, *p, *q
p = &a;
q = NULL;
```

Les pointeurs

On peut affecter une variable pointeur avec :

- l'adresse d'une variable existante *compatible*,
- avec la valeur `NULL` (pointeur vide),
- avec la valeur d'une autre pointeur *compatible*,

Exemple

```
int a, *p, *q
p = &a;
q = NULL;
q = p;
```

Les pointeurs

On peut affecter une variable pointeur avec :

- l'adresse d'une variable existante *compatible*,
- avec la valeur `NULL` (pointeur vide),
- avec la valeur d'une autre pointeur *compatible*,
- avec l'adresse d'une zone mémoire créée par l'utilisateur.

Exemple

```
int a, *p, *q
p = &a;
q = NULL;
q = p;
q = (int*) malloc(sizeof(int)); en C
```

Manipulation avec les pointeurs

Lorsqu'on manipule des variables pointeurs, on peut :

Exemple

```
int a, *p;
a = 10; p = &a;
```

Manipulation avec les pointeurs

Lorsqu'on manipule des variables pointeurs, on peut :

- récupérer la valeur stockée dans la zone mémoire pointée

Exemple

```
int a, *p;
a = 10; p = &a;
a = *p + 1;
```

Manipulation avec les pointeurs

Lorsqu'on manipule des variables pointeurs, on peut :

- récupérer la valeur stockée dans la zone mémoire pointée
- affecter la valeur stockée dans la zone mémoire pointée

Exemple

```
int a, *p;
a = 10; p = &a;
a = *p + 1;
*p = 13;
```

Manipulation avec les pointeurs

Lorsqu'on manipule des variables pointeurs, on peut :

- récupérer la valeur stockée dans la zone mémoire pointée
- affecter la valeur stockée dans la zone mémoire pointée
- libérer l'espace mémoire attribué à la zone mémoire pointée [1]

Exemple

```
int a, *p;
a = 10; p = &a;
a = *p + 1;
*p = 13;
```

1. si la mémoire a été allouée dynamiquement.

Plan du cours

Intérêt de l'allocation dynamique

Il est souvent nécessaire d'allouer de la mémoire en fonction de données qui ne sont pas connues à la compilation.
(*e.g.*, un tableau avec une taille saisie par l'utilisateur).

Attention

Une variable allouée dynamiquement doit toujours être « rattachée » à un pointeur, car elle ne possède aucun identificateur (hormis le déréférencement du dit pointeur).

Rappel sur les pointeurs

Definition

`nomType *ptr;`

déclare une variable de nom `ptr` qui est un pointeur sur une donnée de type `nomType`.

- cette déclaration réserve une zone mémoire vide de la taille d'un entier
- `nomType` peut être n'importe quel type C++ (classe comprise)
- le pointeur vide est soit `0` soit `NULL` (en incluant `iostream`)
- on ne peut pas saisir directement la valeur d'un pointeur

Opérateur d'allocation dynamique : `new`

En C++, l'allocation dynamique se fait avec le mot clé `new`

Definition

`new nomType;`
créé une variable dynamique du type `nomType`

`nomType` peut être :

- n'importe quel type scalaire C++ (classe comprise)
- un type multi-dimensionnel (tableau)
- un pointeur

Opérateur de libération mémoire : `delete`

En C++, la libération de mémoire dynamique se fait avec le mot clé `delete`

Definition

`delete ptr;`
libère la zone mémoire située à l'adresse mémoire stockée dans `ptr`

Attention

- l'adresse mémoire stockée dans `ptr` doit avoir été allouée dynamiquement
- `ptr` ne doit pas correspondre à un tableau dynamique

Variable dynamique scalaire

Definition

`nomTypeSimple *ptr = new nomTypeSimple;`

- crée une zone mémoire de taille `sizeof(nomTypeSimple)` dans la mémoire dynamique (appelée le tas),
- affecte `ptr` avec l'adresse de cette zone mémoire.

Exemple :

```
1 int    *a = new int;    \\ entier dynamique   (4 octets)
2 double *b = new double;\\ flottant dynamique (8 octets)
```

- on libère la mémoire ainsi créée par : `delete ptr;`

Variable dynamique scalaire : un exemple

```
1  #include <iostream>
2  using namespace std;
3
4  int main(int argc, char** argv) {
5    int *ptr1, *ptr2, a;
6    a = 10; // memoire non-dynamique
7    ptr1 = &a; // memoire non-dynamique
8    ptr2 = new int(11); // memoire dynamique
9
10   cout << "ptr1 : " << ptr1 << " - " << *ptr1 << endl;
11   cout << "ptr2 : " << ptr2 << " - " << *ptr2 << endl;
12
13   delete ptr2;
14   // delete ptr1; ERREUR (non dynamique)
15   return 0;
16 }
```

Variable dynamique de type tableau

Definition

`nomTypeSimple *ptr = new nomTypesimple[nb];`
où `nb` est un entier positif non nul.

Cet appel :

- crée `nb` cases contiguës de type `nomTypeSimple` dans le tas
- affecte `ptr` avec l'adresse de la 1ère case
- chacune des cases est accessible par l'opérateur `[]` :
  `ptr[0]`, `ptr[1]`, $\cdots$, `ptr[nb-1]`

Remarque :
On peut faire de l'arithmétique sur les pointeurs, ainsi `ptr+1` correspond à un pointeur initialisé avec l'adresse de la case `ptr[1]`.

Variable dynamique de type tableau

Attention

Pour libérer la mémoire dynamique allouée sous forme de tableau, il faut libérer toute la zone mémoire d'un coup (`nb` cases).

Pour cela, on utilise la commande : `delete[]`

Exemple :

```
1 int *a = new int[4]; // alloue 16 octets contigus
2 delete[] a; // libere les 16 octets
3 // delete a; // ERREUR, ne libererait que 4 octets
```

Variable dynamique de type tableau : un exemple

```
1  #include <iostream>
2  using namespace std;
3
4  int main(int argc, char** argv) {
5    int *ptr;
6    ptr = new int[3];
7    ptr[0] = 2;
8    ptr[1] = 4;
9    ptr[2] = 8;
10
11   cout << "ptr[0] : " << ptr << " - " << ptr[0] << endl
12     << "ptr[1] : " << ptr+1 << " - " << ptr[1] << endl
13     << "ptr[2] : " << ptr+2 << " - " << ptr[2] << endl;
14
15   delete[] ptr;
16   return 0;
17 }
```

Instance dynamique d'objet

Definition

```
MaClasse *ptr = new MaClasse;
```

- crée un objet dynamique de type `MaClasse` dans le tas en appelant le constructeur sans paramètre de `MaClasse`
- affecte `ptr` avec l'adresse mémoire de l'objet dynamique

On accède à l'objet en utilisant le déréférencement du pointeur `ptr` : `*ptr`

Remarque : On peut remplacer le constructeur sans paramètre par n'importe quel constructeur de la classe (paramétrés, copie).

Instance dynamique d'objet

L'accès aux attributs et aux méthodes des objets dynamiques se fait par :

- déréférencement du pointeur et appel classique : `(*ptr).nomAttribut;` et `(*ptr).nomMethode(...);`
- ou en utilisant l'opérateur `->` : `ptr->nomAttribut;` et `ptr->nomMethode(...);`

La libération mémoire de l'objet se fait avec `delete ptr;` qui appellera automatiquement le destructeur de la classe.

Instance dynamique d'objet : exemple

Point.h

```
1  class Point {
2    private:
3      double x;
4      double y;
5
6    public:
7      Point (double a, double b): x(a), y(b) { }
8      int getX() const {return x;}
9      int getY() const {return y;}
10     void setX(int abscisse) { x = abscisse; }
11     void setY(int ordonnee) { y = ordonnee; }
12 };
```

Instance dynamique d'objet : exemple

```
1  #include "Point.h"
2  #include <iostream>
3  using namespace std;
4
5  int main(int argc, char** argv) {
6    Point *ptr = new Point(0,0);
7
8    cout << ptr->getX() << ", " << ptr->getY() << endl;
9
10   delete ptr;
11
12   return 0;
13 }
```

Plan du cours

Les objets et les attributs pointeurs

Sources majeures d'erreur :

- les attributs pointeurs non initialisés
- les attributs pointeurs partageant une même zone mémoire
- les attributs pointeurs initialisés sur une variable locale

Initialisation dans les constructeurs

Attention

Un attribut de type pointeur doit être initialisé avec une adresse mémoire valide dans chaque constructeur

Cette initialisation peut se faire par :

- appel à l'allocation dynamique avec `new`
- affectation du pointeur vide `NULL`

Rq : l'allocation et/ou l'affectation de valeur à la zone mémoire pointée peut se faire en dehors des constructeurs, l'important est d'initialiser le pointeur.

Initialisation dans les constructeurs

```
1  #include "point.h"
2  class Polygone {
3    private:
4      int nbrCotes;
5      Point *tabPts;
6
7    public:
8      Polygone(): nbrCotes(0), tabPts(NULL) { }
9      Polygone(int n)
10     : nbrCotes(n), tabPts(new Point[n]) { }
11 };
```

Modification de la zone mémoire pointée

Grâce à l'initialisation du pointeur, on sait si il est nécessaire ou pas d'allouer de la mémoire :

```
1  class Polygone {
2    ...
3
4    public:
5      void saisie() {
6        if (tabPts == NULL){
7          cin >> nbrCotes;
8          tabPts = new Point[nbrCotes];
9        }
10       for (int i = 0 ; i < nbrCotes ; i++) {
11         double x,y;
12         cin >> x >> y;
13         tabPts[i].setX(x);
14         tabPts[i].setY(y);
15       }
16     }
17 };
```

Exemple :

```
1 int main(int argc, char** argv) {
2   Polygone p1; // pas d'allocation dynamique
3   Polygone p2(3); // allocation dynamique
4
5   p1.saisie(); // allocation dynamique
6   p2.saisie(); // pas d'allocation dynamique
7
8   return 0;
9 }
```

Libération des pointeurs

Si un pointeur a été initialisé avec l'adresse d'une zone mémoire allouée dynamiquement, il faut « absolument » libérer cette mémoire avant la destruction du pointeur :

rôle du destructeur dans les classes

Attention

Dans l'exemple précédent, les zones mémoires allouées dynamiquement pour `p1` et `p2` ne sont pas libérées...

Libération des pointeurs

```
1 int main(int arc, char** argv) {
2   Polygone p1; // pas d'allocation dynamique
3   Polygone p2(3); // allocation dynamique
4   p1.saisie(); // allocation dynamique
5   p2.saisie(); // pas d'allocation dynamique
6   return 0;
7 }
```

À la fin du `main()`, le destructeur de `p1` et `p2` est appelé :
Ce destructeur

- exécute son code : ici rien car vide
- libère la mémoire des attributs de l'objet : nbrCote et tabPts
- n'a pas libéré la zone mémoire pointée par tabPts ! ! !

Libération mémoire dans le destructeur

Definition

Si une classe possède un attribut pointeur alloué dynamiquement, le destructeur de cette classe doit libérer la zone mémoire pointée par ce pointeur (appel au `delete` correspondant).

Exemple

```
class MaClasse {
   private:
      int *ptr;
   public:
      MaClasse(int a) { ptr = new int(a); }
      ~MaClasse() { delete ptr; }
};
```

Libération mémoire dans le destructeur : exemple

```
1  #include "point.h"
2  class Polygone {
3    private:
4      int nbrCotes;
5      Point *tabPts;
6    public:
7      ...
8      ~Polygone() {
9        if (tabPts != NULL)
10         delete[] tabPts;
11     }
12 };
```

Attention

ici `tabPts` est un tableau dynamique donc libération avec `delete[]`.

Problème avec la copie d'objet

Pour copier un objet, on peut :

- utiliser le constructeur par copie
- utiliser l'opérateur d'affectation =

Si la classe ne définit pas la méthode, le compilateur la fournit automatiquement en se basant sur la copie d'attribut bit à bit [2] :

copie de pointeurs → copie d'adresse

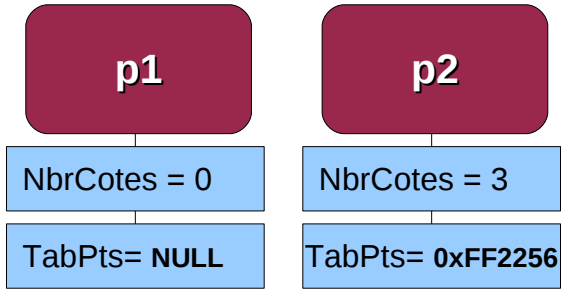
2. si l'attribut est un objet, le compilateur fait appel au constructeur par copie ou à l'opérateur de copie

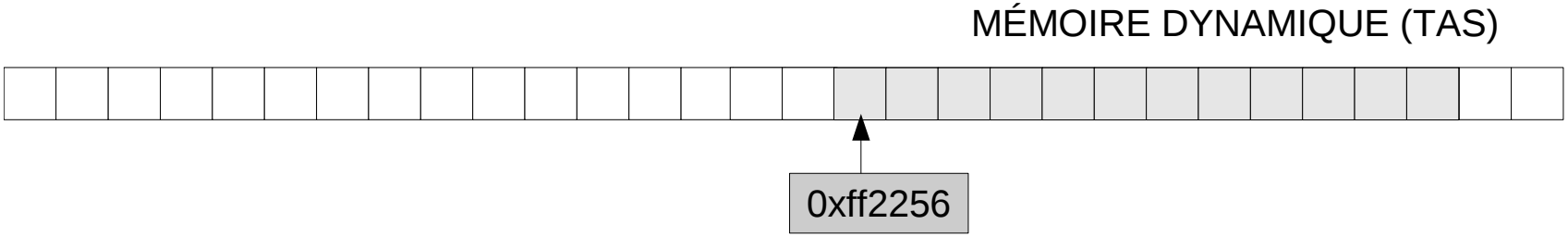
Problème avec la copie d'objet

```
1  int main(int argc, char** argv) {
2     Polygone p1;
3     Polygone p2(3);
4     p1.saisie();
5     p2.saisie();
6
7     p1 = p2;
8     Polygone p3(p1);
9     return 0;
10 }
```

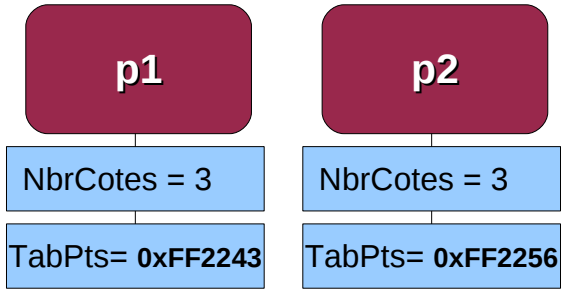
Attention

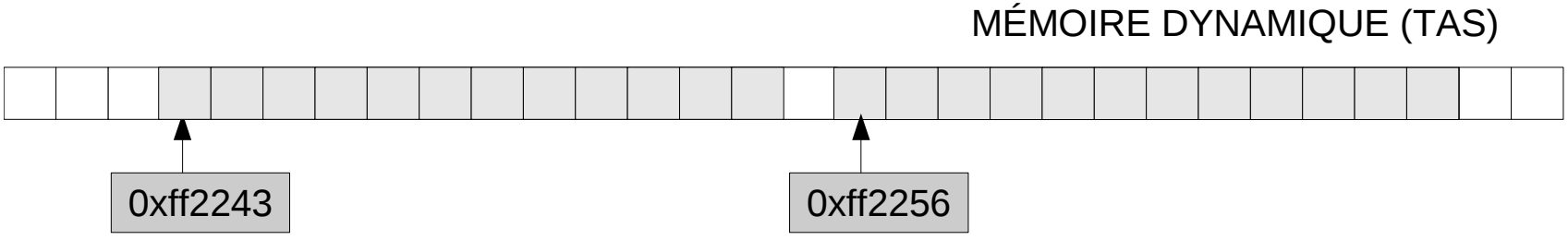
dans cet exemple, `p1.tabPts = p2.tabPts = p3.tabPts`, cela signifie qu'ils partagent tous la même zone mémoire $\rightarrow$ Pb à la destruction



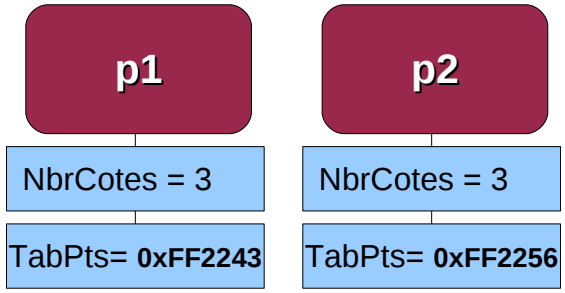


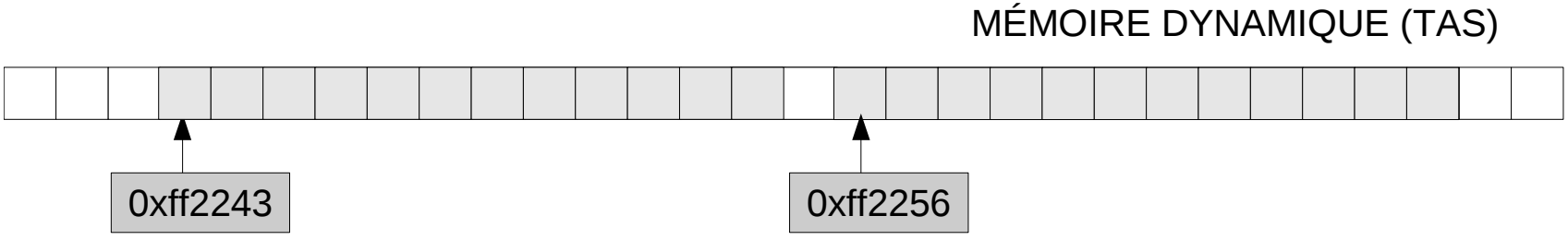

```
1 Polygone p1;
2 Polygone p2(3);
3 p1.saisie();
4 p2.saisie();
5 p1 = p2;
6 Polygone p3(p1);
```



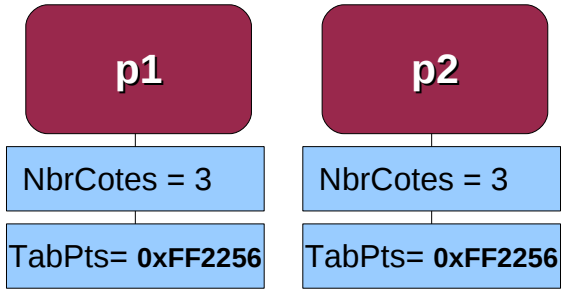


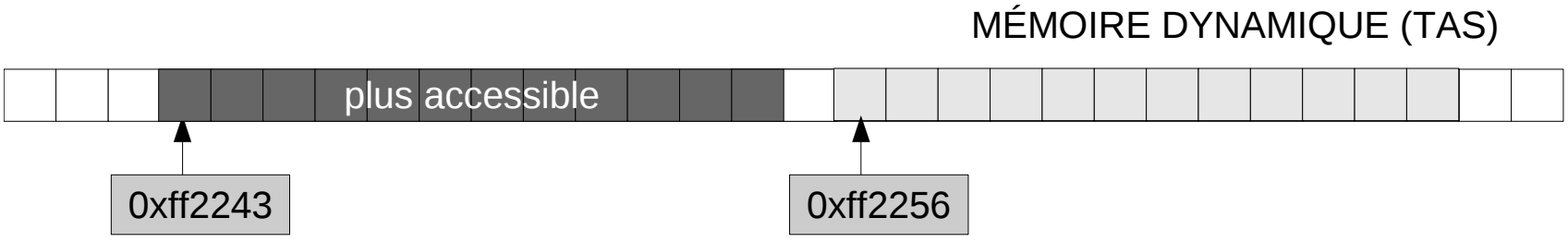

```
1 Polygone p1;
2 Polygone p2(3);
3 p1.saisie();
4 p2.saisie();
5 p1 = p2;
6 Polygone p3(p1);
```



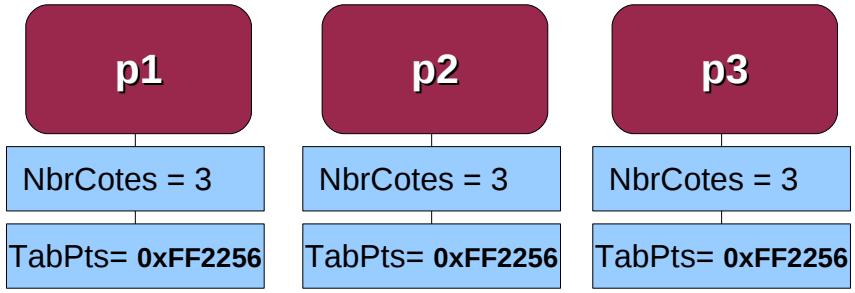


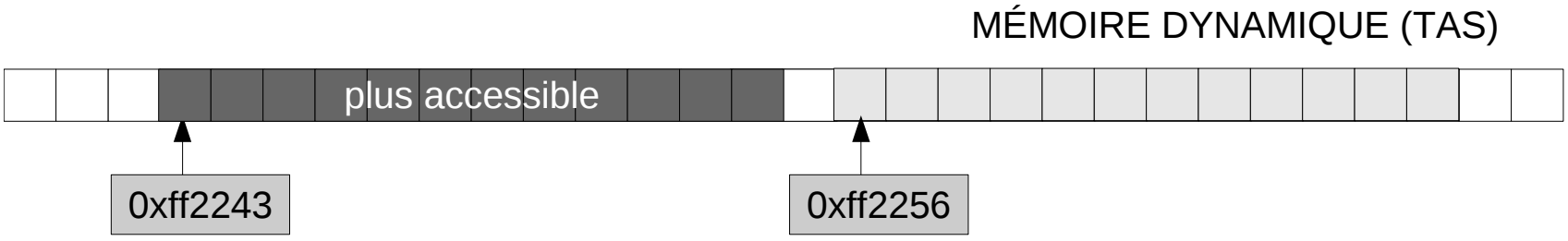

```
1 Polygone p1;
2 Polygone p2(3);
3 p1.saisie();
4 p2.saisie();
5 p1 = p2;
6 Polygone p3(p1);
```






```
1 Polygone p1;
2 Polygone p2(3);
3 p1.saisie();
4 p2.saisie();
5 p1 = p2;
6 Polygone p3(p1);
```






```
1 Polygone p1;
2 Polygone p2(3);
3 p1.saisie();
4 p2.saisie();
5 p1 = p2;
6 Polygone p3(p1);
```

Copie en profondeur des objets

Solution : il faut effectuer une allocation dynamique et copier les données explicitement dans cette zone mémoire.

Constructeur par copie

1. construction par copie des attributs non pointeurs
2. construction par défaut des attributs pointeurs
3. allocation dynamique des attributs pointeurs
4. copie des zones mémoires pointées

Exemple

```
class MaClasse {
   private:
      int nonptr;
      int *ptr;
   public:
      MaClasse(const MaClasse &o): nonptr(o.nonptr) {
         ptr = new int;
         *ptr = *(o.ptr);
      }
};
```

Constructeur par copie : Exemple

```
1  #include "point.h"
2  class Polygone {
3    private:
4      int nbrCotes;
5      Point *tabPts;
6
7    public:
8      ...
9      Polygone(const Polygone &p)
10     : nbrCotes(p.nbrCotes),
11       tabPts(new Point[p.nbrCotes]) {
12       for (size_t i = 0 ; i < nbrCotes ; i++)
13         tabPts[i] = p.tabPts[i];
14     }
15 };
```

Opérateur d'affectation (copie)

Schéma différent du constructeur par copie car l'objet est déjà créé : il faut peut être libérer de la mémoire

1. copie des attributs non pointeur
2. libération des zones mémoires pointées si pointeur $\neq$ `NULL`
3. allocation dynamique des attributs pointeurs
4. copie des zones mémoires pointées

Attention

l'étape 2 est dangereuse si un objet fait une copie de lui même (p=p)

Opérateur d'affectation (copie)

Exemple

```
class MaClasse {
   private:
      int nonptr;
      int *ptr;
   public:
      MaClasse& operator=(const MaClasse &o) {
         if (&o != this){
            nonptr = o.nonptr;
            delete ptr;
            ptr = new int;
            *ptr = *(o.ptr);
         }
         return *this;
      }
};
```

Opérateur de copie : Exemple

```
1  #include "point.h"
2  class Polygone {
3    private:
4      int nbrCotes;
5      Point *tabPts;
6
7    public:
8      ...
9      Polygone& operator=(const Polygone &p) {
10       if (&p != this) {
11         nbrCotes = p.nbrCotes;
12         delete[] tabPts;
13         tabPts = new Point[nbrCotes];
14         for (size_t i = 0 ; i < nbrCotes ; i++)
15           tabPts[i]=p.tabPts[i];
16       }
17       return *this;
18     }
19 };
```

Remarques importantes

Le constructeur par copie, la méthode `operator=` et le destructeur forment un groupe inséparable pour des attributs pointeurs.

- le constructeur par copie et l'opérateur d'affectation (`operator=`) vont de paire car ils gèrent le même problème : copie en profondeur
- si on écrit uniquement le destructeur, il y aura des erreurs de segmentation : double libération de mémoire
- si on écrit tout sauf le destructeur, on va saturer le tas avec des données inaccessibles : fuite mémoire

Le problème de la copie en profondeur intervient également dans les constructeurs ayant un pointeur comme paramètre.

Copie superficielle des objets

Il est possible que des objets utilisent des pointeurs pour faire référence à des données allouées par d'autres objets.

Dans ce cas, il ne faut pas faire de copie en profondeur et le destructeur ne doit rien libérer. On parle de copie superficielle.

On verra ce type de classe quand on parlera des listes.

Plan du cours

Passage de paramètre du type pointeur

Comme pour les types classiques, il y a deux manières de passer un pointeur en paramètres :

- passage par valeur
- passage par adresse

Passage de pointeurs par adresse

Definition

```
type_retour mafonction(type_param* &ptr) {...}
```

Dans la fonction, c'est le pointeur lui même qui est manipulé et non pas une copie.

À l'intérieur de la fonction, on peut :

- accèder et modifier la zone mémoire pointée : `*ptr = ...`
- modifier la valeur du pointeur : `ptr = ...`

Passage de pointeurs par adresse

```
1  void echPtr(int* &p1, int* &p2) {
2    int *p;
3    p = p1;
4    p1 = p2;
5    p2 = p;
6  }
7  int main(int argc, char** argv) {
8    int *a, *b, c;
9    a = new int;
10   b = &c;
11   echPtr(a, b); // OK
12   echPtr(a, &c); // ERREUR
13   return 0;
14 }
```

ligne 12 : les valeurs de a et b sont bien échangées.

ligne 13 : erreur car &c n'est pas une variable et n'a pas d'adresse.

Passage de pointeurs par adresse

```
1
2  #include <iostream>
3  using namespace std;
4
5  void copyTab(int* &T1, int* &T2, size_t n) {
6    T1 = new int[n];
7    for(size_t i = 0 ; i < n ; i++)
8      T1[i] = T2[i];
9  }
10
11 int main(int argc, char** argv) {
12   int *a = new int[2]; a[0] = 3; a[1] = 4;
13   int *b;
14   copyTab(b, a, 2);
15
16   cout << b[0] << ", " << b[1] << endl;
17   return 0;
18 }
```

Passage de pointeurs par valeur

Definition

```
type_retour mafonction(type_param * ptr) { ... }
```

Dans la fonction, c'est une copie du pointeur qui est manipulée et non pas le pointeur.

À l'intérieur de la fonction, on peut uniquement :

- accèder et modifier la zone mémoire pointée : `*ptr = ...`

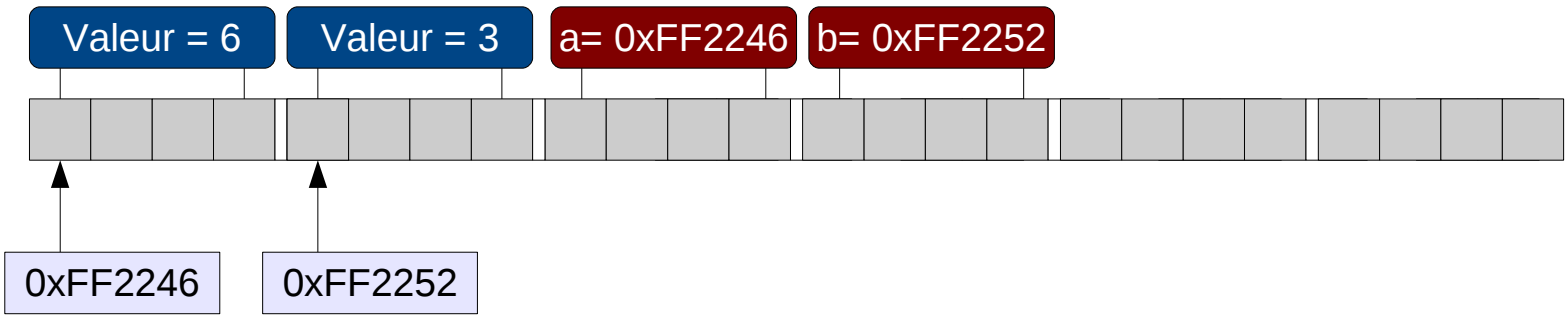
Passage de pointeurs par valeur

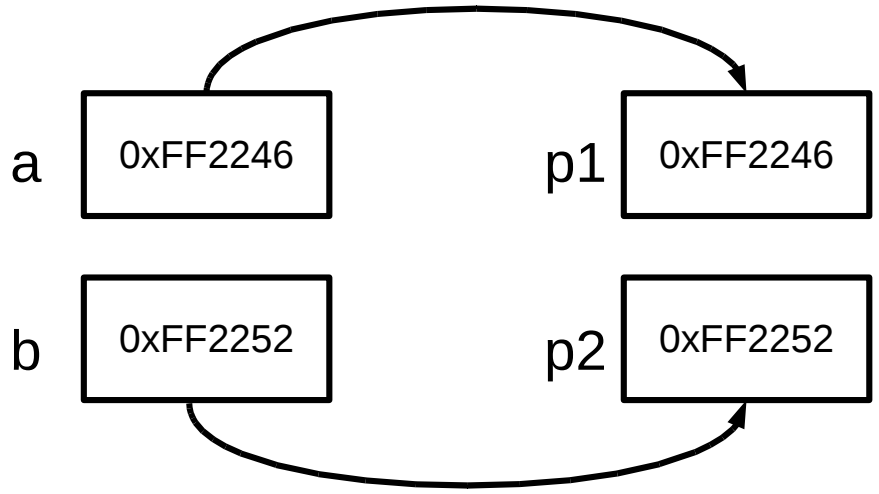
```
1  #include <iostream>
2  using namespace std;
3
4  void plusUn(int *T, size_t n){
5    for (size_t i = 0 ; i < n ; i++)
6      T[i]++;
7  }
8
9  int main(int argc, char** argv) {
10   int *a = new int[2]; a[0] = 3; a[1] = 4;
11   cout << a[0] << ", " << a[1] << endl; // affiche 3,4
12
13   plusUn(a, 2);
14   cout << a[0] << ", " << a[1] <<endl;  // affiche 4,5
15
16   return 0;
17 }
```

Passage de pointeur par valeur : vue mémoire

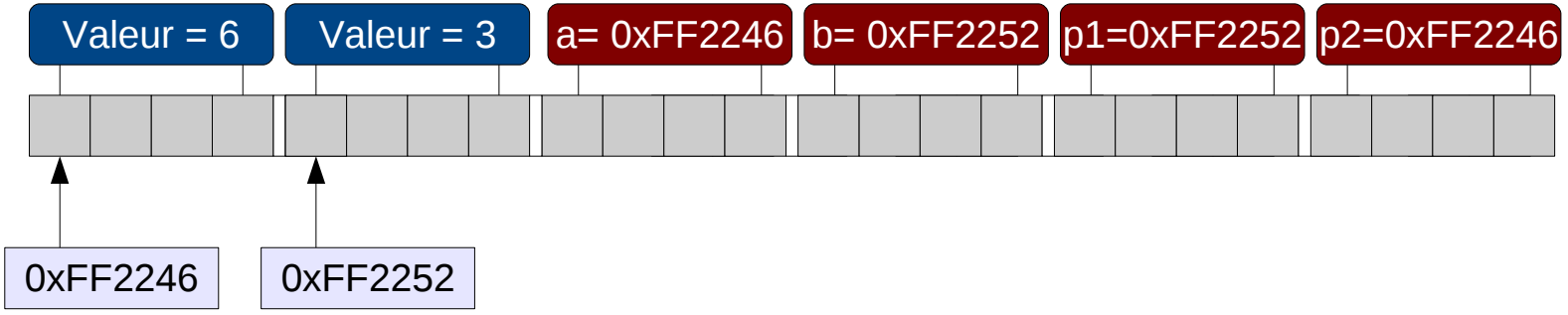
```
1  void echPtr(int* p1, int* p2) {
2    int *p;
3    p = p1;
4    p1 = p2;
5    p2 = p;
6  }
7
8  int main(int argc, char** argv) {
9    int *a,*b;
10   a = new int(6);
11   b = new int(3);
12   echPtr(a, b); // pas d'echange
13   return 0;
14 }
```

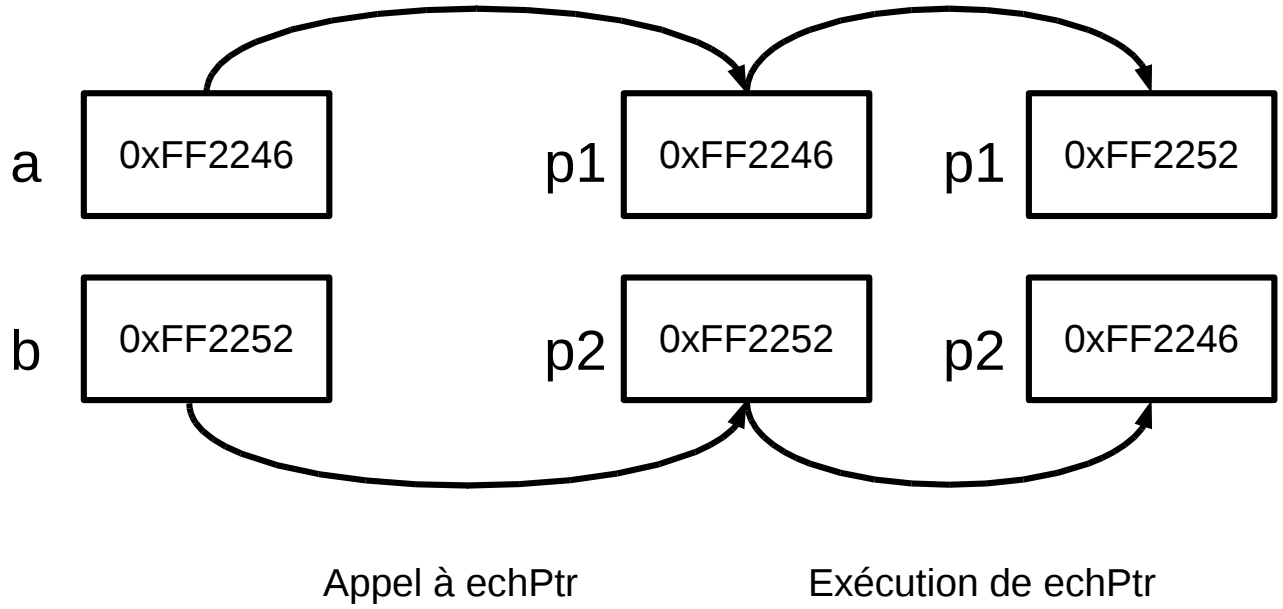

a
0xFF2246
b
0xFF2252

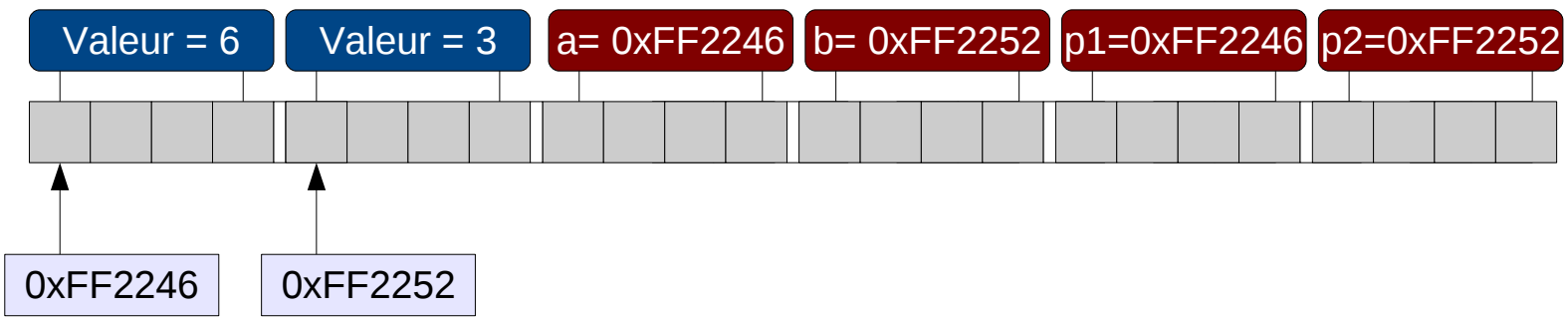

Valeur = 6
Valeur = 3
a= 0xFF2246
b= 0xFF2252
0xFF2246
0xFF2252

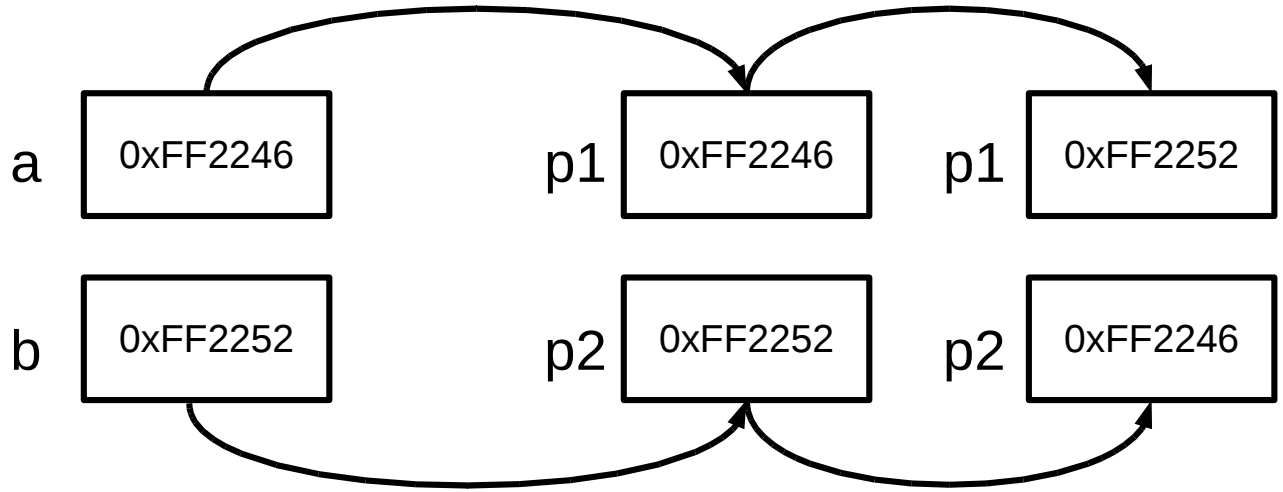


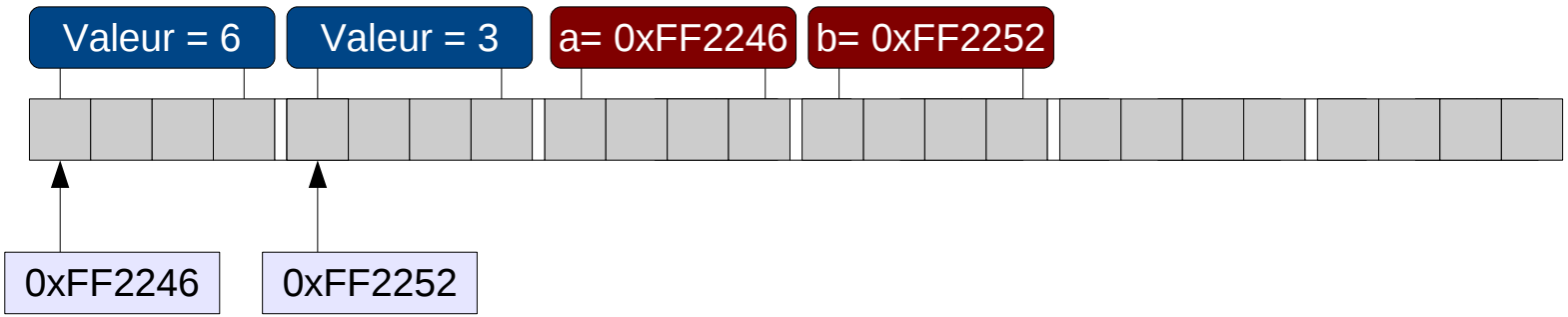
Appel à echPtr




a
0xFF2246
b
0xFF2252
p1
0xFF2246
p2
0xFF2252
p1
0xFF2252
p2
0xFF2246
Appel à echPtr
Exécution de echPtr


Valeur = 6
Valeur = 3
a= 0xFF2246
b= 0xFF2252
p1=0xFF2246
p2=0xFF2252
0xFF2246
0xFF2252


a
0xFF2246
p1
0xFF2246
p1
0xFF2252
b
0xFF2252
p2
0xFF2252
p2
0xFF2246
Appel à echPtr
Exécution de echPtr
Retour de echPtr


Valeur = 6
Valeur = 3
a= 0xFF2246
b= 0xFF2252
0xFF2246
0xFF2252

Plan du cours

Retour de pointeurs

Comme pour le passage de paramètre, il y a deux manières de retourner un pointeur d'une fonction/méthode :

- retour par valeur
- retour par adresse

Retour de pointeurs par valeur

Definition

```
type_retour* mafonction(...) {...}
```

La fonction renvoie une copie d'un pointeur existant ou bien renvoie un pointeur construit à partir d'une adresse mémoire valide.

Exemple

```
int* f(int a) { return &a; }

int* g(int n) {
   int *ptr = new int[n];
   return ptr;
}
```

Retour de pointeurs par adresse

Definition

```
type_retour* &mafonction(...) {...}
```

La fonction renvoie une variable déjà existante de type pointeur (sans faire de copie).
Attention, une adresse mémoire n'est pas une variable.

Exemple

```
int* &dernierTab(int **T, int n) { return T[n-1]; }
```

Retour de pointeur : exemple

```
1  #include <iostream>
2  using namespace std;
3
4  int* plusGrand(int *T, int n){
5    size_t idx_max = 0;
6    for (size_t i = 1 ; i < n ; i++)
7      if (T[i] > T[idx_max]) idx_max = i;
8    return &T[idx_max];
9  }
10
11 int main(int argc, char** argv) {
12   int *T;
13   T = new int[5];
14   for (size_t i = 0 ; i < 5 ; i++) {
15     T[i] = 2 * i + 1;
16   }
17   cout << "plus grand: " << T[4] << " - " << &T[4] << endl
18        << "plus grand: " << *plusGrand(T, 5)
19        << " - " << plusGrand(T, 5) << endl;
20   return 0;
21 }
```

Plan du cours

Rappel passage de paramètre d'une fonction/méthode

Il y a deux méthodes pour passer un objet comme paramètre dans une fonction ou une méthode :

- par adresse :
  c'est l'objet lui même qui est utilisé dans la fonction

- par valeur :
  c'est une copie de l'objet qui est utilisée dans la fonction

Passage par adresse d'un objet avec pointeur

Comme l'objet n'est pas copié, il n'y a aucun problème avec les attributs de type pointeurs

Passage par valeur d'un objet avec pointeur

Fonctionnement :

- la fonction construit une copie locale de l'objet :
  en appelant le constructeur par copie sur cet objet
- la fonction travaille sur la copie locale
- avant de rendre la main, la fonction détruit la copie locale :
  en appelant le destructeur

Attention

- constructeur copie absent $\rightarrow$ copie superficielle
- destructeur absent $\rightarrow$ fuite mémoire

Plan du cours

Pointeurs et déclaration `const`

Le mot clé `const` permet de déclarer des variables constantes :

e.g., `const double Pi = 3.14`.

De la même manière, on peut déclarer des pointeurs constants.

Attention

Le caractère constant peut être appliqué soit au pointeur soit à la zone mémoire pointée

Constante de type pointeur

Definition

```
int * const ptr = val ;
```

- le pointeur `ptr` est constant et ne peut changer sa valeur *val*
- l'initialisation du pointeur doit se faire avec sa déclaration

Exemple

```
int a = 3, b;
int *const ptr_a = &a;
ptr_a = &b; // ERREUR ptr_a est constant
```

Pointeur en lecture seule

Definition

```
const int * ptr = val ;
```

- le pointeur `ptr` peut changer sa valeur *val*
- l'initialisation du pointeur n'est pas obligatoire
- le déréférencement du pointeur `*ptr` est constant

Exemple

```
int a = 3, b; const int *ptr_a;
ptr_a = &a; // OK ptr_a n'est pas constant
b = *ptr_a; // OK *ptr_a n'est pas modifié
*ptr_a = 2; // ERREUR *ptr_a est constant
```

Constante pointeur en lecture seule

Definition

`const int * const ptr =` *val* `;`

- le pointeur `ptr` est constant et ne peut changer sa valeur *val*
- l'initialisation du pointeur doit se faire avec sa déclaration
- le déréférencement du pointeur `*ptr` est constant

Exemple

```
int a = 3, b; const int const *ptr_a = &a;
ptr_a = &b; // ERREUR ptr_a est constant
b = *ptr_a; // OK *ptr_a n'est pas modifié
*ptr_a = 2; // ERREUR *ptr_a est constant
```

Pointeurs et tableaux statiques

Un tableau statique peut être vue comme un pointeur constant :

```
int t[10] ≡ int * const t = new int[10]
```

La différence se fait sur l'implantation sous-jacente :

- le pointeur correspondant au tableau appartient au système et non pas à l'utilisateur
- la mémoire allouée au tableau n'est pas dans le tas mais dans la pile et c'est le système qui la gère

La gestion mémoire d'un programme

Un programme occupe un espace mémoire découpé en plusieurs zones :

- une zone contenant le texte du programme en binaire : segment de texte
- une zone pour les données statiques (variable globale) : segment de données
- une zone pour les variables locales et la chaîne d'activation des fonctions : la pile
- une zone pour les données dynamique : le tas

Lorsque le programme se termine, cette mémoire est libérée automatiquement.

Plan du cours

Plan du cours

Plan du cours

Structure de données classiques

Les structures de données sont la base de l'algorithmique et donc de la programmation :

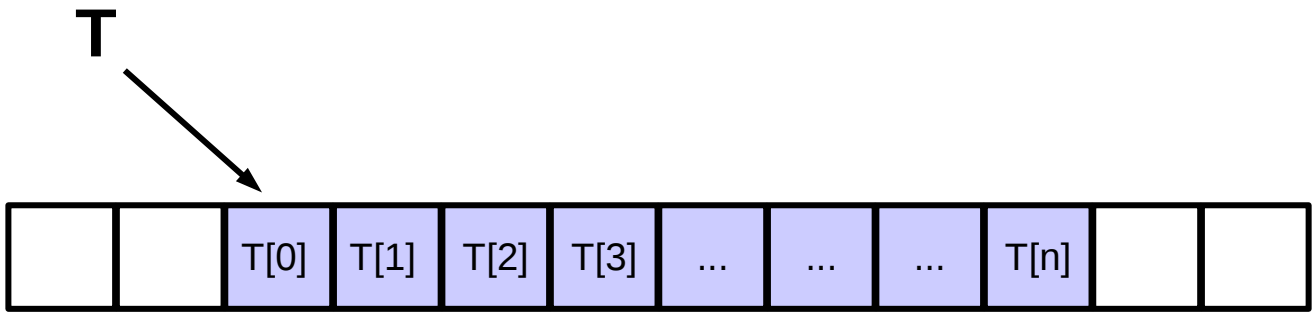
- Tableaux
- Listes
- Piles, Files
- Table de hachage
- Arbres
- ⋮

Attention

Toutes ces structures ont des propriétés différentes. Bien les choisir permet de traiter les problèmes le plus efficacement possible.

Les tableaux

Données structurées linéairement en mémoire, à la manière d'un vecteur.

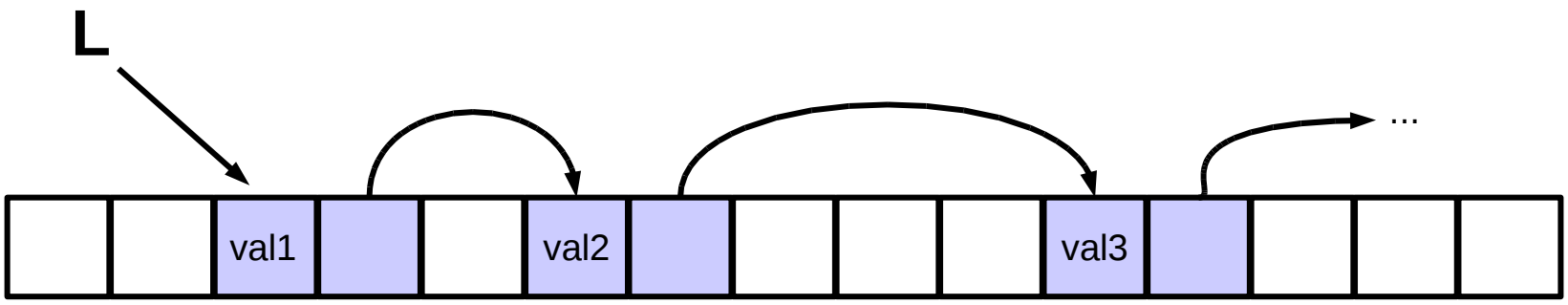


Caractéristiques :

- accès aux éléments en $O(1)$
- insertion/suppression en $O(n)$

Les listes

Données structurées non-linéairement à accès séquentiel.



Caractéristiques :

- accès aux éléments en $O(n)$
- insertion/suppression en $O(1)$

Piles

Données structurées linéairement à partir de :

- un tableau [3]
- un indice du sommet de la pile

Caractéristiques :

- accès à l'élément de tête en $O(1)$
- insertion/suppression en tête en $O(1)$

3. le nombre d'élément est fini

Files

Données structurées linéairement à partir de :

- un tableau [4]
- un indice du début de la file
- un indice de fin de la file

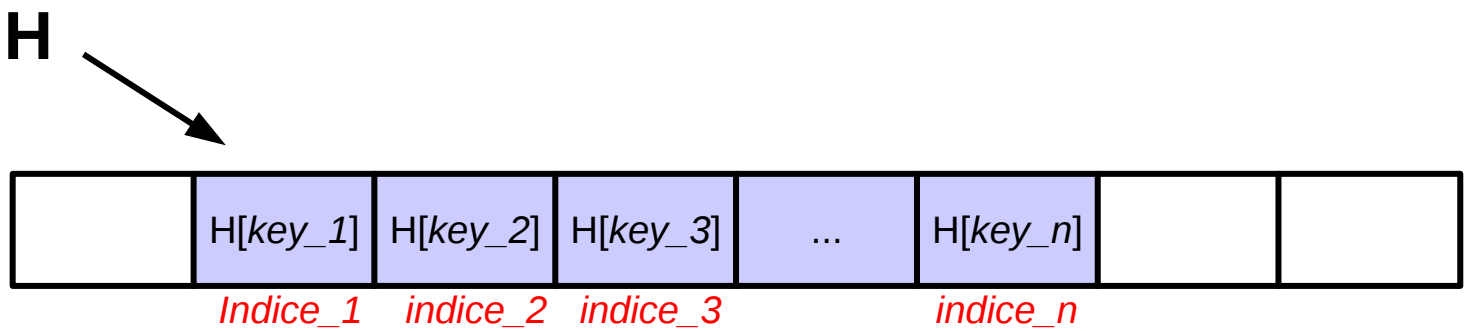
Caractéristiques :

- accès à l'élément de tête en $O(1)$
- insertion en fin en $O(1)$
- suppression en tête en $O(1)$

4. le nombre d'élément est fini

Table de hachage

Données structurées non-linéairement à accès haché (stockage linéaire[5]).



Caractéristiques :

- accès aux éléments en $O(1)$
- insertion/suppression en $O(1)$

5. le nombre d'élément est fini

Plan du cours

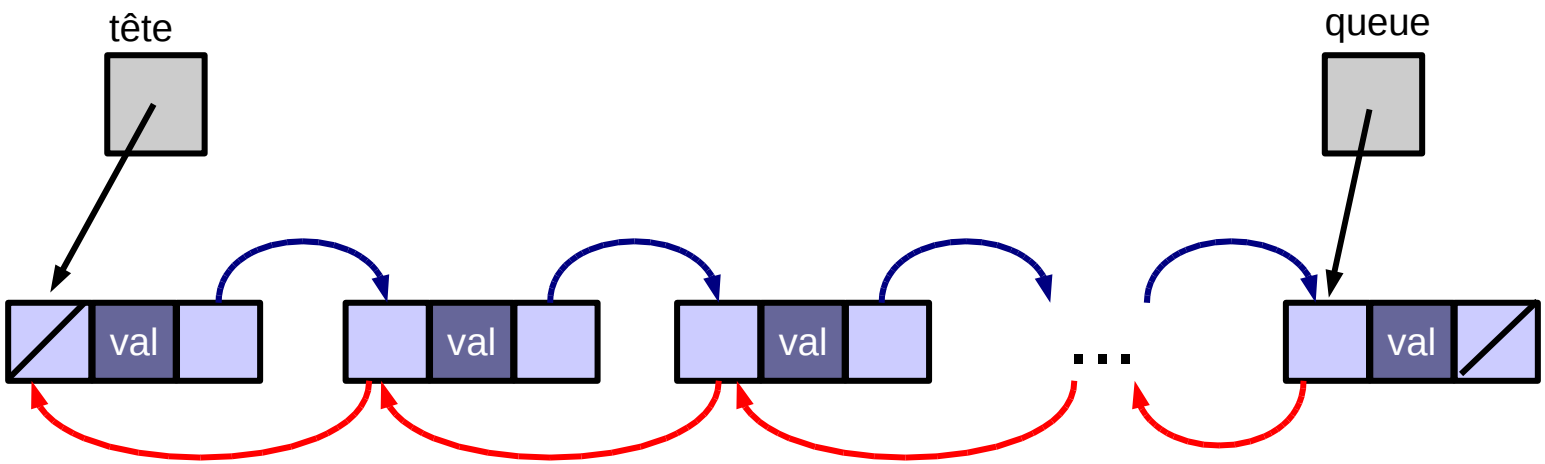
Représentation interne

Definition

Une liste doublement chaînée est un ensemble de triplets :

<précédent, valeur, suivant>

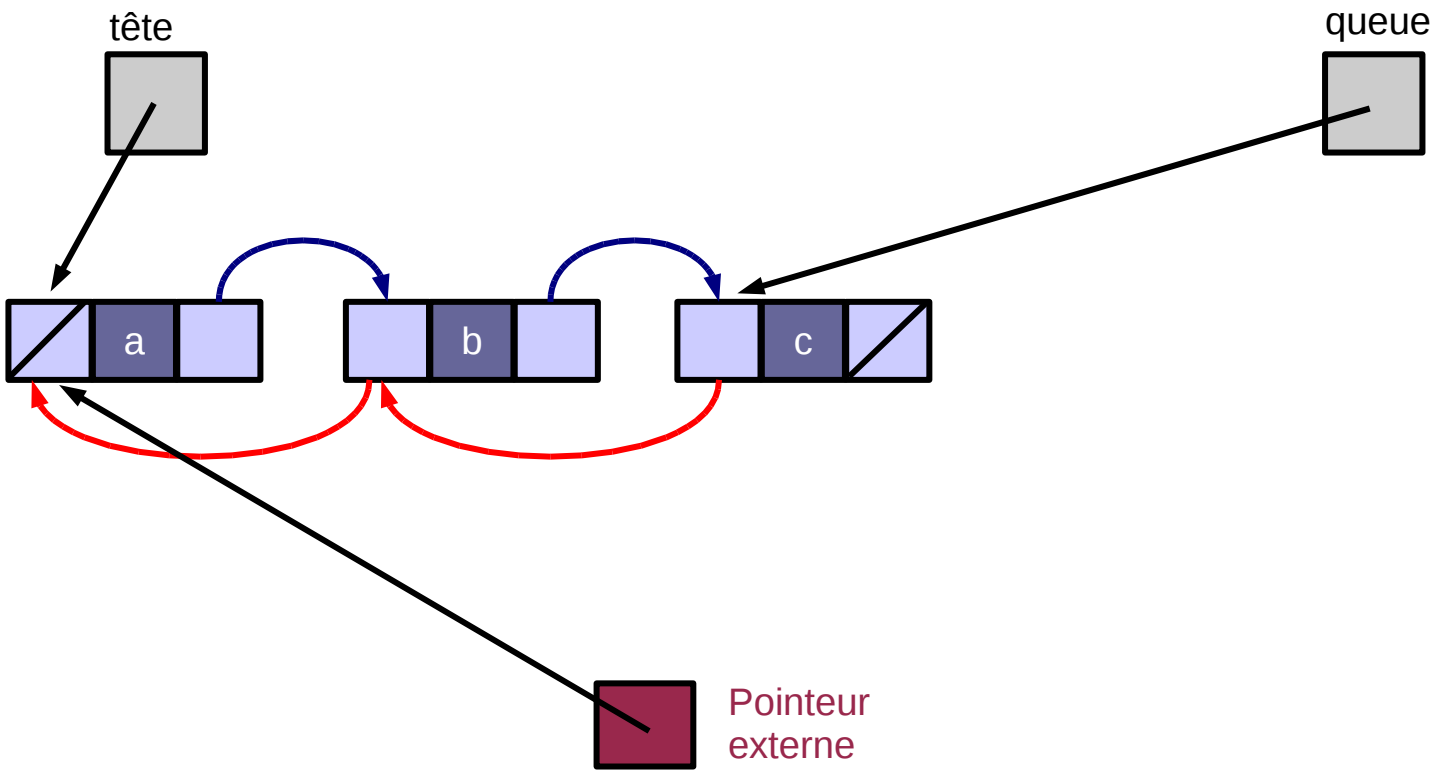
On identifie la liste par son premier et son dernier triplet.



La tête, la queue et les liens sont implantés par des pointeurs

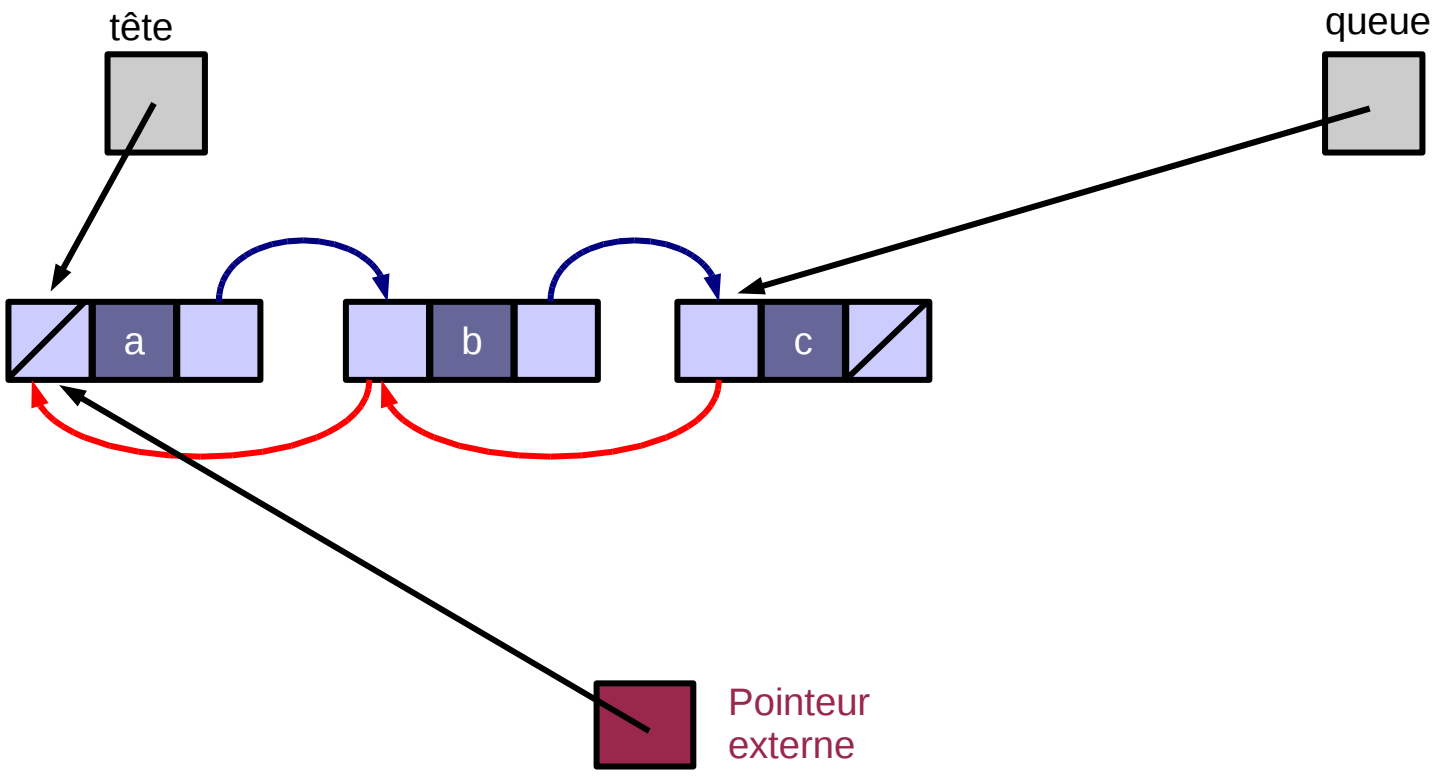
Parcours de la liste

On utilise un pointeur externe qui balaye les éléments de la liste.



Parcours de la liste

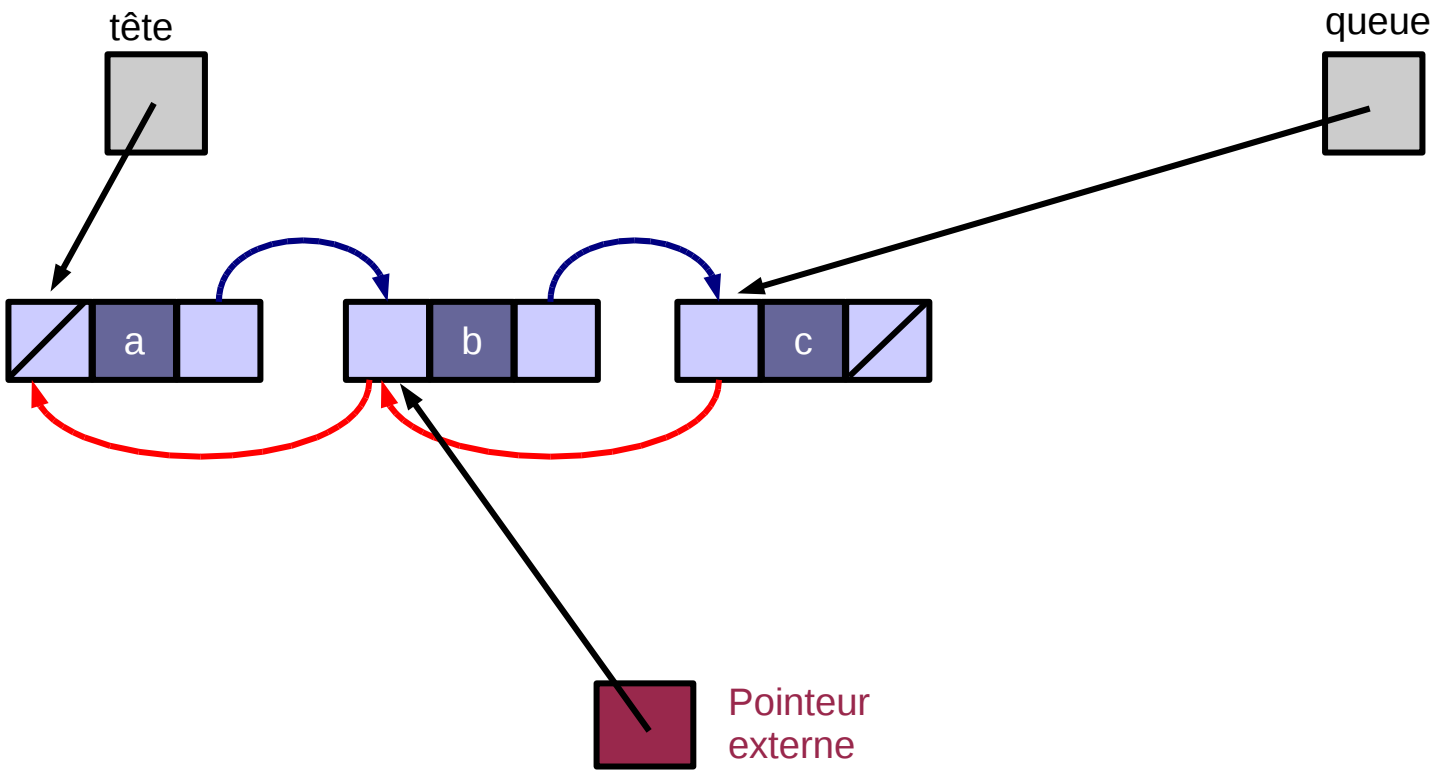
On utilise un pointeur externe qui balaye les éléments de la liste.



On modilie la valeur du pointeur externe en utilisant les pointeurs de précédence et de succession.

Parcours de la liste

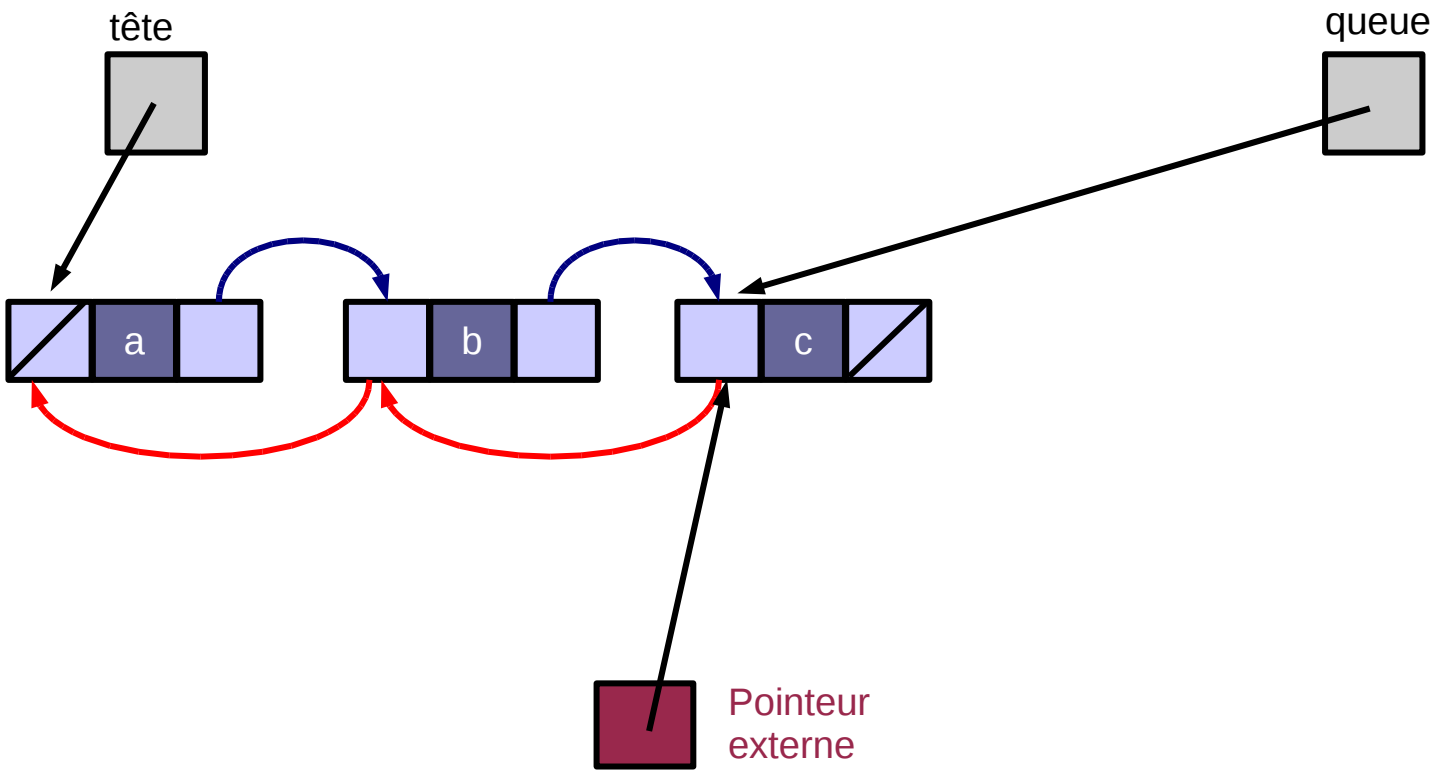
On utilise un pointeur externe qui balaye les éléments de la liste.



On modilie la valeur du pointeur externe en utilisant les pointeurs de précédence et de succession.

Parcours de la liste

On utilise un pointeur externe qui balaye les éléments de la liste.



On modilie la valeur du pointeur externe en utilisant les pointeurs de précédence et de succession.

Accès à un élément de la liste

On utilise le parcours de liste pour trouver l'élément en question :

- accès au $i^{ème}$ élément : parcours liste + comptage
- appartenance à la liste : parcours liste + test d'égalité

On peut parcourir la liste soit en partant du début (*tête*) soit en partant de la fin (*queue*).

Accès à un élément de la liste

On utilise le parcours de liste pour trouver l'élément en question :

- accès au $i^{ème}$ élément : parcours liste + comptage
- appartenance à la liste : parcours liste + test d'égalité

On peut parcourir la liste soit en partant du début (*tête*) soit en partant de la fin (*queue*).

$\hookrightarrow$ operation de complexité linéaire en la taille de la liste

Modification de la liste

Opérations courantes :

- insertion d'un élément en début/en fin de liste
- insertion avant/après un élément désigné
- retrait du premier/dernier élément
- retrait d'un élément désigné

Ces opérations consistent à modifier les pointeurs de précédence et de succession de quelques éléments de la liste (pas de parcours).

Modification de la liste

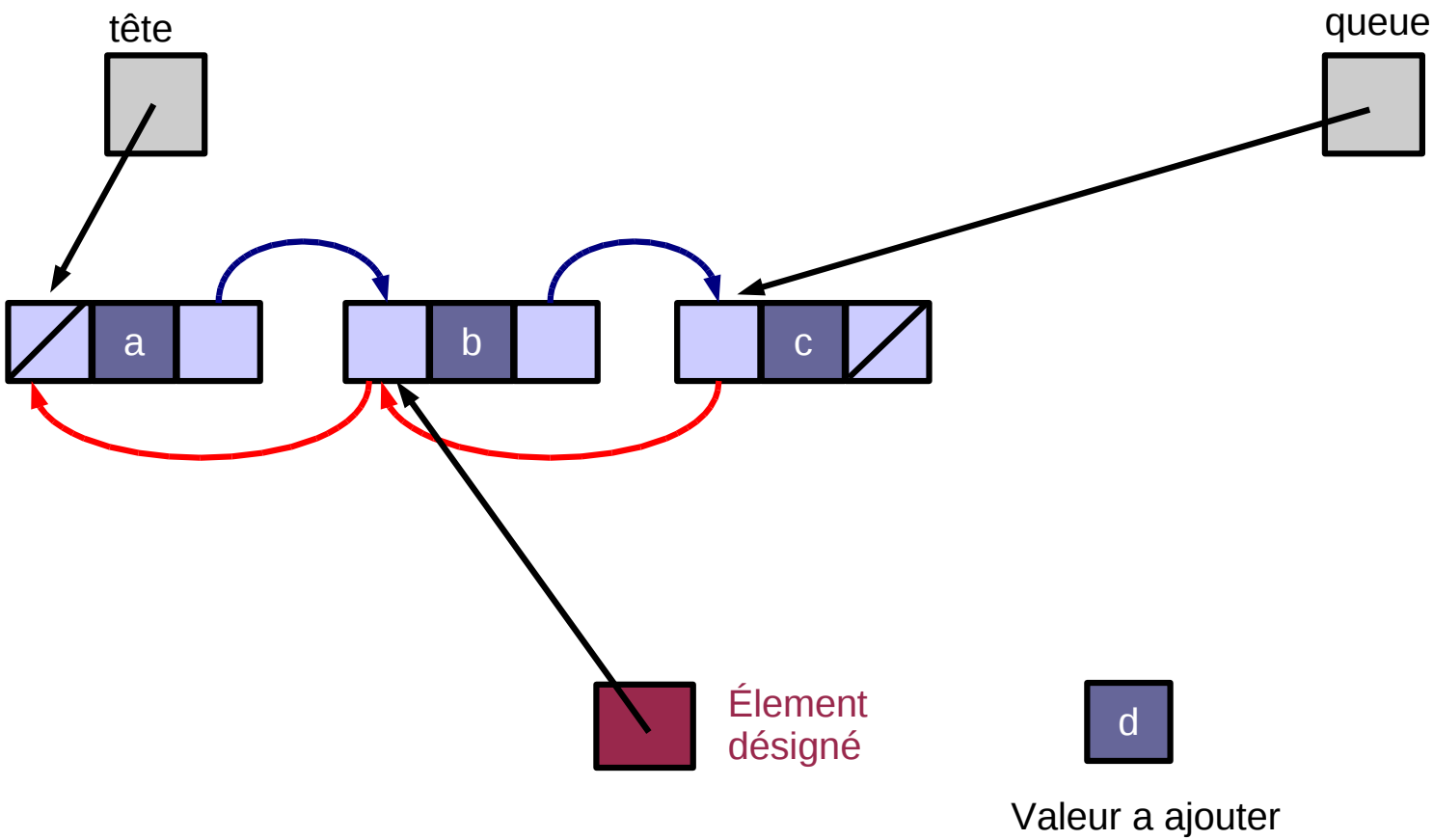
Opérations courantes :

- insertion d'un élément en début/en fin de liste
- insertion avant/après un élément désigné
- retrait du premier/dernier élément
- retrait d'un élément désigné

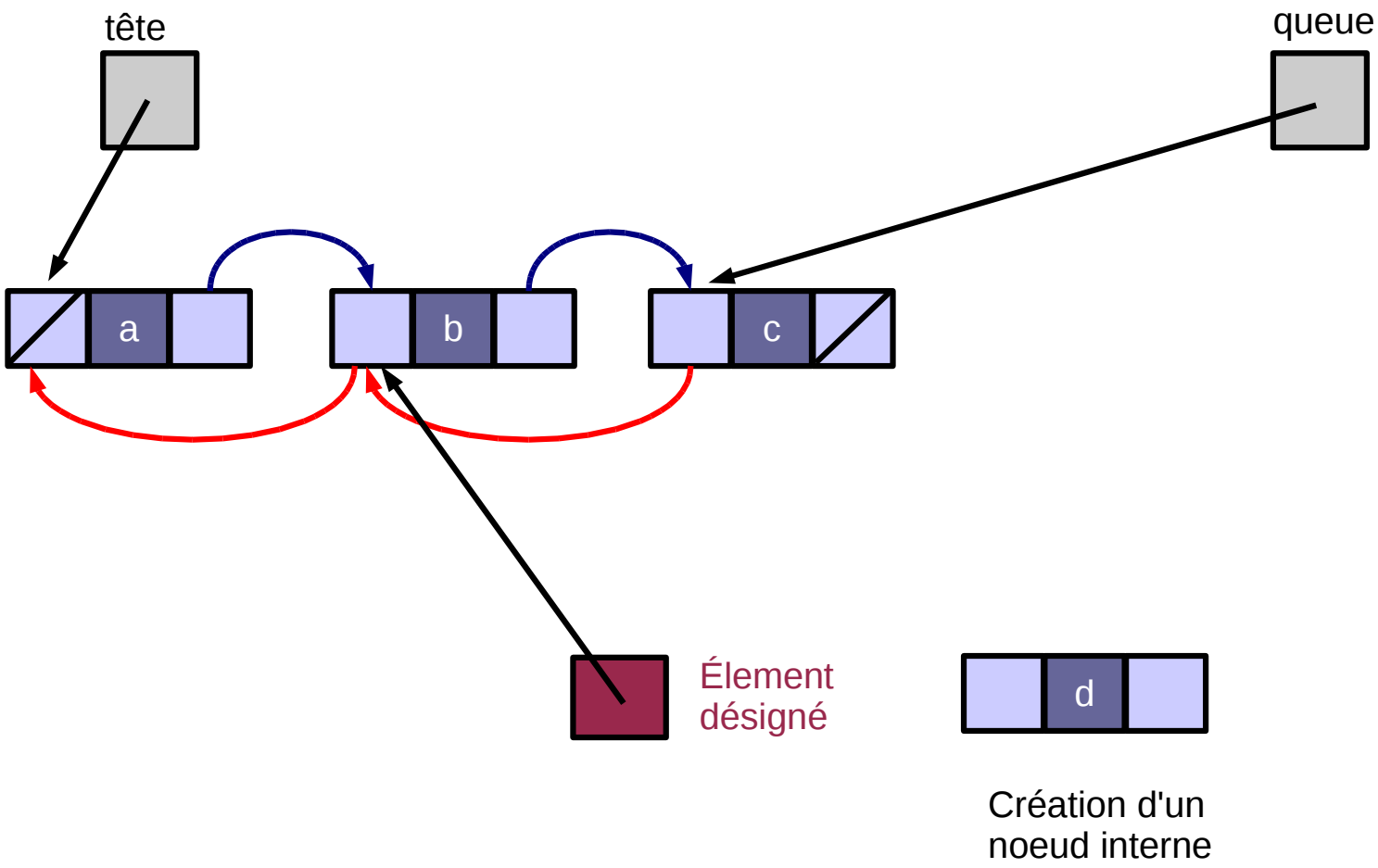
Ces opérations consistent à modifier les pointeurs de précédence et de succession de quelques éléments de la liste (pas de parcours).

$\hookrightarrow$ operation de complexité constante

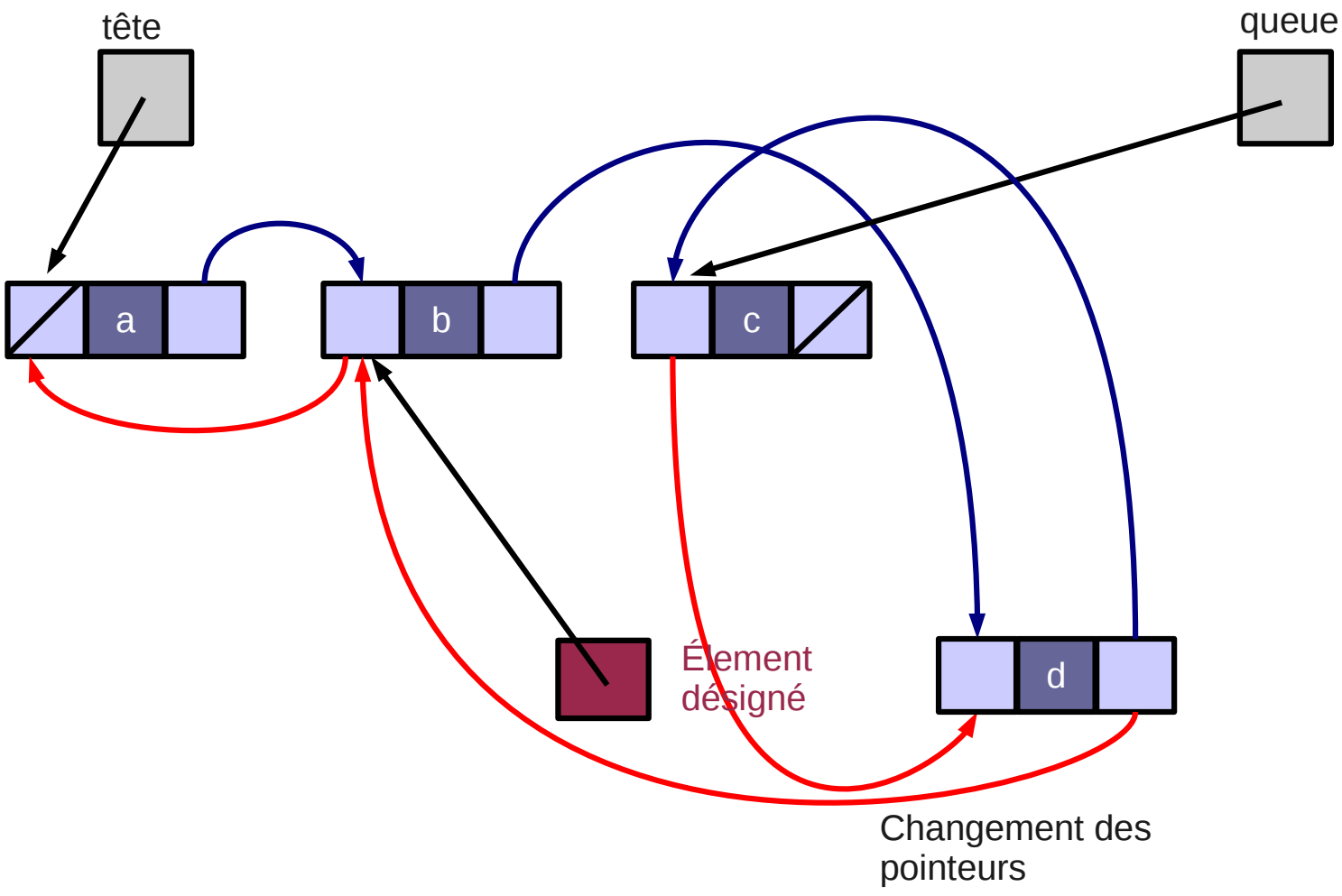
Ajout d'un élément à la liste



Ajout d'un élément à la liste



Ajout d'un élément à la liste



Spécification d'une classe `Liste`

Objectifs :
Séparer les opérations interne à la structure de données des opérations de manipulation de la liste.

concepteur

- définit l'ensemble des fonctionnalités interne à la structure de liste (indépendant des valeurs)
- *e.g.*, positionnement dans la liste, ajout d'un élément, ...

utilisateur

- définit tous ce qui dépend du type des éléments à stocker
- *e.g.*, définition des algorithmes sur les listes (*tri*, *affichage*, ...)

Liste pseudo-générique

Le code d'une liste reste quasiment le même quelque soit le type des valeurs à stockées (`int, float, ..., class`).

Afin de s'abstraire dans le code du type des élément de la liste, nous supposerons que le type des éléments est `TypeVal`.

Remarque

On substituera aisément le type que l'on souhaite utilisé dans la liste en utilisant la macro préprocesseur : `#define TypeVal int`

Opérations nécessaires dans une liste doublement chaînée

On appelle élément désigné d'une liste un élément de la liste référencé par un pointeur.

Opérations de base pour un élément désigné :

- accès en consultation/modification du suivant
- accès en consultation/modification du précédent
- accès en consultation de la valeur

Opérations nécessaires dans une liste doublement chaînée

Opérations de modification de la structure :

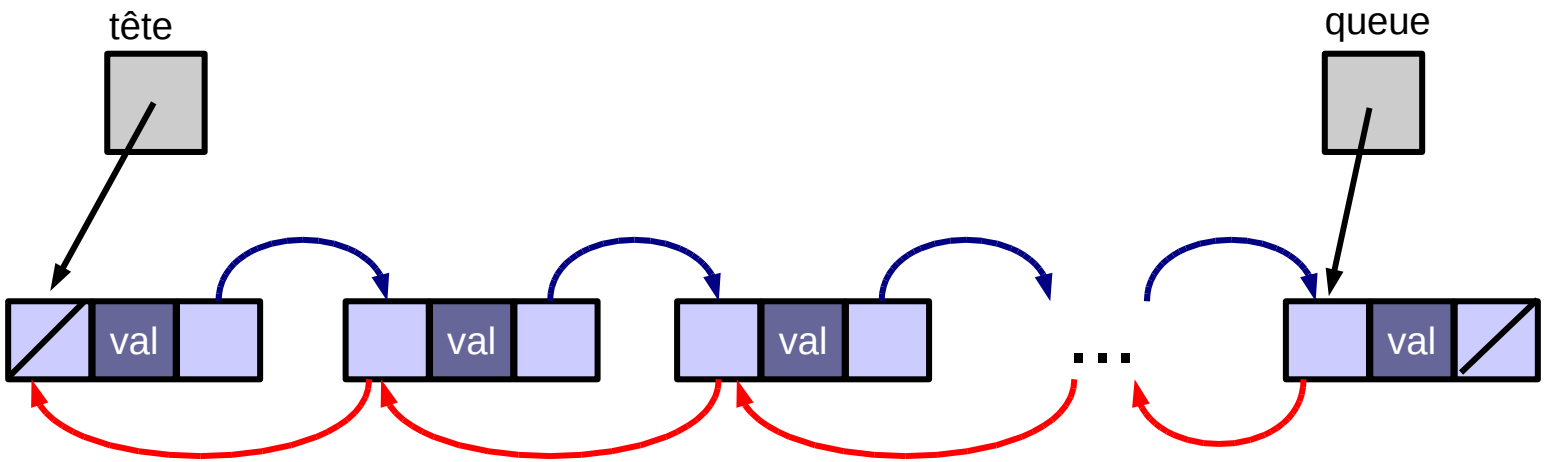
- insertion d'un élément après un élément désigné
- insertion d'un élément avant un élément désigné
- retrait/destruction d'un élément désigné

Opérations souvent utiles :

- renvoyer la longueur de la liste
- tester si la liste est vide
- vider une liste
- copier une liste

Plan du cours

Représentation des triplets : la classe Cell



Un élément de la liste est constitué par :

- un emplacement pour une valeur de type `TypeVal`,
- un pointeur pour désigner l'élément suivant,
- un pointeur pour désigner l'élément précédent.

Représentation des triplets : la classe Cell

Structure minimale pour matérialiser les éléments d'une liste doublement chaînée :

```
1 class Cell {
2   private:
3     TypeVal  val;
4     Cell*    suiv;
5     Cell*    prec;
6 };
```

À cela, il faut rajouter les constructeurs, le destructeur et les accesseurs.

Représentation des triplets : la classe Cell

Structure minimale pour matérialiser les éléments d'une liste doublement chaînée :

```
1 class Cell {
2   private:
3     TypeVal  val;
4     Cell*    suiv;
5     Cell*    prec;
6 };
```

À cela, il faut rajouter les constructeurs, le destructeur et les accesseurs.

Attention

Aucune allocation dynamique dans cette classe, les pointeurs `suiv` et `prec` seront gérés à l'extérieur.

La classe Cell : signature

```
1  #ifndef CELL_H
2  #define CELL_H
3
4  class Cell {
5    private:
6      TypeVal  val;
7      Cell*    suiv;
8      Cell*    prec;
9    public:
10     Cell();          // Constructeur par defaut
11     Cell(TypeVal); // Constructeur parametre
12     TypeVal getVal() const;  // Accesseur en lecture
13     void    setVal(TypeVal); // Accesseur en ecriture
14     Cell*   getPrec() const; // Accesseur en lecture
15     void    setPrec(Cell*);  // Accesseur en ecriture
16     Cell*   getSuiv() const; // Accesseur en lecture
17     void    setSuiv(Cell*);  // Accesseur en ecriture
18 };
19 #endif
```

La classe Cell : implantation

```
1  #include <iostream> // pour NULL
2  #include "Cell.h"
3
4  Cell::Cell() : suiv(NULL), prec(NULL) { }
5  Cell::Cell(TypeVal v) : val(v), suiv(NULL), prec(NULL) { }
6
7  TypeVal Cell::getVal() const      { return val;}
8  void    Cell::setVal(TypeVal v)   { val = v;}
9
10 Cell*   Cell::getPrec() const     { return prec;}
11 void    Cell::setPrec(Cell* c)    { prec = c;}
12
13 Cell*   Cell::getSuiv() const     { return suiv;}
14 void    Cell::setSuiv(Cell* c)    { suiv = c;}
```

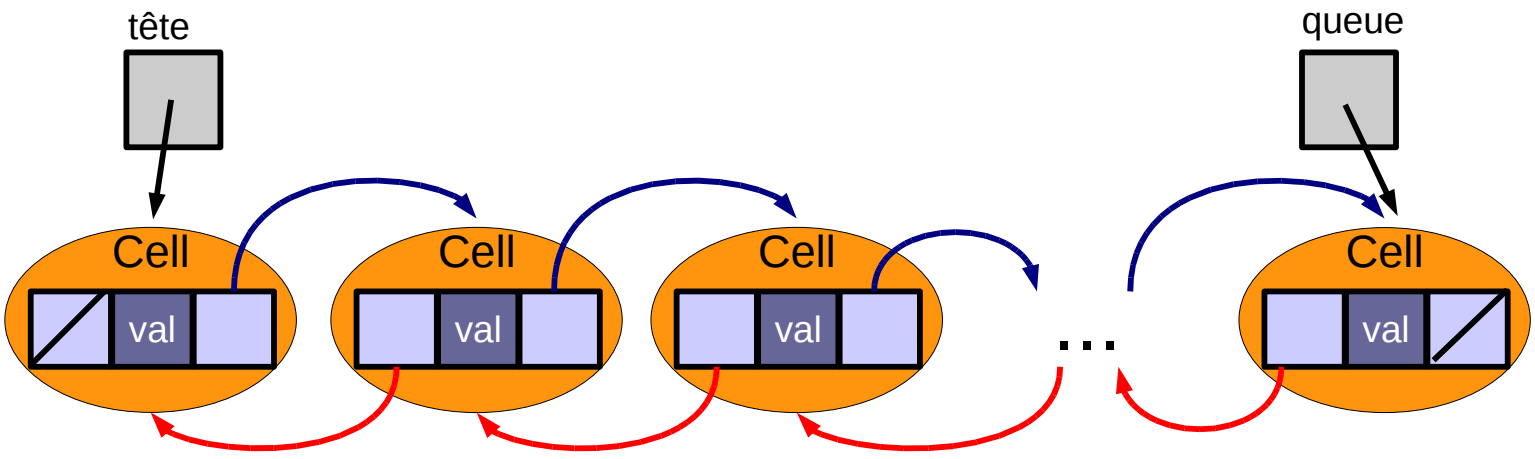
La classe Cell : exemple

Un début de liste *à la main*

```
1  typedef int TypeVal;
2  #include "Cell.h"
3
4  int main(int argc, char** argv) {
5     Cell a(1), b(2), c(3);
6
7     a.setSuiv(&b);
8
9     b.setSuiv(&c);
10    b.setPrec(&a);
11
12    c.setPrec(&b);
13
14    return 0;
15 }
```

Plan du cours

Représentation d'une liste avec la classe Cell



La tête et la queue d'une liste sont des pointeurs sur Cell.

Représentation d'une liste : la classe Liste

Structure minimale pour matérialiser une liste doublement chaînée :

```
1 class Liste {
2   private:
3     Cell*    tete;
4     Cell*   queue;
5     int     nbelem; //nbr d'element dans la liste
6 };
```

Représentation d'une liste : la classe Liste

Structure minimale pour matérialiser une liste doublement chaînée :

```
1 class Liste {
2   private:
3     Cell*    tete;
4     Cell*   queue;
5     int     nbelem; //nbr d'element dans la liste
6 };
```

Attention

C'est la liste qui doit gérer l'allocation et la libération mémoire de ses éléments (`Cell`).

La classe Liste : masquage de la structure interne

Afin de ne pas expliciter la class Cell aux utilisateurs de la liste, on l'encaspule dans la classe Liste :

```
1 class Liste {
2   public:
3     typedef Cell* Place;
4   private:
5     Place  tete;
6     Place  queue;
7     int    nbelem;
8 };
```

Attention

On utilise alors le type `Liste::Place` à la place de `Cell*`, cela rend la classe liste plus cohérente.

La classe Liste : masquage de la structure interne

Afin de faciliter la modification de la structure interne d'une liste, on fournit des accesseurs privés pour la modification des objets de type Cell* (Place) :

```
1  class Liste {
2    ...
3    private:
4      void setSuivant(Place p, Place suiv)   {
5        p->setSuiv(suiv);
6      }
7      void setPrecedent(Place p, Place prec) {
8        p->setPrec(prec);
9      }
10 };
```

Attention

La classe Liste ne doit pas fournir une méthode directe de modification des éléments de sa structure interne.

La classe Liste : constructeur par défaut

Ce constructeur construit la liste vide :

- les attributs tete et queue sont initialisés à NULL
- le nombre d'éléments est mis à zéro

```
1 class Liste {
2   ...
3   public:
4     Liste() : tete(NULL), queue(NULL), nbelem(0) {
5     }
6 };
```

La classe Liste : opérations d'information

Certaines opérations auront un comportement différent suivant si la liste est vide ou non :

```
1 bool Liste::estVide() const {
2   return !tete; // equivalent a return tete == NULL;
3 }
```

Certaines opérations auront besoin de connaître la longueur de la liste :

```
1 int Liste::longueur() const {
2   return nbelem;
3 }
```

La classe Liste : accesseurs publiques

```
1  class Liste {
2    ...
3    public:
4      Place premier() const { return tete; }
5      Place dernier() const { return queue; }
6
7      Place   suivant(Place p) const {
8        return p->getSuiv();
9      }
10     Place precedent(Place p) const {
11       return p->getPrec();
12     }
13     TypeVal  valeur(Place p) const {
14       return p->getVal();
15     }
16 };
```

Insertion d'une valeur dans une liste

En premier lieu, il faut créer une cellule interne pour représenter un nouvel élément contenant la donnée `TypeVal val`.

```
1 Cell* = new Cell(val);
```

Ensuite, il faut positionner cet élément dans la liste :

- insertion en début de liste
- insertion en fin de liste
- insertion après un élément désigné
- insertion avant un élément désigné

Insertion positionnée dans la liste

Rappel

Un élément désigné correspond à un élément de la liste référencé par un pointeur externe (ici une `Place`).

Deux types d'insertion possibles :

- insertion d'une valeur après un élément désigné
- insertion d'une valeur avant un élément désigné

```
1 void ajoutAvant(Place p, TypeVal v);
2 void ajoutApres(Place p, TypeVal v);
```

Insertion après un élément désigné de la liste

```
1 void ajoutApres(Place p, TypeVal v);
```

Trois cas possibles :

- insertion dans une liste vide[6] : aucun voisin à modifier
- insertion en fin de liste : 1 seul voisin à modifier
- insertion en milieu de liste : 2 voisins à modifier

6. dans ce cas p=NULL

Insertion après un élément désigné de la liste

```
1  void Liste::ajoutApres(Liste::Place p, Liste::TypeVal v) {
2    Cell *elt = new Cell(val);
3    nbelem++;
4    if (!p) { // aucun voisin
5      tete = queue = elt;
6    } else {
7      if (p == queue){ // 1 seul voisin
8        setPrecedent(elt, p);
9        setSuivant(p, elt);
10       queue = elt;
11     } else { // 2 voisins
12       setPrecedent(elt, p);
13       setSuivant(elt, suivant(p));
14       setPrecedent(suivant(p), elt);
15       setSuivant(p, elt);
16     }
17   }
18 }
```

Insertion avant un élément désigné de la liste

```
1 void ajoutAvant(Place p, TypeVal v);
```

Trois cas possibles :

- insertion dans une liste vide[7] : aucun voisin à modifier
- insertion en debut de liste : 1 seul voisin à modifier
- insertion en milieu de liste : 2 voisins à modifier

7. dans ce cas p=NULL

Insertion avant un élément désigné de la liste

```
1  void Liste::ajoutAvant(Liste::Place p, Liste::TypeVal v) {
2    Cell *elt = new Cell(val);
3    nbelem++;
4    if (!p) { // aucun voisin
5      tete = queue = elt;
6    } else {
7      if (p == tete){ // 1 seul voisin
8        setSuivant(elt, p);
9        setPrecedent(p, elt);
10       tete = elt;
11     } else { // 2 voisins
12       setSuivant(elt, p);
13       setPrecedent(elt, precedent(p));
14       setSuivant(precedent(p), elt);
15       setPrecedent(p, elt);
16     }
17   }
18 }
```

Insertion en début et en fin de liste

```
1 void Liste::ajoutDebut(Liste::TypeVal val) {
2   ajoutAvant(tete,val);
3 }
```

```
1 void Liste::ajoutFin(Liste::TypeVal val) {
2   ajoutApres(queue, val);
3 }
```

La classe `Liste` : constructeur par copie

Attention

Il faut spécifier explicitement un constructeur par copie pour avoir une copie profonde de la liste.

Il suffit de :

- parcourir la liste à copier en partant de la `tete` vers la queue
- rajouter chaque valeur parcourue à la liste courante en effectuant une insertion en fin de liste

La classe Liste : constructeur par copie

```
1 Liste::Liste(const Liste &L) : tete(NULL), queue(NULL),
2                                nbelem(0) {
3   for (Place p = L.premier(); p; p = L.suivant(p)) {
4     ajoutFin(L.valeur(p));
5   }
6 }
```

La classe `Liste` : destructeur

Attention

Si la liste n'est pas vide, il faut expliciter la destruction de l'ensemble des cellules qui compose la liste (allocation dynamique).

Il suffit de

- parcourir la liste à détruire en partant de la `tete` vers la queue
- supprimer chaque élément parcourue en ne modifiant que partiellement la liste pour continuer le parcours

La classe Liste : destructeur et *nettoyeur*

Histoire de factoriser un peu notre code source, on fournira la méthode `void vider();` qui sera utilisée dans le destructeur.

```
1  void Liste::vider() {
2    Place p = tete;
3    while (p) {
4      p = suivant(p);
5      delete tete;
6      tete = p;
7    }
8    tete = queue = NULL; nbelem = 0;
9  }
10
11 Liste::~Liste() {
12   vider();
13 }
```

Suppression d'un élément dans une liste

Il faut garder en tête que l'élément retiré de la liste doit être libérer de la mémoire (allocation dynamique)

Il y quatre cas de suppression possibles :

- suppression de l'unique élément d'une liste
- suppression du premier élément d'une liste
- suppression du dernier élément d'une liste
- suppression d'un élément au milieu d'une liste

Suppression d'un élément désigné

```
1  void Liste::enleve(Liste::Place p){
2    if (!p) { return; }
3    if (nbelem == 1) {
4      tete = queue = NULL;
5    } else {
6      if (p == tete) {
7        tete = suivant(p);
8        setPrecedent(tete, NULL);
9      } else {
10       if (p == queue) {
11         queue = precedent(p);
12         setSuivant(queue, NULL);
13       } else {
14         setSuivant(precedent(p), suivant(p));
15         setPrecedent(suivant(p), precedent(p));
16       }
17     }
18   }
19   nbelem--;
20   delete p;
21 }
```

La classe Liste : opérateur de copie

Attention

- L'opérateur = fournit par le compilateur n'est pas satisfaisant ! ! ! copie des pointeurs.
- La liste recevant la copie n'est pas forcément vide ! ! ! besoin de vider.

La classe Liste : opérateur de copie

```
1 Liste& Liste::operator=(const Liste &L) {
2   if (this != &L) {
3     this->vider();
4     for (Place p = L.premier(); p; p = L.suivant(p)) {
5       ajoutFin(L.valeur(p));
6     }
7   }
8   return *this;
9 }
```

Plan du cours

Plan du cours

Plan du cours

Surcharge d'opérateurs : définition

Definition

On appelle surcharge *statique* le fait de définir plusieurs fonctions ou méthodes ayant le même nom, mais des paramètres différents.

Exemple

```
void affiche(int a) { cout << a << endl; }
void affiche(double b) { cout << b << endl; }
...
```

Rq : *ceci n'est pas possible en C.*

Surcharge d'opérateurs : objectif

Un des objectifs de la surcharge des opérateurs est d'étendre la syntaxe du langage :

Exemple

```
int a, b, c;
...
a = b + c;
c = a * b;
```

Il serait intéressant de pouvoir étendre ces opérateurs à des types non-natifs du langage (une classe par exemple).

Surcharge d'opérateurs : objectif

Un des objectifs de la surcharge des opérateurs est d'étendre la syntaxe du langage :

Exemple

```
class fraction {...};
fraction a, b, c;
...
a = b + c;
c = a * b;
```

Le principe de la surcharge d'opérateur est d'autoriser la définition d'opérateurs *déjà existant* en C++ pour des types définis par l'utilisateur.

Surcharge d'opérateurs : que peut-on surcharger ?

Tous les opérateurs du C++ sont surchargeables **SAUF**

- `objet.membre`
- `objet.*pointeur_vers_membre`
- `nom_classe::membre`
- `exp?exp:exp`

Attention

Pour les opérateurs surchargeables, on ne peut pas changer :

- le nombre d'opérandes
- la priorité
- le sens d'associativité

Surcharge d'opérateurs : que peut-on surcharger ?

Ainsi, on peut par exemple surcharger :

- les opérateurs arithmétique (binaire et unaire) : `+,-,*,/,%`
- les opérateurs de comparaison : `==,!=,<,>,<=,>=`
- l'opérateur d'indicage : `[]`
- ...

Surcharge d'opérateurs : comment les définir ?

Deux possibilités :

- par une méthode de la classe
- par une fonction

Exemple

```
class fraction {
   fraction operator*(const fraction&) const;
};
```

ou

```
fraction operator*(const fraction&, const fraction&);
```

Surcharge d'opérateurs : comment l'utiliser ?

par appel explicite :

```
fraction a, b, c;
a = b.operator*(c);  // methode
a = operator*(b, c); // fonction
```

par appel implicite :

```
fraction a, b, c;
a = b * c; // methode et fonction
```

Surcharge d'opérateurs : comment choisir ?

Attention

Certains opérateurs ne peuvent être surchargés qu'au travers de leurs méthodes :

- `operator=` l'opérateur d'affectation
- `operator[]` l'opérateur d'indicage
- `operator()` l'opérateur d'appel de fonction
- `operator->` l'opérateur de sélection de membres (pointeur)

Surcharge d'opérateurs : comment choisir ?

Attention

Il faut choisir entre méthodes et fonctions, les deux surcharges ne peuvent pas coexister ! ! !

Remarque

La méthode est souvent préférable à la fonction sauf si l'on souhaite autoriser l'opérateur par conversion implicite des opérandes.

Surcharge d'opérateurs : comment choisir ?

```
1  class fraction {
2    public: fraction operator*(const fraction&) const;
3  };
4  fraction operator*(const fraction&, const fraction&);
5
6  int main(int argc, char** argv) {
7    fraction a, b, c;
8    a = b * c;
9    return 0;
10 }
```

Erreur de compilation

*In function 'int main(int, char**)' :*
13 : error : ISO C++ says that these are ambiguous, even though the worst conversion for the first is better than the worst conversion for the second :
9 : note : candidate 1 : fraction operator(const fraction&, const fraction&)*
6 : note : candidate 2 : fraction fraction : :operator(const fraction&) const*

Déclaration formelle de la surcharge d'opérateurs

Définition dans une méthode de classe

```
classe NomClasse {
   ...
   TypeRetour OpBinaire(TypeOp op2)
}
```

- `TypeRetour` est le type de retour de l'opérateur
- `OpBinaire` est le nom de l'opérateur à surcharger
- `TypeOp` est le type de l'opérande de droite `op2`
- `*this` est l'opérande de gauche

Exemple d'opérateur binaire

```
1  class fraction {
2    public:
3      int num, den;
4      ...
5      fraction operator*(const fraction &x) const {
6        fraction f(*this);
7        f.num *= x.num;
8        f.den *= x.den;
9        return f;
10   }
11 };
```

Déclaration formelle de la surcharge d'opérateurs

Définition dans une méthode de classe

```
classe NomClasse{
   ...
   TypeRetour OpUnaire()
}
```

- `TypeRetour` est le type de retour de l'opérateur
- `OpUnaire` est le nom de l'opérateur à surcharger
- `*this` est la seule opérande

Exemple d'opérateur unaire

```
1  class fraction{
2    public:
3      int num, den;
4      ...
5      fraction operator-() const {
6        fraction f(*this);
7        f.num = -f.num;
8        return f;
9      }
10 };
```

Déclaration formelle de la surcharge d'opérateurs

Définition dans une fonction

```
TypeRetour OpBinaire(TypeOp1 op1, TypeOp2 op2) {...}
TypeRetour  OpUnaire(TypeOp2 op2) {...}
```

- `TypeRetour` est le type de retour de l'opérateur
- `OpBinaire, OpUnaire` est le nom de l'opérateur à surcharger
- `TypeOp1` est le type de l'opérande de gauche `op1`
- `TypeOp2` est le type de l'opérande de droite `op2`

Exemple d'opérateurs : par fonction

```
1  class fraction {
2    public:
3      int num, den;
4  };
5
6  fraction operator*(const fraction &a, const fraction &b) {
7    fraction f(a);
8    f.num *= b.num;
9    f.den *= b.den;
10   return f;
11 }
12
13 fraction operator-(const fraction &a){
14   fraction f(a);
15   f.num = -f.num;
16   return f;
17 }
```

Plan du cours

Surcharge de l'opérateur d'affectation

Nous l'avons déjà utilisé dans les classes :

- Soit le compilateur en fournit un par défaut (copie bit à bit des attributs)
- Soit le concepteur de la classe surcharge la méthode du compilateur en proposant sa propre version

Attention

- Il faut bien respecter la signature de l'opérateur d'affectation
- Il faut toujours le surcharger par une méthode

Surcharge de l'opérateur d'affectation

Definition

```
MaClasse {
   ...
   MaClasse& operator=(const MaClasse& a) {...}
};
```

Remarque :

Cet opérateur renvoi toujours une référence sur l'objet courant pour permettre l'enchaînement d'affectation `a = b = c;`.

Exemple : opérateur d'affectation

```
1  class fraction{
2    public:
3      int num, den;
4
5      fraction& operator=(const fraction& a) {
6        if (this != &a){
7          num = a.num;
8          den = a.den;
9        }
10       return *this;
11     }
12 };
```

Plan du cours

Surcharge des opérateurs de comparaison

On peut surcharge tous les opérateurs de comparaison :

- `==` et `!=`
- `<` et `>`
- `<=` et `>=`

Remarque

On peut surcharger ces opérateurs soit par une méthode soit par une fonction.

Surcharge des opérateurs de comparaison

Définition dans une méthode

```
class MaClasse {
   ...
   bool operator!=(const MaClasse& b) const;
};
```

Définition dans une fonction

```
bool operator!=(const MaClasse& a, const MaClasse& b);
```

Exemple : surcharge du test d'inégalité

```
1 class fraction {
2   public:
3     int num, den;
4
5     bool operator!=(const fraction& a) const {
6       return ((num != a.num) or (den != a.den));
7     }
8 };
```

Surcharge des opérateurs de comparaison

Remarque

Il faut que l'utilisation des opérateurs de comparaison ait du sens pour l'objet :

- le test d'égalité et d'inégalité ont souvent du sens
- les autres opérateurs impliquent une relation d'ordre sur les objets

Remarque

On peut définir des opérateurs de comparaison entre des objets de type différent (si cela a du sens)

Comparaison d'objets différents

On peut comparer deux objets différent si cela à du sens :

Attention

Il faut toujours que l'un des deux objets soit un type utilisateur :

- toujours le cas si on surcharge par méthode
- peut ne pas être le cas si on surcharge par fonction

Exemple

```
bool operator<(const fraction& a, const double& b) {
   return (double(a.num) / double(a.den)) < b;
}
```

Plan du cours

Les opérateurs d'entrée-sortie

En C++, on utilise :

- `cin >> a;` pour saisir une valeur à `a`
- `cout << a;` pour afficher la valeur de `a`

quand `a` est une donnée de type natif du C (`int double, ...`).

Remarque :

Il est possible de surcharger ces entrées-sorties pour des types utilisateurs

Les opérateurs d'entrée-sortie

En fait :

- `cin >>` correspond à l'appel de l'opérateur `operator>>` de la classe de l'objet `cin` (la classe `istream`)
- `cout <<` correspond à l'appel de l'opérateur `operator<<` de la classe de l'objet `cout` (la classe `ostream`)

Comme l'objet appelant de ces opérateurs sont soit de la classe `istream` soit de la classe `ostream`, il faut forcement surcharger ces opérateurs au travers de fonction pour des types utilisateurs.

Surcharge de l'opérateur d'affichage

Definition

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

- la 1re opérande correspond au flot de sortie (`cout`) passé par référence car il est modifié par l'affichage
- la 2e opérande correspond à l'objet qu'on souhaite afficher (passage constant par référence)
- le type de retour est `ostream&` pour enchaîner les affichages `cout << x << y;`

Exemple : opérateur d'affichage

```
1 class fraction {
2   public:
3     int num, den;
4 };
5
6 ostream& operator<<(ostream& os, const fraction& a) {
7   return os << a.num << "/" << a.den << endl;
8 }
```

Attention

la fonction `operator<<` est en dehors de la classe `fraction`, elle n'a donc pas accès aux attributs privés...

Surcharge de l'opérateur de saisie

Definition

```
istream& operator>>(istream&, MaClasse&);
```

- la 1re opérande correspond au flot d'entrée (`cin`) passé par référence car il est modifié par l'affichage
- la 2e opérande correspond à l'objet qu'on souhaite saisir (passage non-constant par référence)
- le type de retour est `istream&` pour enchaîner les saisies `cin >> x >> y;`

Exemple : opérateur de saisie

```
1 class fraction {
2   public:
3     int num, den;
4 };
5
6 istream& operator>>(istream& is, fraction& a) {
7   return is >> a.num >> a.den;
8 }
```

Attention

la fonction `operator>>` est en dehors de la classe `fraction`, elle n'a donc pas accès aux attributs privés...

Surcharge des opérateurs d'entrée-sortie

Une bonne gestion des attributs privés est de définir des méthodes publiques d'affichage et de saisie paramétrées par un flot.

```
1  class fraction {
2    private:
3      int num, den;
4    public:
5      ostream& affiche(ostream& os) const {
6        return os << num << "/" << den << endl;
7      }
8  };
9
10 ostream& operator<<(ostream& os, const fraction &a) {
11   return a.affiche(os);
12 }
```

Surcharge des opérateurs d'entrée-sortie

```
1  #include <iostream>
2  using namespace std;
3
4  class fraction {
5    private:
6      int num, den;
7    public:
8      istream& saisie(istream& is) {
9        return is >> num >> den;
10     }
11     ostream& affiche(ostream& os) const {
12       return os << num << "/" << den << endl;
13     }
14 };
15
16 ostream& operator<<(ostream& os, const fraction& a) {
17   return a.affiche(os);
18 }
19 istream& operator>>(istream& is, fraction& a) {
20   return a.saisie(is);
21 }
```

Surcharge des opérateurs d'entrée-sortie

```
1 int main(int argc, char** argv) {
2   fraction a;
3   cin >> a;
4   cout << "la fraction est: " << a << endl;
5   return 0;
6 };
```

Plan du cours

Surcharge des opérateurs arithmétiques

Même caractéristiques que pour les opérateurs de comparaison

Definition

```
class MaClasse {
   ...
   MaClasse operator+(const MaClasse&) const;
};
```

Attention

Les opérateurs arithmétiques renvoient généralement un nouvel objet différent des opérandes : besoin des constructeur par copie et par défaut

Plan du cours

Surcharge des opérateurs [] et ()

Ils sont définis forcément par une méthode :

operator[] est un opérateur binaire

```
class MaClasse {
   ...
   typeRetour operator[](typeIndice) const;
   typeRetour& operator[](typeIndice) ;
};
```

operator() est un opérateur n-aire

```
class MaClasse {
   ...
   typeRetour operator()(type1, type2, ...);
};
```

Surcharge des opérateurs ++ et --

Chacun de ces deux opérateurs existe en deux versions :
⇒ pré/post-incrément/décrément

operator++ et operator-- sont des opérateurs unaires

```
class MaClasse {
   ...
   /* opérateurs de pré-incrémentation */
   MaClasse &operator++();
   MaClasse &operator--();
   /* opérateurs de post-incrémentation */
   const MaClasse operator++(int);
   const MaClasse operator--(int);
};
```

Surcharge des opérateurs de transtypage

Definition

Le transtypage (`cast` en anglais) est une opération qui permet la transformation d'une instance d'un type (ou d'une classe) donné(e) en une instance d'un autre type (d'une autre classe).

Le transtypage existe nativement pour les conversions entre types primitifs.

Transtypage personnalisés

Le transtypage peut être implicite (*e.g.*, `int i = 3.14`) ou explicite.

Dans certains cas, il est utile d'expliquer au compilateur comment faire...

Surcharge des opérateurs de transtypage

Exemple

```
const float pi = 3.14;
cout << "pi = " << pi << endl
   << "(int) pi = " << (int) pi << endl
   << "(char) 15.*pi = " << (char) 15.*pi << endl
   << "(char) (15.*pi) = " << (char) (15.*pi) << endl
   << "(char) pi*15. = " << (char) pi*15. << endl;
```

qui donne :

```
pi = 3.14
(int) pi = 3
(char) 15.*pi = 47.1
(char) (15.*pi) = /
(char) pi*15. = 45
```

Surcharge des opérateurs de transtypage

Définition d'un opérateur de transtypage

```
class MaClasse {
   ...
   operator type_desire(void);
};
```

Exemple

```
class fraction {
   ...
   operator double () {
      return ((double) num)/den;
   }
};
```

Plan du cours

Plan du cours

Plan du cours

Contexte
Définitions

Polymorphisme (dictionnaire de l'académie française – 9e édition)

n. m. XIXe siècle. Dérivé de polymorphe.
Caractère de ce qui se présente sous différentes formes.

Polymorphe (dictionnaire de l'académie française – 9e édition)

adj. XIXe siècle. Emprunté du grec *polumorphos*, « qui a plusieurs formes », lui-même composé à l'aide de *polus*, « abondant, nombreux », et *morphê*, « forme ».
Qui offre des formes différentes, est sujet à changer de forme.

En informatique polymorphisme paramétrique $\Rightarrow$ fonctions qui peuvent être appliquées à des types paramétrés.

Contexte
Motivation

La généricité introduite dans la classe `Liste` par l'utilisation de `TypeVal` est peu satisfaisante.

$\Rightarrow$ Difficile avec ce modèle de disposer dans un même programme d'une liste d'entiers (avec `#define TypeVal int`) et d'une liste de `Points` (étant donné une classe `Point`).

Remarque

Introduction d'une classe générique permettrait de définir à tout moment une liste d'entiers, une liste de réels, une liste de `Points`, ..., dans un même programme en faisant figurer le type des éléments comme paramètre ($\neq$ attribut) de la classe.

Contexte
Utilisation simple

Supposons qu'il existe la classe paramétrique `Liste` permettant de définir une liste quel que soit le type de données qu'elle contient.

Les opérations sur la liste demeurent génériques (ajout/suppression d'un élément, longueur de la liste, ...).

On peut alors écrire :

```
1 Liste<int>     L1;
2 Liste<Point>   L2;
3 L1.ajoutFin(3);
4 L2.ajoutDebut(Point(2, -2));
```

Le paramétrage est entre < et > lors de la déclaration des instances.

Plan du cours

Syntaxe

Définir une classe paramétrique, c'est définir un modèle ⇒ *template* en anglais.

Le mot-clé `template` permet de définir les arguments du modèle, le mot-clé `typename` permet de spécifier que l'argument est un nom de type (ou de classe).

```
1  template <typename T>
2  class MaClasse {
3    private:
4      T attr;
5    public:
6      MaClasse();
7      MaClasse(T);
8      T getAttr() const;
9      void setAttr(T);
10 };
```

Syntaxe

Définir une classe paramétrique, c'est définir un modèle ⇒ *template* en anglais.

Le mot-clé `template` permet de définir les arguments du modèle, le mot-clé `typename` permet de spécifier que l'argument est un nom de type (ou de classe).

```
1  template <typename T>
2  class MaClasse {
3    private:
4      T attr;
5    public:
6      MaClasse();
7      MaClasse(const T &);        // pas de copie
8      const T &getAttr() const; // pas de copie
9      void setAttr(const T &);  // pas de copie
10 };
```

Plan du cours

Exemple

Retour sur la classe `Cell` : signature

```
1  #ifndef __CELL_H__
2  #define __CELL_H__
3
4  template <typename T>
5  class Cell {
6    private:
7      T          val;
8      Cell<T> *prec, *suiv;
9    public:
10     Cell();                          // Constructeur par defaut
11     Cell(const T &);                 // Constructeur parametre
12     const T &getVal() const;         // Accesseur en lecture
13     void       setVal(const T &);    // Accesseur en ecriture
14     Cell<T>* getPrec() const;        // Accesseur en lecture
15     void       setPrec(Cell<T> *);   // Accesseur en ecriture
16     Cell<T>* getSuiv() const;        // Accesseur en lecture
17     void       setSuiv(Cell<T> *);   // Accesseur en ecriture
18 };
19 #endif
```

Exemple

Retour sur la classe `Cell` : implémentation

```
1  #include <iostream> // pour NULL
2  #include "Cell.h"
3
4  template <typename T>
5  Cell<T>::Cell(): val(), prec(NULL), suiv(NULL) {}
6  template <typename T>
7  Cell<T>::Cell(const T &v): val(v), prec(NULL), suiv(NULL) {}
8
9  template <typename T>
10 const T &Cell<T>::getVal() const { return val; }
11 template <typename T>
12 void Cell<T>::setVal(const T &v) { val = v; }
13 template <typename T>
14 Cell<T> *Cell<T>::getPrec() const { return prec; }
15 template <typename T>
16 void Cell<T>::setPrec(Cell<T> *c) { prec = c; }
17 template <typename T>
18 Cell<T> *Cell<T>::getSuiv() const { return suiv; }
19 template <typename T>
20 void Cell<T>::setSuiv(Cell<T>* c) { suiv = c;}
```

Exemple

Retour sur la classe `Liste` : signature

```
1  template <typename T>
2  class Liste {
3    public:
4      typedef Cell<T>* Place;
5
6    private:
7      Place    tete;
8      Place    queue;
9      int      nbelem;
10
11     void setSuivant(Place p, Place suiv);
12     void setPrecedent(Place p, Place prec);
13
14   public:
15     Liste();
16     Liste(const Liste<T> &L);
17     ~Liste();
18     Liste<T> &operator=(const Liste<T> &L);
19     ...
20 };
```

Exemple

Retour sur la classe `Liste` : signature

```
1  template <typename T>
2  class Liste {
3      ...
4
5    public:
6      ...
7      bool estVide() const;
8      void vider();
9      size_t longueur() const;
10     Place premier() const;
11     Place dernier() const;
12     Place precedent(Place p) const;
13     Place suivant(Place p) const;
14     const T &valeur(Place p) const;
15     void ajoutAvant(Place p, const T &v);
16     void ajoutApres(Place p, const T &v);
17     void ajoutDebut(const T &v);
18     void ajoutFin(const T &v);
19     void enleve(Place p);
20 };
```

Plan du cours

Quelques considérations pratiques

Attention

Les `templates` sont instanciés à la compilation.

$\Rightarrow$ Le fichier d'implémentation n'est donc pas directement compilé (comme le seraient les fichiers d'extensions `.cpp`).

Quelques considérations pratiques

Attention

Les `templates` sont instanciés à la compilation.

⇒ Le fichier d'implémentation n'est donc pas directement compilé (comme le seraient les fichiers d'extensions `.cpp`).

- Il faut inclure l'implémentation des templates dans les fichiers sources qui les utilisent.

Quelques considérations pratiques

Attention

Les `templates` sont instanciés à la compilation.

⇒ Le fichier d'implémentation n'est donc pas directement compilé (comme le seraient les fichiers d'extensions `.cpp`).

- Il faut inclure l'implémentation des templates dans les fichiers sources qui les utilisent.

⇒ 2 solutions ½ :

1. Maintient de la règle « 1 classe = 2 fichiers »
   - l'extension du fichier d'entête peut rester `.h`
   - l'extension du fichier d'implémentation doit pouvoir se différencier : `.tcc` ou `.tpp`

Quelques considérations pratiques

Attention

Les `templates` sont instanciés à la compilation.

⇒ Le fichier d'implémentation n'est donc pas directement compilé (comme le seraient les fichiers d'extensions `.cpp`).

- Il faut inclure l'implémentation des templates dans les fichiers sources qui les utilisent.

⇒ 2 solutions ½ :

1. Maintient de la règle « 1 classe = 2 fichiers »
   - l'extension du fichier d'entête peut rester `.h`
   - l'extension du fichier d'implémentation doit pouvoir se différencier : `.tcc` ou `.tpp`

2. Création d'une nouvelle règle « 1 template = 1 fichier »
   - Extension `.h`, `.tcc` ou `.tpp`

Quelques considérations pratiques

Attention

Les `templates` sont instanciés à la compilation.

⇒ Le fichier d'implémentation n'est donc pas directement compilé (comme le seraient les fichiers d'extensions `.cpp`).

- Il faut inclure l'implémentation des templates dans les fichiers sources qui les utilisent.

⇒ 2 solutions ½ :

1. Maintient de la règle « 1 classe = 2 fichiers »
   - l'extension du fichier d'entête peut rester `.h`
   - l'extension du fichier d'implémentation doit pouvoir se différencier : `.tcc` ou `.tpp`
   - le fichier `.h` inclue le fichier `.tpp` (après la déclaration).
2. Création d'une nouvelle règle « 1 template = 1 fichier »
   - Extension `.h`, `.tcc` ou `.tpp`

Plan du cours

Plus loin avec les *templates*

Modèles de fonctions

```
1  using namespace std;
2
3  void afficher(const Liste<int> &l) {
4    bool first = true;
5    cout << "[";
6    for (Liste<int>::Place p = l.premier();
7         p; p = l.suivant(p)) {
8      cout << (first ? "" : ", ") << l.valeur(p);
9      first = false;
10   }
11   cout << "]" << endl;
12
13 }
```

Plus loin avec les *templates*

Modèles de fonctions

```
1  using namespace std;
2
3  ostream &operator<<(ostream &os, const Liste<int> &l) {
4    bool first = true;
5    os << "[";
6    for (Liste<int>::Place p = l.premier();
7         p; p = l.suivant(p)) {
8      os << (first ? "" : ", ") << l.valeur(p);
9      first = false;
10   }
11   os << "]" << endl;
12   return os;
13 }
```

Plus loin avec les *templates*

Modèles de fonctions

```
1  using namespace std;
2
3  ostream &operator<<(ostream &os, const Liste<int> &l) {
4    bool first = true;
5    os << "[";
6    for (Liste<int>::Place p = l.premier();
7         p; p = l.suivant(p)) {
8      os << (first ? "" : ", ") << l.valeur(p);
9      first = false;
10   }
11   os << "]" << endl;
12   return os;
13 }
```

- Possibilité de faire des modèles de fonctions(algorithmes).

Plus loin avec les *templates*

Modèles de fonctions

```
1  using namespace std;
2  template <typename T>
3  ostream &operator<<(ostream &os, const Liste<T> &l) {
4    bool first = true;
5    os << "[";
6    for (Liste<T>::Place p = l.premier();
7         p; p = l.suivant(p)) {
8      os << (first ? "" : ", ") << l.valeur(p);
9      first = false;
10   }
11   os << "]" << endl;
12   return os;
13 }
```

- Possibilité de faire des modèles de fonctions(algorithmes).

Plus loin avec les *templates*
Modèles de fonctions

```
1  using namespace std;
2  template <typename T>
3  ostream &operator<<(ostream &os, const Liste<T> &l) {
4    bool first = true;
5    os << "[";
6    for (Liste<T>::Place p = l.premier(); // erreur
7         p; p = l.suivant(p)) {
8      os << (first ? "" : ", ") << l.valeur(p);
9      first = false;
10   }
11   os << "]" << endl;
12   return os;
13 }
```

- Possibilité de faire des modèles de fonctions(algorithmes).

Rappel

L'instantiation des templates est faite à la compilation.

Plus loin avec les *templates*

Modèles de fonctions

```
1  using namespace std;
2  template <typename T>
3  ostream &operator<<(ostream &os, const Liste<T> &l) {
4    bool first = true;
5    os << "[";
6    for (typename Liste<T>::Place p = l.premier(); // Ok!!!
7         p; p = l.suivant(p)) {
8      os << (first ? "" : ", ") << l.valeur(p);
9      first = false;
10   }
11   os << "]" << endl;
12   return os;
13 }
```

- Possibilité de faire des modèles de fonctions(algorithmes).
- Il faut parfois aider le compilateur.

Plus loin avec les *templates*

Modèles de fonctions

```
1  using namespace std;
2  template <typename T>
3  ostream &operator<<(ostream &os, const T &l) { // erreur!!!
4    bool first = true;
5    os << "[";
6    for (typename T::Place p = l.premier();
7         p; p = l.suivant(p)) {
8      os << (first ? "" : ", ") << l.valeur(p);
9      first = false;
10   }
11   os << "]" << endl;
12   return os;
13 }
```

- Possibilité de faire des modèles de fonctions(algorithmes).
- Il faut parfois aider le compilateur.
- Mais il faut faire attention à ne pas être trop générique. . .

Plus loin avec les *templates*
Modèles algorithmiques

- Plusieurs modèles permettent de représenter des collections d'objets
  - $\Rightarrow$ listes simplement chaînées, listes doublement chaînées, tableaux, ...

Plus loin avec les *templates*
Modèles algorithmiques

- Plusieurs modèles permettent de représenter des collections d'objets
  - $\Rightarrow$ listes simplement chaînées, listes doublement chaînées, tableaux, ...
- Mise à disposition d'un ensemble de fonctions génériques :
  - afficher une collection,
  - rechercher une valeur dans une collection,
  - trier une collection,
  - fusionner deux collections,
  - ...

Plus loin avec les *templates*
Modèles algorithmiques

- Plusieurs modèles permettent de représenter des collections d'objets
  - $\Rightarrow$ listes simplement chaînées, listes doublement chaînées, tableaux, ...
- Mise à disposition d'un ensemble de fonctions génériques :
  - afficher une collection,
  - rechercher une valeur dans une collection,
  - trier une collection,
  - fusionner deux collections,
  - ...

$\Rightarrow$ Nécessite que les collections soient définies selon un même schéma algorithmique.

Plus loin avec les *templates*

Modèles algorithmiques

- Supposons que toutes nos modèles de collection définissent chacune :
  - un type `Place`
  - une méthode `premier()`,
  - une méthode `suivant()`
  - une méthode `valeur()`.

Plus loin avec les *templates*

Modèles algorithmiques

- Supposons que toutes nos modèles de collection définissent chacune :
  - un type `Place`
  - une méthode `premier()`,
  - une méthode `suivant()`
  - une méthode `valeur()`.

$\Rightarrow$ Il est alors possible de définir une fonction générique qui effectue un traitement de la collection.

Plus loin avec les *templates*

Modèles algorithmiques

```
1  using namespace std;
2  template <typename T>
3  ostream &afficher(ostream &os, const T &l) {
4    bool first = true;
5    os << "[";
6    for (typename T::Place p = l.premier();
7         p; p = l.suivant(p)) {
8      os << (first ? "" : ", ") << l.valeur(p);
9      // requiert que l'element soit affichable
10     first = false;
11   }
12   os << "]" << endl;
13   return os;
14 }
```

Plus loin avec les *templates*

Modèles algorithmiques

```
1  using namespace std;
2  template <typename T, typename U>
3  typename T::Place recherche(const T &l, const U &v) {
4    typename T::Place p = l.premier();
5    while (p && (l.valeur(p) != v)) {
6      // requiert que les elements soient comparables
7      p = l.suivant(p);
8    }
9    return p;
10 }
```

Plus loin avec les *templates*

Modèles algorithmiques

```
1  template <typename T> void tribulle(T &l) {
2    bool estTrie = l.longueur() < 2;
3    typename T::Place last = l.dernier();
4    while (!estTrie) {
5      estTrie = true;
6      typename T::Place cur = l.premier();
7      while (cur != last) {
8        typename T::Place nxt = l.suivant(cur);
9        if (l.valeur(cur) > l.valeur(nxt)) {
10         // requiert que les elements soient munis
11         // d'une relation d'ordre
12         echanger(l, cur, nxt); // Cette fonction doit
13                                // etre definie...
14         estTrie = false;
15       }
16       if (nxt != last) cur = nxt;
17       else last = cur;
18     }
19   }
20 }
```

Plus loin avec les *templates*

Spécialisation

Il est possible de définir un *template* relativement générique et par ailleurs de fournir une version spécifique pour un paramétrage donné du modèle.

$\Rightarrow$ C'est la spécialisation des modèles.

Par exemple, étant donnée une collection `CollTriee` dont la spécification assure à tout moment que celle-ci est triée, il est possible de définir une spécialisation de la fonction `tribulle` précédente :

```
1 template <> void tribulle(CollTriee &l) { }
```

Plus loin avec les *templates*

Autres remarques

Attention

Le mot clé `class` est parfois utilisé à la place de `typename` dans la déclaration du *template*. C'est équivalent, mais source de confusion.

Cette pratique est donc à éviter.

Plus loin avec les *templates*

Autres remarques

Remarque

Il est possible de définir des constantes d'un type donné dans les paramètres d'un *template*.

```
template <typename T, size_t N>
class Matrice {
   private:
      T mat[N][N];
   :
};
```

Plus loin avec les *templates*

Autres remarques

Remarque

Il est possible de définir des constantes d'un type donné dans les paramètres d'un *template*.

```
template <typename T, size_t N>
class Matrice {
   private:
      T mat[N][N];
   ⋮
};
```

Définir des constantes dans les *templates* a souvent pour objectif d'optimiser les performances (vitesse) à l'exécution mais ralentit considérablement la compilation et peut augmenter drastiquement la taille des codes générés.

```
Matrice<bool, 15> mb;
Matrice<int, 20> mi;
```

Plan du cours

Plan du cours

Plan du cours

Polymorphisme et valeurs par défaut

Le polymorphisme permet de définir plusieurs fonctions/méthodes de même nom.

```
1  void AfficheTableau(int *tab, size_t n,
2                      const string &sep, const string &delim1,
3                      const string &delim2,
4                      bool newline) {
5    cout << delim1;
6    for (size_t i = 0; i < n, i++) {
7      cout << tab[i] << (i < n - 1 ? sep : delim2);
8    }
9    if (newline) {
10     cout << endl;
11   }
12 }
```

Polymorphisme et valeurs par défaut

Le polymorphisme permet de définir plusieurs fonctions/méthodes de même nom.

```
1  void AfficheTableau(int *tab, size_t n,
2                      const string &sep, const string &delim1,
3                      const string &delim2,
4                      bool newline) {
5    cout << delim1;
6    for (size_t i = 0; i < n, i++) {
7      cout << tab[i] << (i < n - 1 ? sep : delim2);
8    }
9    if (newline) {
10     cout << endl;
11   }
12 }
16 ...
17 void AfficheTableau(int *tab, size_t n) {
18   AfficheTableau(tab, n, ", ", "[", "]", true);
19 }
```

Polymorphisme et valeurs par défaut

Le polymorphisme permet de définir plusieurs fonctions/méthodes de même nom.

```
1  void AfficheTableau(int *tab, size_t n,
2                      const string &sep, const string &delim1,
3                      const string &delim2,
4                      bool newline) {
5    cout << delim1;
6    for (size_t i = 0; i < n, i++) {
7      cout << tab[i] << (i < n - 1 ? sep : delim2);
8    }
9    if (newline) {
10     cout << endl;
11   }
12 }
23 ...
24 void AfficheTableau(int *tab, size_t n, const string &sep) {
25   AfficheTableau(tab, n, sep, "[", "]", true);
26 }
```

Polymorphisme et valeurs par défaut

Le polymorphisme permet de définir plusieurs fonctions/méthodes de même nom.

```
1  void AfficheTableau(int *tab, size_t n,
2                      const string &sep, const string &delim1,
3                      const string &delim2,
4                      bool newline) {
5    cout << delim1;
6    for (size_t i = 0; i < n, i++) {
7      cout << tab[i] << (i < n - 1 ? sep : delim2);
8    }
9    if (newline) {
10     cout << endl;
11   }
12 }
30 ...
31 void AfficheTableau(int *tab, size_t n,
32                      const string &sep, const string &delim1,
33                      const string &delim2) {
34   AfficheTableau(tab, n, sep, delim1, delim2, true);
35 }
```

Polymorphisme et **valeurs par défaut**

Une autre façon de gérer cette surcharge est de fournir des valeurs par défaut aux arguments **les plus à droite** de la fonction.

Polymorphisme et **valeurs par défaut**

Une autre façon de gérer cette surcharge est de fournir des valeurs par défaut aux arguments **les plus à droite** de la fonction.

```
1  void AfficheTableau(int *tab, size_t n,
2                      const string &sep = ", ",
3                      const string &delim1 = "[",
4                      const string &delim2 = "]",
5                      bool newline = true) {
6    cout << delim1;
7    for (size_t i = 0; i < n, i++) {
8      cout << tab[i] << (i < n - 1 ? sep : delim2);
9    }
10   if (newline) {
11     cout << endl;
12   }
13 }
```

Polymorphisme et **valeurs par défaut**

Une autre façon de gérer cette surcharge est de fournir des valeurs par défaut aux arguments **les plus à droite** de la fonction.

⇒Dans ce cas, les arguments **les plus à droite** qui ne sont pas initialisés explicitement lors de l'appel le sont avec leur valeur par défaut.

```
1  int *t = new int[20];
2  for (size_t i = 0; i < 20; i++) {
3    t[i] = 2 * i;
4  }
5  AfficheTableau(t, 20);                                  // Ok.
6  AfficheTableau(t, 20, " - ");                           // Ok.
7  AfficheTableau(t, 20, " | ", "> ");                     // Ok.
8  AfficheTableau(t, 20, " | ", "< ", " >");               // Ok.
9  AfficheTableau(t, 20, " | ", "< ", " >", false);        // Ok.
10 AfficheTableau(t, 20, false);                           // Erreur.
11 AfficheTableau(t, 20,,,, false);                        // Erreur.
```

Polymorphisme et **valeurs par défaut**

Une autre façon de gérer cette surcharge est de fournir des valeurs par défaut aux arguments **les plus à droite** de la fonction.

Dés lors qu'un argument possède une valeur par défaut, il n'est plus possible de spécifier d'argument sans valeur par défaut.
La déclaration suivante est **interdite** :

```
1 void AfficheTableau(int *tab, size_t deb = 0, size_t fin);
```

Polymorphisme et **valeurs par défaut**

Une autre façon de gérer cette surcharge est de fournir des valeurs par défaut aux arguments **les plus à droite** de la fonction.

Les arguments par défaut doivent apparaître lors de la déclaration de la fonction. Ils ne doivent plus apparaître dans l'implémentation si la fonction est déclarée au préalable.

```
1  void AfficheTableau(int *tab, size_t n,
2                      const string &sep = ", ",
3                      const string &delim1 = "[",
4                      const string &delim2 = "]",
5                      bool newline = true);
6  ...
7  void AfficheTableau(int *tab, size_t n,
8                      const string &sep, const string &delim1,
9                      const string &delim2, bool newline) {
10                        ...
11 }
```

Valeurs par défaut dans une méthode/un contructeur

À l'instar des fonctions, il est possible de fournir des valeurs par défaut aux arguments des méthodes d'une classe ou de son constructeur.

Valeurs par défaut dans une méthode/un contructeur

À l'instar des fonctions, il est possible de fournir des valeurs par défaut aux arguments des méthodes d'une classe ou de son constructeur.

```
1  class MaClasse {
2    ...
5    private:
6      type_1 attr1;
7      type_2 attr2;
8      ...
11   public:
12     ...
15     MaClasse(type_1 attr1 = val1, type_2 attr2 = val2);
16     // remplace la definition de:
17     //   MaClasse();
18     //   MaClasse(type_1 attr1);
19     //   MaClasse(type_1 attr1, type_2 attr2);
28     ...
29 };
```

Valeurs par défaut dans une méthode/un contructeur

À l'instar des fonctions, il est possible de fournir des valeurs par défaut aux arguments des méthodes d'une classe ou de son constructeur.

```
1  class MaClasse {
2    ...
5    private:
6      type_1 attr1;
7      type_2 attr2;
8      ...
11   public:
12     ...
22     type_r maMethode(type_a attra, type_b attrb = valb);
23     // remplace la definition de:
24     //   type_r maMethode(type_a attra);
25     //   type_r maMethode(type_a attra, type_b attrb);
28     ...
29 };
```

Valeurs par défaut dans une méthode/un contructeur

À l'instar des fonctions, il est possible de fournir des valeurs par défaut aux arguments des méthodes d'une classe ou de son constructeur.

De même, dés lors qu'un argument possède une valeur par défaut, il n'est plus possible de spécifier d'argument sans valeur par défaut. La déclaration suivante est **interdite** :

```
1  class MaClasse {
2    ...
11   public:
12     ...
22     type_r maMethode(type_a attra = vala, type_b attrb);
28     ...
29 };
```

Valeurs par défaut dans une méthode/un contructeur

À l'instar des fonctions, il est possible de fournir des valeurs par défaut aux arguments des méthodes d'une classe ou de son constructeur.

De la même manière, les valeurs par défaut ne doivent apparaître que lors des déclarations si l'implémentation est faire ultérieurement.

```
1  #include "maClasse.h"
2  ...
3  MaClasse::MaClasse(type_1 attr1, type_2 attr2):
4     attr1(attr1), attr2(attr2) {
5     ...
6  }
7  ...
8  type_r MaClasse::maMethode(type_a attra, type_b attrb) {
9     ...
10 }
11 ...
```

Plan du cours

Oui c'est possible !

Comme en `C`, il est possible de définir une fonction variadique (ayant un nombre *a priori* inconnu de variables, et dont le type n'est éventuellement connu que lors de l'appel).

$\Rightarrow$ Cela peut se faire en utilisant la librairie standard `cstdarg`.

Remarque

C'est ce qui est utilisé en `C` pour déclarer les fonctions `printf`, `fprintf`, `scanf`, ...

Comment faire ?

La signature doit d'abord définir le ou les paramètres obligatoires (au moins un), puis symboliser les paramètres optionnels en utilisant une ellipse « `...` ».

Exemple

```
#include <cstdarg>
double somme(size_t nb, ...);
```

La récupération et le traitement des arguments optionnels se fait *via* les macros `va_start`, `va_end`, `va_arg`, `va_copy` et le type `va_list` définis dans l'entête `<cstdarg>`.

Explication des macros de <cstdarg>

`va_list` Ce type permet de créer une variable de parcours de la liste des arguments optionnels.

`va_start` Cette macro permet d'initialiser la variable de parcours de la liste des arguments de la fonction à partir du dernier argument obligatoire[8] de la fonction (il permet de fournir le point de départ de la lecture des arguments en mémoire).

`va_end` Cette macro permet de libérer la variable de parcours de la liste des arguments optionnels.

`va_arg` Cette macro permet de récupérer le prochain argument pointé par la variable de parcours de la liste et de spécifier le type attendu pour cette variable.

`va_copy` Cette macro permet de dupliquer la variable de parcours afin de faire un second traitement sur les arguments optionnels.

8. Cet argument ne peut être ni une référence, ni un tableau (pointeur), ni un type défini par une fonction.

Exemple simple

```
1  #include <iostream>
2  using namespace std:
3  double somme(size_t nb, ...) {
4    double som = 0.;
5    va_list varg; // variables de parcours des parametres
6    va_start(varg, nb); // Positionnement sur la variable nb.
7    for (size_t i = 0; i < nb ; i++) {
8      // va_arg recupere l'argument suivant varg dans la liste
9      // et met a jour varg. Le second argument de va_arg
10     // indique que celui-ci est de type double
11     som += va_arg(varg, double);
12   }
13   va_end(varg); // Fin du parcours
14   return som;
15 }
16
17 int main(int argc, char** argv) {
18   cout << somme(4, 1.2, .2, -3, 5.4) << endl;
19   cout << somme(7, 1.2, .2, -3, 5.4, -5, 0.6, 0.1) << endl;
20   return 0;
21 }
```

Comment s'en passer... ?

L'intérêt est plus limité en C++.

En effet, la syntaxe de ce langage offre d'une part la possibilité de **surcharger** des opérateurs et d'autre part un mécanisme de **chaînage** par le renvoi de références sur des variables/instances.

Ainsi, il est possible de s'affranchir de cette contrainte (*e.g.*, opérateurs << et >>, ...)

Exemple

```
1  #include <iostream>
2
3  class Somme {
4    private:
5      double som;
6    public:
7      Somme():som(0) { }
8      Somme &operator<<(double v) { som += v; return *this; }
9      operator double() const { return som; }
10 };
11
12 int main (int argc, char** argv) {
13   Somme s1, s2;
14   s1 << 1.2 << .2 << -3 << 5.4
15   s2 << 1.2 << .2 << -3 << 5.4 << -5 << 0.6 << 0.1;
16   std::cout << double(s1) << std::endl
17             << double(s2) << std::endl;
18   return 0;
19 }
```

Exemple avancé

- Parfois, cela peut tout de même être pratique de définir de telles implémentations.

Exemple avancé

```
1  #include <cmath>
2
3  double ecartType(size_t nb, ...) {
4    double som = 0., moy;
5    va_list varg, varg_copy;
6    va_start(varg, nb);
7    va_copy(varg_copy, varg); // Copie de varg vers varg_copy.
8    for (size_t i = 0; i < nb ; i++) {
9      som += va_arg(varg, double);
10   }
11   va_end(varg);
12   moy = som / nb;
13   som = 0.;
14   for (size_t i = 0; i < nb ; i++) {
15     double v = va_arg(varg_copy, double);
16     som += (v - moy) * (v - moy);
17   }
18   va_end(varg_copy);
19   return std::sqrt(som / nb);
20 }
```

Exemple avancé

- Parfois, cela peut tout de même être pratique de définir de telles implémentations.
- Il n'est pas possible de définir des méthodes variadiques.

Plan du cours

Le mot clé `static`

Les attributs et méthodes décrites dans une classe sont propres à chaque instance de la dite classe.

Il est cependant possible de définir des attributs et méthodes qui soient détachées des instances en utilisant le mot clé `static`.

⇒ existe dans la classe, même en l'absence d'instance.

⇒ moyen de communication entre toutes les instances.

- Le mot clé `static` ne doit apparaître que dans la signature.
- Les attributs statiques sont à définir comme attributs de la classe lors de l'implémentation.
- Les méthodes statiques sont à définir comme les méthodes classiques.

Exemple

L'exemple ci-dessous permet d'une part d'associer un identifiant unique à chaque instance de la classe et de connaître à tout moment le nombre d'instances en cours pour cette classe.

maClasse.h

```
1  class MaClasse {
2    private:
3      const size_t id;
4      static size_t cpt;
5      static size_t nb;
6    public:
7      MaClasse();
8      MaClasse(const MaClasse &);
9      ~MaClasse();
10     size_t getId() const;
11     static size_t Combien();
12     bool EstDerniereInstanceCree() const;
13 };
```

Exemple

L'exemple ci-dessous permet d'une part d'associer un identifiant unique à chaque instance de la classe et de connaître à tout moment le nombre d'instances en cours pour cette classe.

maClasse.cpp

```
1  #include "maClasse.h"
2
3  size_t MaClasse::cpt = 0;
4  size_t MaClasse::nb = 0;
5
6  size_t MaClasse::getId() const { return id; }
7  size_t MaClasse::Combien() { return nb; }
8
9  MaClasse::MaClasse(): id(++cpt) { nb++; }
10 MaClasse::MaClasse(const MaClasse &m): id(++cpt) { nb++; }
11 MaClasse::~MaClasse() { nb--; }
12 bool MaClasse::EstDerniereInstanceCree() const {
13    return (id == cpt);
14 }
```

Plan du cours

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     cout << "Plus(5, 3)  = " << Plus(5, 3)  << endl;
     cout << "Moins(5, 3) = " << Moins(5, 3) << endl;
     cout << "Mult(5, 3)  = " << Mult(5, 3)  << endl;
     cout << "Div(5, 3)   = " << Div(5, 3)   << endl;
     cout << "Inverse(5)  = " << Inv(5)      << endl;

     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction




     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction
     fct = &Plus;       cout << "fct(5, 3) = " << (*fct)(5, 3) << endl;



     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction
     fct = Plus;         cout << "fct(5, 3) = " << fct(5, 3) << endl;



     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction
     fct = Plus;         cout << "fct(5, 3) = " << fct(5, 3) << endl;
     fct = Moins;        cout << "fct(5, 3) = " << fct(5, 3) << endl;



     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction
     fct = Plus;         cout << "fct(5, 3) = " << fct(5, 3) << endl;
     fct = Moins;        cout << "fct(5, 3) = " << fct(5, 3) << endl;
     fct = Mult;         cout << "fct(5, 3) = " << fct(5, 3) << endl;


     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction
     fct = Plus;         cout << "fct(5, 3) = " << fct(5, 3) << endl;
     fct = Moins;        cout << "fct(5, 3) = " << fct(5, 3) << endl;
     // fct = Mult;      cout << "fct(5, 3) = " << fct(5, 3) << endl; // Erreur


     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction
     fct = Plus;         cout << "fct(5, 3) = " << fct(5, 3) << endl;
     fct = Moins;        cout << "fct(5, 3) = " << fct(5, 3) << endl;
     // fct = Mult;      cout << "fct(5, 3) = " << fct(5, 3) << endl; // Erreur
     fct = Div;          cout << "fct(5, 3) = " << fct(5, 3) << endl;

     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction
     fct = Plus;         cout << "fct(5, 3) = " << fct(5, 3) << endl;
     fct = Moins;        cout << "fct(5, 3) = " << fct(5, 3) << endl;
     // fct = Mult;      cout << "fct(5, 3) = " << fct(5, 3) << endl; // Erreur
     // fct = Div;       cout << "fct(5, 3) = " << fct(5, 3) << endl; // Erreur

     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction
     fct = Plus;         cout << "fct(5, 3) = " << fct(5, 3) << endl;
     fct = Moins;        cout << "fct(5, 3) = " << fct(5, 3) << endl;
     // fct = Mult;      cout << "fct(5, 3) = " << fct(5, 3) << endl; // Erreur
     // fct = Div;       cout << "fct(5, 3) = " << fct(5, 3) << endl; // Erreur
     fct = Inverse;      cout << "fct(5)    = " << fct(5)    << endl;
     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction
     fct = Plus;         cout << "fct(5, 3) = " << fct(5, 3) << endl;
     fct = Moins;        cout << "fct(5, 3) = " << fct(5, 3) << endl;
     // fct = Mult;      cout << "fct(5, 3) = " << fct(5, 3) << endl; // Erreur
     // fct = Div;       cout << "fct(5, 3) = " << fct(5, 3) << endl; // Erreur
     // fct = Inverse;   cout << "fct(5)    = " << fct(5)    << endl; // Erreur
     return 0;
}
```

Pointeur vers une fonction

Un pointeur vers une fonction est une variable de type pointeur qui a la particularité d'avoir la même signature que la fonction pointée.

Exemple

```
#include <iostream>
using namespace std;
double Plus(double, double);  // Déclaration de la fonction Plus
double Moins(double, double); // Déclaration de la fonction Moins
double Mult(double, float);   // Déclaration de la fonction Mult
float  Div(double, double);   // Déclaration de la fonction Div
double Inverse(double);       // Déclaration de la fonction Inverse
     .
     .
     .
int main(int argc, char** argv) {
     double (*fct)(double, double); // déclaration d'un pointeur sur fonction
     fct = Plus;         cout << "fct(5, 3) = " << fct(5, 3) << endl;
     fct = Moins;        cout << "fct(5, 3) = " << fct(5, 3) << endl;
     // fct = Mult;      cout << "fct(5, 3) = " << fct(5, 3) << endl; // Erreur
     // fct = Div;       cout << "fct(5, 3) = " << fct(5, 3) << endl; // Erreur
     // fct = Inverse;   cout << "fct(5)    = " << fct(5)    << endl; // Erreur
     return 0;
}
```

Fonction en argument

Il est possible de passer une fonction comme argument d'une autre fonction.

Exemple

```
#include <iostream>

int PlusCinq(int a) { return a + 5; }
int Neg(int a) { return -a; }
int Aff(int a) { std::cout << a << " "; return a; }
int* AppliqueFct(int* tab, size_t n, int (*f)(int)) {
     for(size_t i = 0; i < n; i++) { tab[i] = f(tab[i]); }
     return tableau;
}
int main(int argc, char** argv) {
     int t[] = { 1, 2, 3, 4, 5 };
     std::cout << "[ ";
     AppliqueFct(AppliqueFct(AppliqueFct(t, 5, Neg), 5, PlusCinq), 5, Aff);
     std::cout << std::endl;      // Affichera : "4 3 2 1 0"
     return 0;
}
```

Fonction en argument

Il est possible de passer une fonction comme argument d'une autre fonction.

Il est également possible d'utiliser la commande `typedef` pour simplifier l'écriture de la signature des fonctions...

Exemple

```
#include <iostream>
typedef int (*Fct)(int); // Le type 'Fct' est un pointeur de fonction
int PlusCinq(int a) { return a + 5; }
int Neg(int a) { return -a; }
int Aff(int a) { std::cout << a << " "; return a; }
int* AppliqueFct(int* tab, size_t n, Fct f) {
    for(size_t i = 0; i < n; i++) { tab[i] = f(tab[i]); }
    return tableau;
}
int main(int argc, char** argv) {
    int t[] = { 1, 2, 3, 4, 5 };
    std::cout << "[ ";
    AppliqueFct(AppliqueFct(AppliqueFct(t, 5, Neg), 5, PlusCinq), 5, Aff);
    std::cout << std::endl;      // Affichera : "4 3 2 1 0"
    return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une. . .

méthode privée : ceci n'est pas possible en dehors de la classe.

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
    public:



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









    return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une...

méthode privée : ceci n'est pas possible en dehors de la classe.

méthode statique :

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
    public:
        static void MethodeStatique() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }


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




    return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une...

méthode privée : ceci n'est pas possible en dehors de la classe.

méthode statique : vu comme une fonction encapsulée dans une classe.

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
     public:
          static void MethodeStatique() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }


};
int main(int argc, char** argv) {
     void (*mth_s)() = TestPtr::MethodeStatique;
     mth_s(); // Affiche 'MethodeStatique:5'




     return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une...

méthode privée : ceci n'est pas possible en dehors de la classe.

méthode statique : vu comme une fonction encapsulée dans une classe.

méthode d'instance :

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
     public:
          static void MethodeStatique() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void Methode() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void MethodeConst() const { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
};
int main(int argc, char** argv) {
     void (*mth_s)() = TestPtr::MethodeStatique;
     mth_s(); // Affiche 'MethodeStatique:5'




     return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une...

méthode privée : ceci n'est pas possible en dehors de la classe.

méthode statique : vu comme une fonction encapsulée dans une classe.

méthode d'instance : doit être rattachée à une instance.

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
     public:
          static void MethodeStatique() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void Methode() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void MethodeConst() const { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
};
int main(int argc, char** argv) {
     void (*mth_s)() = TestPtr::MethodeStatique;
     mth_s(); // Affiche 'MethodeStatique:5'

     void (*mth)() = TestPtr::Methode; // Ne compile pas (2 messages d'erreurs)



     return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une...

méthode privée : ceci n'est pas possible en dehors de la classe.

méthode statique : vu comme une fonction encapsulée dans une classe.

méthode d'instance : doit être rattachée à une instance.

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
     public:
          static void MethodeStatique() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void Methode() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void MethodeConst() const { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
};
int main(int argc, char** argv) {
     void (*mth_s)() = TestPtr::MethodeStatique;
     mth_s(); // Affiche 'MethodeStatique:5'

     void (TestPtr::*mth)() = TestPtr::Methode; // Ne compile toujours pas (1 message d'erreur)



     return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une...

méthode privée : ceci n'est pas possible en dehors de la classe.

méthode statique : vu comme une fonction encapsulée dans une classe.

méthode d'instance : doit être rattachée à une instance.

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
     public:
          static void MethodeStatique() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void Methode() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void MethodeConst() const { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
};
int main(int argc, char** argv) {
     void (*mth_s)() = TestPtr::MethodeStatique;
     mth_s(); // Affiche 'MethodeStatique:5'

     void (TestPtr::*mth)() = &TestPtr::Methode; // Ok



     return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une...

méthode privée : ceci n'est pas possible en dehors de la classe.

méthode statique : vu comme une fonction encapsulée dans une classe.

méthode d'instance : doit être rattachée à une instance.

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
     public:
          static void MethodeStatique() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void Methode() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void MethodeConst() const { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
};
int main(int argc, char** argv) {
     void (*mth_s)() = TestPtr::MethodeStatique;
     mth_s(); // Affiche 'MethodeStatique:5'
     TestPtr p; // Besoin d'une instance pour tester les pointeurs
     void (TestPtr::*mth)() = &TestPtr::Methode; // Ok



     return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une...

méthode privée : ceci n'est pas possible en dehors de la classe.

méthode statique : vu comme une fonction encapsulée dans une classe.

méthode d'instance : doit être rattachée à une instance.

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
     public:
          static void MethodeStatique() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void Methode() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void MethodeConst() const { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
};
int main(int argc, char** argv) {
     void (*mth_s)() = TestPtr::MethodeStatique;
     mth_s(); // Affiche 'MethodeStatique:5'
     TestPtr p; // Besoin d'une instance pour tester les pointeurs
     void (TestPtr::*mth)() = &TestPtr::Methode; // Ok
     (p.*mth)(); // Affiche 'Methode:6'


     return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une...

méthode privée : ceci n'est pas possible en dehors de la classe.

méthode statique : vu comme une fonction encapsulée dans une classe.

méthode d'instance : doit être rattachée à une instance.

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
     public:
          static void MethodeStatique() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void Methode() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void MethodeConst() const { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
};
int main(int argc, char** argv) {
     void (*mth_s)() = TestPtr::MethodeStatique;
     mth_s(); // Affiche 'MethodeStatique:5'
     TestPtr p; // Besoin d'une instance pour tester les pointeurs
     void (TestPtr::*mth)() = &TestPtr::Methode; // Ok
     (p.*mth)(); // Affiche 'Methode:6'
     void (TestPtr::*mth_c)() const = &TestPtr::MethodeConst;
     (p.*mth_c)(); // Affiche 'MethodeConst:7'
     return 0;
}
```

Pointeur vers une méthode

Pointeurs sur une...

méthode privée : ceci n'est pas possible en dehors de la classe.

méthode statique : vu comme une fonction encapsulée dans une classe.

méthode d'instance : doit être rattachée à une instance.

constructeur/destructeur : impossible.

Exemple

```
#include <iostream>
using namespace std;
class TestPtr {
     public:
          static void MethodeStatique() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void Methode() { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
          void MethodeConst() const { cout << __FUNCTION__ << ":" << __LINE__ << endl; }
};
int main(int argc, char** argv) {
     void (*mth_s)() = TestPtr::MethodeStatique;
     mth_s(); // Affiche 'MethodeStatique:5'
     TestPtr p; // Besoin d'une instance pour tester les pointeurs
     void (TestPtr::*mth)() = &TestPtr::Methode; // Ok
     (p.*mth)(); // Affiche 'Methode:6'
     void (TestPtr::*mth_c)() const = &TestPtr::MethodeConst;
     (p.*mth_c)(); // Affiche 'MethodeConst:7'
     return 0;
}
```

Exemple

tabReels.h

```
1  #ifndef __TABREELS_H__
2  #define __TABREELS_H__
3
4  #include <iostream>
5
6  class TableauReels {
7    private:
8      double *tab;
9      size_t n;
10     static void TriRapide_aux(TableauReels &, size_t, size_t);
11     static double zero();
12   public:
13     typedef double (*Rnd)();
14     TableauReels(size_t n = 0, Rnd generateur = zero);
15     TableauReels(const TableauReels &);
16     ~TableauReels();
17     TableauReels &operator=(const TableauReels &);
18     double operator[](size_t) const;
19     double &operator[](size_t);
20     size_t length() const;
21     bool esttrie();
22     static void InitTableau(TableauReels &, Rnd);
23     static void TriBulle(TableauReels &);
24     static void TriRapide(TableauReels &);
25     static void TriParTas(TableauReels &);
26 };
27
28 std::ostream &operator<<(std::ostream &, const TableauReels &);
29 #endif
```

Exemple

tabReels.cpp

```
1  #include "tabReels.h"
2
3  using namespace std;
4
5  TableauReels::TableauReels(size_t n, TableauReels::Rnd generateur)
6    : tab(new double[n]), n(n) {
7    for (size_t i = 0; i < n; i++) { tab[i] = generateur(); }
8  }
9
10 TableauReels::TableauReels(const TableauReels &t)
11   : tab(new double[t.n]), n(t.n) {
12   for (size_t i = 0; i < n; i++) { tab[i] = t.tab[i]; }
13 }
14
15 TableauReels::~TableauReels() {
16   delete [] tab;
17 }
18
19 TableauReels &TableauReels::operator=(const TableauReels &t) {
20   if (this != &t) {
21     if (t.n != n) {
22       delete [] tab;
23       n = t.n;
24       tab = new double[n];
25     }
26     for (size_t i = 0; i < n; i++) { tab[i] = t.tab[i]; }
27   }
28   return *this;
29 }
```

Exemple

tabReels.cpp

```
31 double TableauReels::operator[](size_t i) const {
32   return tab[i];
33 }
34 double &TableauReels::operator[](size_t i) {
35   return tab[i];
36 }
37
38 size_t TableauReels::length() const {
39   return n;
40 }
41
42 bool TableauReels::esttrie() {
43   bool ok = true;
44   size_t i = 0;
45   while (ok && (++i < n)) {
46     ok = (tab[i - 1] <= tab[i]);
47   }
48   return ok;
49 }
50
51 void ech(double &a, double &b) {
52   double tmp = a;
53   a = b;
54   b = tmp;
55 }
56
57 double TableauReels::zero() {
58   return 0;
59 }
```

Exemple

tabReels.cpp

```
61 void TableauReels::TriRapide_aux(TableauReels &t, size_t deb, size_t fin) {
62   if (deb >= fin) return;
63   size_t d = deb, f = fin, p;
64   p = (d + f--) / 2;
65   while (d < f) {
66     while ((d < p) && (t[d] <= t[p])) d++;
67     while ((p < f) && (t[p] <= t[f])) f--;
68     ech(t[d], t[f]);
69     if (p == d) p = f++;
70     else if (p == f) p = d--;
71     d++;
72     f--;
73   }
74   TriRapide_aux(t, deb, p);
75   TriRapide_aux(t, p + 1, fin);
76 }
77
78 void TableauReels::TriRapide(TableauReels &t) {
79   TriRapide_aux(t, 0, t.length());
80 }
```

Exemple

tabReels.cpp

```
82  void TableauReels::TriBulle(TableauReels &t) {
83    size_t n = t.length();
84    while (n--) {
85      for (size_t i = 0; i < n; i++) {
86        if (t[i] > t[i + 1]) {
87          ech(t[i], t[i + 1]);
88        }
89      }
90    }
91  }
92
93  void TableauReels::TriParTas(TableauReels &t) {
94    size_t n = t.length();
95
96    size_t deb, fin;
97    double tmp;
98    for (size_t i = 1 ; i < n; i++) {
99      tmp = t[i];
100     deb = i;
101     fin = (deb - 1) / 2;
102     while (deb && (t[fin] < tmp)) {
103       t[deb] = t[fin];
104       deb = fin;
105       fin = (deb - 1) / 2;
106     }
107     t[deb] = tmp;
108   }
```

Exemple

tabReels.cpp

```
110   for (size_t i = n - 1 ; i > 0 ; i--) {
111     tmp = t[i];
112     t[i] = t[0];
113     fin = 0;
114     if (i == 1) {
115       deb = (size_t) -1;
116     } else {
117       deb = 1;
118     }
119     if ((i > 2) && (t[2] > t[1])) {
120       deb = 2;
121     }
122     while ((deb != (size_t) -1) && (tmp < t[deb])) {
123       t[fin] = t[deb];
124       fin = deb;
125       deb = 2 * fin + 1;
126       if ((deb + 1 <= i - 1) && (t[deb] < t[deb + 1])) {
127         deb++;
128       }
129       if (deb > i - 1) {
130         deb = (size_t) -1;
131       }
132     }
133     t[fin] = tmp;
134   }
135 }
```

Exemple

tabReels.cpp

```
137 void TableauReels::InitTableau(TableauReels &t, TableauReels::Rnd generateur) {
138   for (size_t i = 0; i < t.length(); i++) {
139     t[i] = generateur();
140   }
141 }
142
143 ostream &operator<<(ostream &os, const TableauReels &t) {
144   os << "[";
145   for (size_t i = 0; i < t.length(); i++) {
146     os << t[i] << (i != t.length() - 1 ? ", " : "");
147   }
148   return os << "]";
149 }
```

Exemple

testTri.h

```
1  #ifndef __TESTTRI_H__
2  #define __TESTTRI_H__
3
4  #include "tabReels.h"
5  #include <iostream>
6  #include <cstdlib>
7
8  typedef void (*AlgoTri)(TableauReels &);
9  double Random();
10 double RandInt_1_10();
11
12 double temps(TableauReels &t, AlgoTri algo);
13 void Test(AlgoTri algo,
14           TableauReels::Rnd gen,
15           TableauReels &t,
16           unsigned int nb = 1000);
17
18 #endif
```

Exemple

testTri.cpp

```
1  #include "testTri.h"
2
3  using namespace std;
4
5  double Random() {
6    return drand48();
7  }
8  double RandInt_1_10() {
9    return (lrand48() % 10) + 1;
10 }
11
12 double temps(TableauReels &t, AlgoTri algo) {
13   clock_t debut = clock();
14   algo(t);
15   clock_t duree = clock() - debut;
16   if (!t.esttrie()) {
17     cout << "t n'est pas trie : " << t << endl;
18     exit(1);
19   }
20   return double(duree) / CLOCKS_PER_SEC;
21 }
```

Exemple

testTri.cpp

```
23 void Test(AlgoTri algo, TableauReels::Rnd gen, TableauReels &t, unsigned int nb)
24 {
25   cout << "*** Temps pour " << nb << " executions" << endl;
26   cout << "    Remplissage du tableau "
27        << "(taille " << t.length() << ") avec le generateur "
28        << (gen == &Random ? "Random"
29                           : (gen == &RandInt_1_10 ? "RandInt_1_10" : "???"))
30        << endl
31        << "    Tri avec l'algorithme "
32        << (algo == &TableauReels::TriBulle ?
33              "TriBulle"
34            : (algo == &TableauReels::TriRapide ?
35                 "TriRapide"
36               : (algo == &TableauReels::TriParTas ?
37                    "TriParTas"
38                  : "???")))
39        << endl;
40   double tps = 0;
41   for (unsigned int i = 0; i < nb; i++) {
42     TableauReels::InitTableau(t, gen);
43     tps += temps(t, algo);
44   }
45   cout << "    => " << tps << " secondes." << endl;
46 }
```

Exemple

testTri.cpp

```
48 int main(int argc, char** argv) {
49   TableauReels tab(10000);
50   Test(TableauReels::TriBulle,  Random, tab);
51   Test(TableauReels::TriRapide, Random, tab);
52   Test(TableauReels::TriParTas, Random, tab);
53   Test(TableauReels::TriBulle,  RandInt_1_10, tab);
54   Test(TableauReels::TriRapide, RandInt_1_10, tab);
55   Test(TableauReels::TriParTas, RandInt_1_10, tab);
56   return 0;
57 }
```

Compilation avec :

```
for ((i = 0; i < 4; i++)); do
  g++ -Wall -ansi -pedantic -O$i testTri.cpp -c
  g++ -Wall -ansi -pedantic -O$i tabReels.cpp -c
  g++ -o testTri$i tabReels.o testTri.o
  ./testTri$i > output$i.txt
done
```

Exemple

output0.txt

```
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriBulle
    => 928.743 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriRapide
    => 2.84354 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriParTas
    => 3.75085 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriBulle
    => 898.356 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriRapide
    => 25.4763 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriParTas
    => 3.12094 secondes.
```

Exemple

output1.txt

```
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriBulle
    => 222.41 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriRapide
    => 0.933287 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriParTas
    => 0.948886 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriBulle
    => 220.514 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriRapide
    => 3.4004 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriParTas
    => 0.799281 secondes.
```

Exemple

output2.txt

```
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriBulle
    => 179.185 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriRapide
    => 0.899457 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriParTas
    => 0.934227 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriBulle
    => 176.815 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriRapide
    => 2.86941 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriParTas
    => 0.745059 secondes.
```

Exemple

output3.txt

```
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriBulle
    => 175.361 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriRapide
    => 0.90027 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur Random
    Tri avec l'algorithme TriParTas
    => 0.940439 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriBulle
    => 173.175 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriRapide
    => 2.87075 secondes.
*** Temps pour 1000 executions
    Remplissage du tableau (taille 10000) avec le generateur RandInt_1_10
    Tri avec l'algorithme TriParTas
    => 0.74907 secondes.
```

That's All, Folks !

Merci de votre attention.

Vous pouvez rentrer chez vous.

C'est fini.

Au revoir.

Bon courage.

Ceci n'est pas un test ophtalmologique,

mais si vous arrivez à lire jusqu'au bout...

...c'est que vous avez du temps.

Profitez-en pour programmer alors !!!