Qu'est-ce que la «Programmation» ?

D'une manière générale,
l'objectif de la programmation est de
permettre l'**automatisation** d'un certain nombre de tâches,
à l'aide de machines particulières:

les **automates programmables**

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Automates programmables

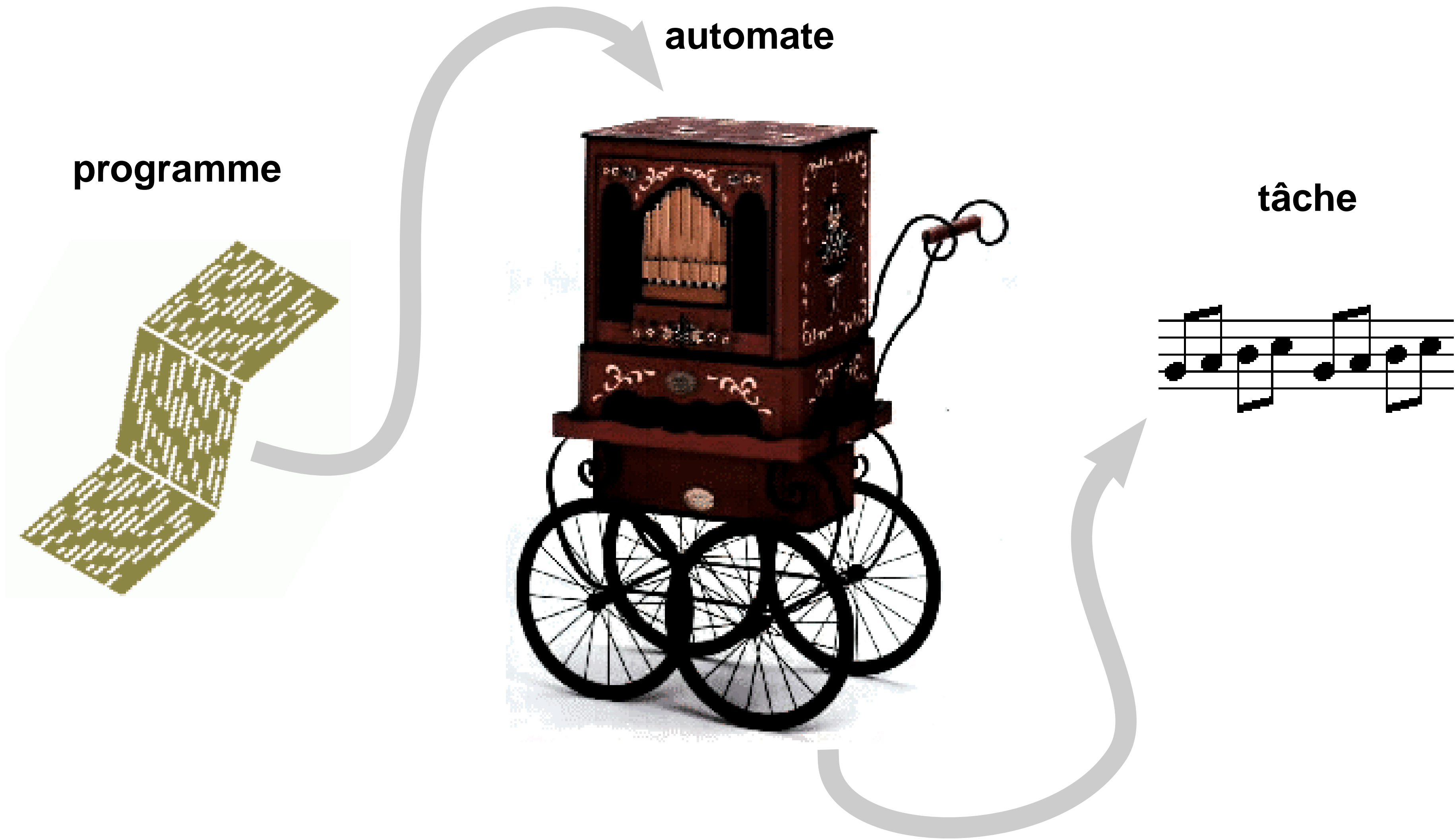
☞ Un **automate** est un dispositif capable d'assurer, sans intervention humaine, un enchaînement d'opérations, correspondant à la réalisation d'une tâche donnée.
Comme exemple d'automates, on peut citer:
la montre, le réveil-matin, le «ramasse-quilles» (du bowling).

☞ L'automate est dit «**programmable**» lorsque l'enchaînement d'opérations effectuées peut être modifié à volonté, pour permettre un changement de la tâche à réaliser, ou un changement dans la manière de réaliser cette tâche. Dans ce cas, la **description de la tâche** à accomplir se fait par le biais d'un **programme**, c'est-à-dire une séquence d'instructions et de données susceptibles d'être traitées
(i.e. «comprises» et «exécutées») par l'automate.

Comme exemples d'automates programmables, citons:

le métier à tisser Jacquard,
l'orgue de barbarie,
...,
et naturellement l'ordinateur.

Exemple d'automate programmable



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Programmer c’est...

En résumé, programmer c’est donc **décomposer** la tâche à automatiser sous la forme d’une **séquence d’instructions et de données** adaptées à l’automate utilisé.

Dès lors, voyons quelles sont ces instructions et données «adaptées», dans le cas où l’automate programmable est un ordinateur.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Instructions et langage machine

Nous l'avons vu, un ordinateur est, en schématisant à l'extrême, constitué:

- d'un [**micro**]**processeur**, capable d'exécuter (réaliser) un jeu donné d'opérations élémentaires.
- d'une **mémoire centrale**, dans laquelle sont stockées les données en cours de traitement, ainsi que le programme lui-même;
- de bus, ports d'entrées-sorties et périphériques;

Le **jeu d'instructions** (~ langage) que l'ordinateur est capable de traiter est donc tout naturellement déterminé par le processeur.

☞ Les instructions comprises par un processeur sont appelées les ***instructions machine*** de ce processeur.

☞ Le langage de programmation qui utilise ces instructions est appelé le ***langage machine***.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Langage machine: format interne

☞Pour que les opérations et les données manipulées soient compréhensibles par le processeur, elles doivents être exprimées dans le seul format qu'il peut prendre en compte:

le *format interne*,

qui est [presque[1]] toujours un ***format binaire***.

⇨ Ce format n'utilise que deux *symboles élémentaires* (généralement «*0*» et «*1*») appelés «*bits*»[2].

☞ Mettre en correspondance la représentation externe des opérations et des données avec leur représentation sous la forme d'une séquence de bits s'appelle le **codage**.

⇨ Le codage permet donc à l'ordinateur de manipuler des données de nature et de type divers sous la forme d'une représentation unique.

---

1. Il y eu quelques tentatives pour construire des machines supportant d'autre formats, en particulier en URSS le format ternaire. Si au niveau électrique il y a un intérêt certain à disposer ainsi de trois états (actif, inactif, neutre), cela ne concerne pas le niveau logique. L'avantage de la représentation binaire est qu'elle est techniquement facile à réaliser (au moyen de bistables), que les opérations fondamentales sont relativement simple à effectuer sous forme de circuits logiques et que de plus, comme l'a démontré Shannon, tous les calculs logiques et arithmétiques peuvent être réalisés à partir de l'arithmétique binaire.
2. Contraction de l'expression anglaise «binary digit».

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

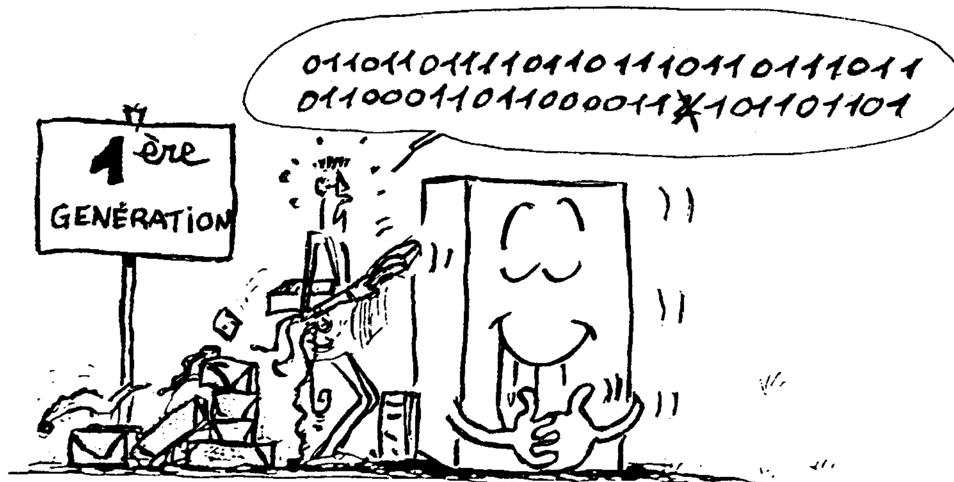
Les générations de langage

Au cours du temps, différents types de langages de programmation firent leur apparition,
et furent regroupés [entre autres] en **générations**

Bien qu'il s'agisse d'une classification différente de celle des machines, les générations de langages sont, du moins pour les premières, liées sur le plan chronologique aux générations de machines, et aux performances de leurs composants

1ère Génération

Au début était le **langage** [de la] **machine**.



Dans le cas des ordinateurs de premières générations, la mémoire n’est pas encore utilisée pour le stockage du programme.

Celui-ci est placé sur des cartes et/ou rubans perforés, voir même totalement **volatile**,[1] i.e. entré à la main à l’aide de panneaux de commutateurs.

On l’imagine facilement, la production des programmes était fastidieuse, et leur relecture et modification presque impossible.
Par ailleurs, les programmes écrits pour une machine ne pouvaient pas être exécutés sur une autre.

⟹ Remarquons que la technique des cartes ou rubans perforés a perduré jusqu’à nos jours (dans des secteurs très ciblés): c’est un moyen de stockage de l’information passablement souple, qui offre surtout une très **grande fiabilité en milieu hostile**.

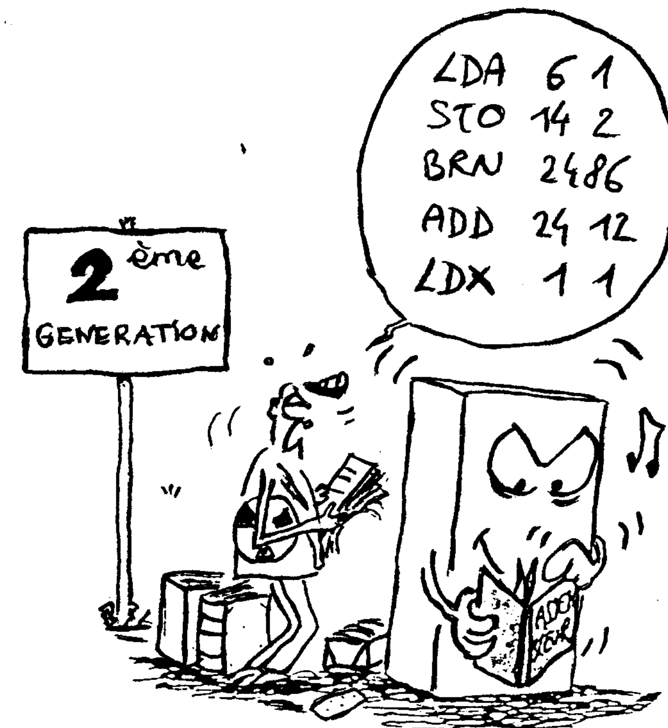
1. Notamment en ce qui concerne l’amorçage du système, et la connexion/prise en charge du système de lecture de cartes.

2ème génération (1)

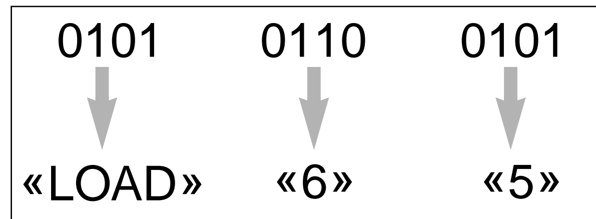
Puis vint le **langage d'assemblage**.

Afin de rendre les programmes plus faciles à écrire et à corriger, on a rapidement songé à remplacer les séquences de bits par des symboles, les instructions machines étant codées par des *mnémoniques* et les données par les caractères alphanumériques associés.

Le nouveau langage ainsi produit s'appelle un **langage d'assemblage**, ou **langage assembleur**.



Exemple



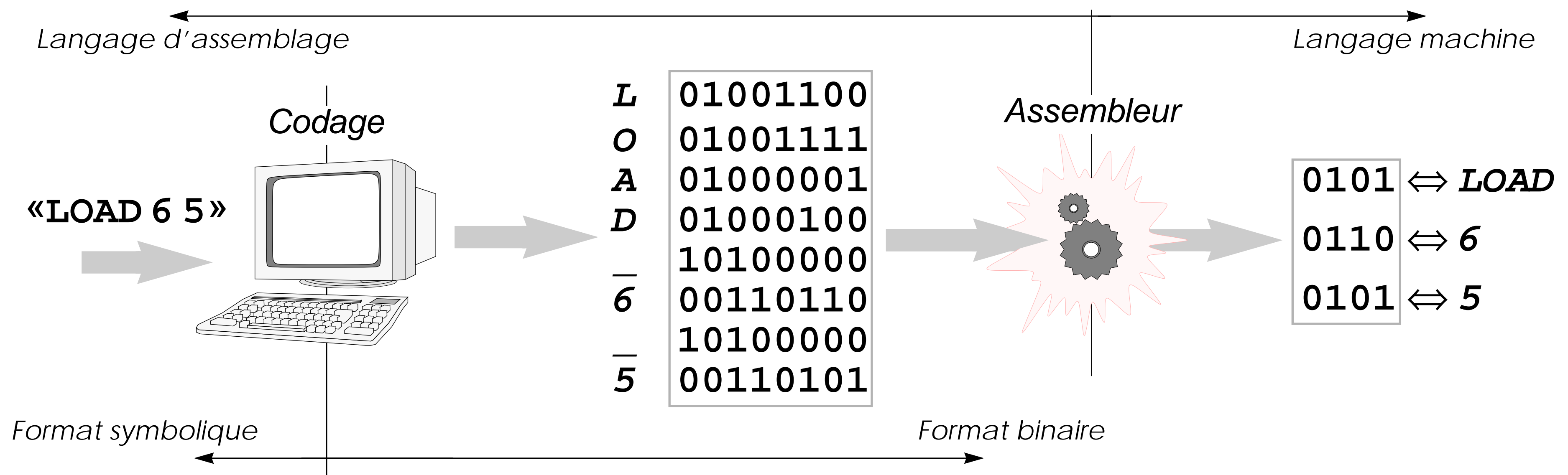
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

2ème génération (2)

☞ L'apparition de périphériques de saisie (clavier) et de visualisation (écran) permet une écriture – respectivement lecture – plus ergonomique des séquences de 0 et de 1, sous la forme de séquences alphanumériques.

☞ Avec l'idée de *Von Neumann* d'enregistrer dans la mémoire le programme à exécuter ainsi que ses données, il devient également envisageable de faire subir à ce programme des traitements avant son exécution (prétraitements).

⇨ La représentation en mémoire du programme en langage d'assemblage devient ainsi elle-même une donnée, pouvant être traitée par un programme spécifique nommé «assembleur», dont le rôle est sa traduction en langage machine.



Exemple de langage d'assemblage (1)

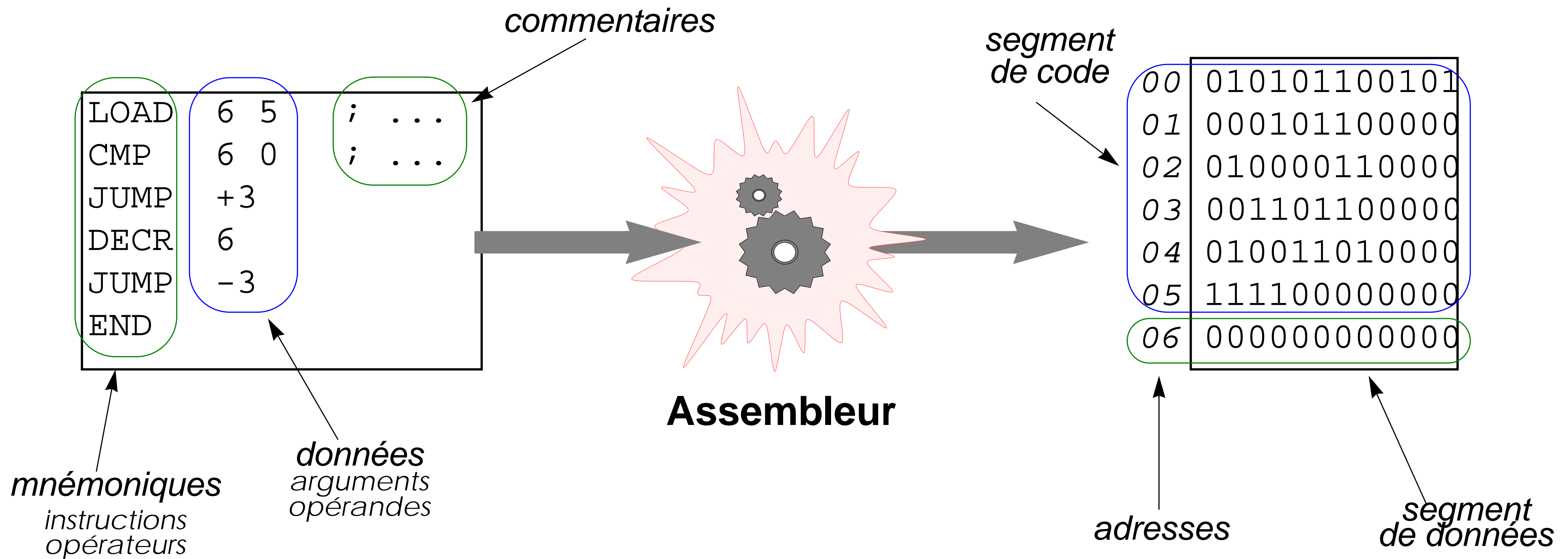
Exemple de langage d'assemblage: *(machine à mots de 4 bits et alignement sur 12)*

Mnémonique	*Code*	*Syntaxe*	*Définition*
NOP	0000	NOP	Opération neutre (ne fait rien)
CMP	0001	CMP [*adresse*] [*valeur*]	Compare la valeur stockée à l'adresse [*addresse*] avec la valeur [*valeur*]. L'exécution se poursuit à l'adresse courante + 1 en cas d'égalité, + 2 sinon.
DECR	0011	DECR [*adresse*]	Décrémente de 1 la valeur stockée en [*addresse*]
JUMP	0100	JUMP [*saut*]	L'exécution se poursuit à l'adresse courante + [*saut*]
LOAD	0101	LOAD [*adresse*] [*valeur*]	Charge la valeur [*valeur*] à l'adresse [*adresse*]
END	1111	END	Fin d'exécution

Codage des données: *(codage sur 4 bits, en complément à 2)*

Données	*Code machine*
-3	1101
0	0000
2	0010
3	0011
5	0101
6	0110

Exemple de langage d'assemblage (2)

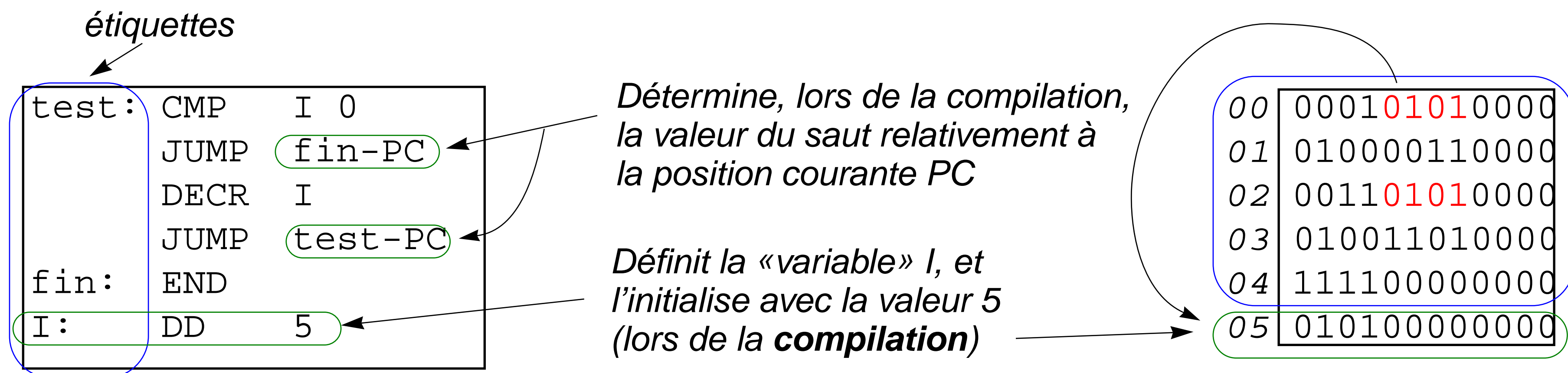


Exemple de langage d'assemblage (3)

☞Un langage d'assemblage tel que celui décrit précédemment est appelé **langage d'assemblage pur**, c'est à dire qu'il y a biunivocité entre les instructions machine et les mnémoniques (le code source en langage d'assemblage comportent le même nombre d'élements (d'instructions) que le code machine résultant).

Les tout premiers assembleurs étaient *purs*, mais cette «propriété» disparut rapidement, avec l'apparition des langages offrant des *macros-commandes* (pour définir une seule fois des portions de codes fréquentes) ainsi que des *pseudo-instructions* (réservation & initialisation de mémoire, chargement de modules séparés,...).

La traduction effectuée par les assembleurs correspondants a permis de rendre le code **portable** (i.e. réutilisable, moyennant son assemblage, sur une autre machine), en masquant la couche matérielle (**abstraction de la machine**).



3ème génération

Dans la lancée apparurent les **langages évolués**.

Le but est de fournir au programmeur la possibilité d'utiliser des **instructions** et des **structures de données de** plus **haut niveau**, afin de rendre plus facile l'écriture des programme et d'améliorer la productivité.

⇨ Ces langages sont plus accessibles, plus proches de notre manière de **penser et de conceptualiser les problèmes.**

Les langages évolués sont nombreux, et il ne cesse d'en apparaître de nouveaux. Citons à titre d'exemple: BASIC, Fortran, Cobol, C, Pascal, Modula-2, Ada, OccamII, Portal, ...

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

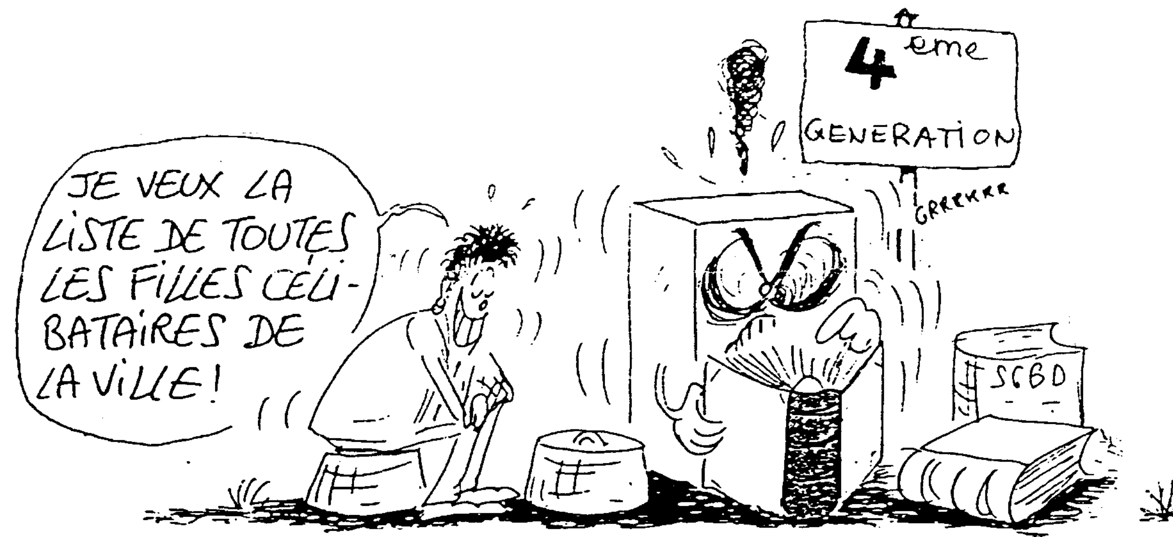
Exemples de langages évolués

Exemples de langage de programmation de haut niveau:

«Vieux» BASIC	*ANSI C*
`1   REM Ceci est un commentaire`	`/* Ceci est un commentaire */`
`4   LET N = 5`	`int n = 5;`
`5   IF (N = 0) THEN GOTO 10`	`while (n != 0) do --n;`
`6   ELSE LET N = N-1`	
`7   GOTO 5`	
`10  END`	

4ème génération

Parallèlement aux langages évolués, des langages encore **plus spécialisés**, voir des **méta-langages**, ont fait leur apparition.



On peut par exemple citer les langages d'intelligence artificielle (Lisp, Prolog), les langages objets (Smalltalk, Eiffel), certains langages de gestion (L4G), ...

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Langage de programmation (1)

Cependant, si nous utilisons des intructions sophistiquées, comment les rendre compréhensible pour l'ordinateur qui doit les exécuter ?

☞ Nous l'avons vu, une solution est de disposer d'un programme capable de **transformer** une séquence d'instructions de haut niveau (aussi appelée **code source**) en une séquence d'instructions machine (aussi appelée **code objet**[1] ou **binaire**).

Les programmes qui convertissent un programme quelconque écrit dans un langage source en un programme écrit dans un langage cible sont appelés ***traducteurs***.

Selon ses caractéristiques, un programme *traducteur* est appelé un ***assembleur***, un ***compilateur***[2] ou un ***interpréteur***; l'ensemble des instructions et données de plus haut niveau que le traducteur est capable de traiter constitue un **langage de programmation.**

1. Ce qui n'a rien à voir avec la Programmation [Orientée] Objet !
2. Equivalent d'un assembleur, mais fonctionnant avec un langage source évolué (non assembleur).

Langage de programmation (2)

Un **langage de programmation** est
donc un moyen formel permettant de décrire
des traitements (i.e. des tâches à effectuer) sous la forme
de programmes (i.e. de séquences d'instructions et de données
de haut niveau, c'est-à-dire compréhensibles par le programmeur)
et pour lequel il existe un traducteur permettant l'exécution
effective des programmes par un ordinateur.

Les aspects syntaxiques (règles d'écriture des programmes)
et sémantiques (définition des instructions) d'un langage
de programmation doivent être spécifiés de manière précise,
généralement dans un manuel de référence.

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Interpréteur v.s. Compilateur (1)

Il est important de bien comprendre la différence entre traduction effectuée par un assembleur ou un compilateur et celle réalisée un interpréteur.

⇨ Les **compilateurs** et **assembleurs** ***traduisent*** tous deux les programmes ***dans leur ensemble***: tout le programme doit être fourni au compilateur pour la traduction. Une fois cette traduction effectuée, son résultat (code objet) peut être soumis au processeur pour traitement.

Un langage de programmation pour lequel un compilateur est disponible est appelé un **langage compilé**.

⇨ Les **interpréteurs** ***traduisent*** les programmes ***instruction par instruction***, et soumettent immédiatement chaque instruction traduite au processeur, pour exécution.

Un langage de programmation pour lequel un interpréteur est disponible est appelé un **langage interpreté**.

Remarquons que pour certains langages (par exemple Lisp), il existe à la fois des compilateurs et des interpréteurs.

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Interpréteur v.s. Compilateur (2)

Naturellement, chacune de ces techniques possède des avantages et des inconvénients, qui la rend plus ou moins adaptée suivant le contexte:

☞ De manière générale, on peut dire que les langages interprétés sont bien adaptés pour le **développement rapide et le prototypage**:

- le cycle de test est plus court qu'avec les langages compilés,
- il est souvent possible de modifier/rectifier le programme en cours d'exécution (test),
- et ces langages offrent généralement une **plus grande liberté d'écriture**.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Interpréteur v.s. Compilateur (3)

☞ A l’inverse, les langages compilés sont à utiliser de préférence pour les **réalisations opérationnelles**, ou les **programmes de grande envergure**:

- Les programmes obtenus sont **plus efficaces**:

  → d’une part, le compilateur peut effectuer des optimisations plus facilement que l’interpréteur, puisqu’il possède une visibilité globale sur le programme,

  → et d’autre part, l’effort de traduction n’est fait qu’une seule fois, qui plus est en prétraitement.

- Par ailleurs, la visibilité globale offerte au compilateur, allié à une structuration plus rigoureuse et un typage plus ou moins contraint, permet une **meilleur détection des erreurs**, lors de la compilation.

Remarquons finalement que la compilation permet la diffusion du programme sous sa forme opérationnelle, sans imposer pour autant sa diffusion sous forme conceptuelle (lisible et compréhensible par un humain)

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Variété des applications

L'utilisation de l'informatique dans les 3 types d'applications – calcul scientifique, gestion d'informations et commande de processus – à conduit à la production d'une grande diversité de programmes:

- **Petits utilitaires**, «drivers» (librairie de fonctions permettant de piloter un matériel), et autre mini-programmes.

  →Effort: ~ 1 à 2 *personnes* × *mois*, de ~1000 à 10'000 lignes de code (source), équipes d'une à deux personnes.

- **Petits logiciels** («maileur», gestionnaire de fichiers, agenda électronique, ...)

  →Effort: ~ 3 à 12 *personnes* × *mois*, de 1'000 à 50'000 lignes, petites équipes de développement (1 à 5 personnes).

- **Progiciels**, logiciels complet (traitement de textes, commande de central téléphonique,...)

  →Effort: ~ 1 à 10 *personnes* × *ans*, de 10'000 à 1'000'000 lignes, équipes de développement moyennes (de 5 à 20 personnes)

- **Gros systèmes** (aéronautique, systèmes d'exploitation, systèmes experts, ...)

  →Effort: ~ plus de 100 *personnes* × *ans*, plusieurs centaines de millions de lignes de code, grosses équipes de développement (plusieurs milliers de personnes).

Le Langage C++

Développé dans les laboratoires d'*AT&T Bell* au début des années 1980 par *Bjarne Stroustrup*, le langage C++ est un langage:

⇨ **à *typage fort***,
⇨ ***compilé*** (impératif),
⇨ **et *orienté objet*** (POO[1]).

Schématiquement:

C++ = C + typage fort + objets (classes)

Avantages	***Défauts***
• Programmes exempts de bogues «syntaxiques», et [un peu] plus robuste, grâce au typage fort • Applications efficaces grâce à la compilation • Compilateur disponible sur pratiquement toutes les plate-formes et documentation abondante, grâce à sa large diffusion.	• Effets indésirables et comportement peu intuitif, conséquence de la production automatique de code • Syntaxe parfois lourde et peu naturelle • Le langage ne définit pas de techniques de récupération automatique de mémoire (*garbage collector*), de multiprogrammation ou de programmation distribuée.

1. Acronyme de *Programmation Orientée Objet*. Remarquons toutefois que C++ n'est pas purement objet (comme le sont par exemple *Eiffel* ou *Smalltak*) mais est un langage hybride: on peut très bien programmer en C++ sans pour autant programmer par objets, c'est d'ailleurs ce qui sera fait pendant tout le premier semestre de ce cours.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Compilation en C++

☞ Pour pouvoir être compilé en une séquence binaire exécutable, le code source doit fournir au compilateur un «*point d'entrée*».

Par convention, ce point d'entrée est en C++ une fonction intitulée **`main`**:

Compilateur C++

fichier source → fichier exécutable

commande: `c++ hello.cc -o hello`

hello.cc

```
#include <iostream>
void main()
{
   cout << "Hello World !"
        << endl;
}
```

hello

```
01001001010001011101010010
10101010100101001010010100
00001010101000000100010000
00010010011101010100100111
10010000000000000100101011
...
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Structure générale d'un programme C++ (1)

Le programme C++ le plus simple,
... et le plus inutile (puisqu'il ne «fait rien») est:

```
void main ()
{
}
```

Ce programme va progressivement être enrichi... et en particulier,
on peut y ajouter des commentaires, qui en C++ peuvent revêtir 2 formes différentes:

- **COMMENTAIRES ORIENTÉS LIGNE:** *de la forme «*`// ...`*»*

```
// Exemple de commentaires orienté ligne,
// c'est-à-dire commentaire délimité à gauche ('//')
// seulement (implicitement délimité à droite par
// le caractère de fin de ligne)
```

Structure générale d'un programme C++ (2)

- **COMMENTAIRES ORIENTÉS BLOC:** *de la forme* « `/* ... */` »

```
/* Exemple de commentaire orienté bloc,
   c'est-à-dire commentaire délimité à gauche ET à
   droite (le commentaire peut s'étendre sur plusieurs
   lignes) */
```

Une version commentée du programme précédent pourrait donc être:

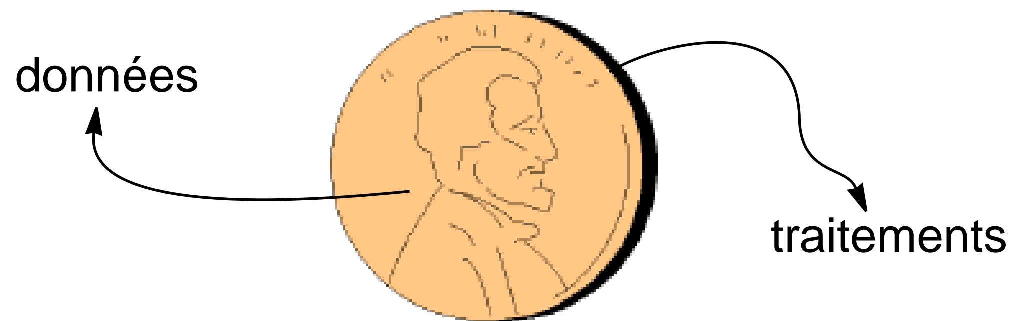
```
// Exemple de petit programme C++ ne faisant rien
// (Le formattage n'est bien sur là
// que pour faciliter la lecture)
void main ()
{
   /*
      corps du programme
      (à définir par la suite)
   */
}
```

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Données et traitements

Comme dans tout langage de programmation évolué,
C++ offre la possibilité de
⇨ définir des **données** (aspect statique du langage), et
⇨ mettre en œuvre des **traitements** (aspect dynamique).

☞ L'aspect orienté objet de C++ va ensuite permettre d'intégrer données et traitements dans une structure unique, les **objets**.
Cet aspect du langage sera développé au cours du second semestre.



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Exemple de programme C++ (1)

On veut réaliser un programme qui résout une équation du second degré:

$$x^2 + \boldsymbol{b}x + \boldsymbol{c} = 0$$

Pour $\boldsymbol{b}$ et $\boldsymbol{c}$ fixés, les solutions réelles d'une telle équation sont:

$$\begin{cases} \dfrac{-b \pm \sqrt{\Delta}}{2} & \text{si } \Delta > 0 \\[2ex] \dfrac{-b}{2} & \text{si } \Delta = 0 \\[2ex] \varnothing & \text{sinon} \end{cases}$$

avec $\Delta = b^2 - 4c$

Exemple de programme C++ (2)

```
#include <cmath>
#include <iostream>

void main()
{
   float b(0.0);
   float c(0.0);
   float delta(0.0);
   cin >> b >> c;
   delta = b*b - 4*c;
   if (delta < 0.0){
      cout << "Pas de solutions réelles !" << endl;
   }
   else if (delta == 0.0) {
      cout << "Une solution unique: " << -b/2.0 << endl;
   }
   else {
      cout  << "Deux solutions: [" << (-b-sqrt(delta))/2.0
            << ", " << (-b+sqrt(delta))/2.0 << ']' << endl;
   }
}
```

données
traitements
structures de contrôle

Données

```
#include <cmath>
#include <iostream>

void main()
{
   float b(0.0);
   float c(0.0);
   float delta(0.0);
   cin >> b >> c;
   delta = b*b - 4*c;
   if (delta < 0.0){
      cout << "Pas de solutions réelles !" << endl;
   }
   else if (delta == 0.0) {
      cout << "Une solution unique: " << -b/2.0 << endl;
   }
   else {
      cout  << "Deux solutions: [" << (-b-sqrt(delta))/2.0
            << ", " << (-b+sqrt(delta))/2.0 << ']' << endl;
   }
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Variables: définition

Pour pouvoir être utilisée dans un programme C++, une donnée doit être associée à une ***variable***, c'est-à-dire un élément informatique qui sera manipulé par le programme. Une variable est décrite à l'aide de 3 caractéristiques:

- ⇨ Sa **valeur littérale**:
  qui permet de définir sa *valeur*. Par exemple, si la donnée est un nombre, sa valeur littérale pourra être (selon les conventions de représentation):
  123, -18, 3.1415, 2e-13, , 0x3A5, ...

- ⇨ Son **identificateur**:
  qui est le *nom* par lequel la donnée est désignée dans le programme.

- ⇨ Son **type**:
  qui correspond à une *classification* de la données, par exemple en fonction des opérations qui peuvent lui être appliquées; cette caractéristique sera développée plus en détails dans la suite du cours, et généralisée en la notion de classe.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Déclaration et initialisation de variables

Dans les langages fortement typés comme C++, la création d'une variable se fait à l'aide d'une ***déclaration***, et l'association effective d'une valeur à la variable créée se fait à l'aide d'une ***initialisation***.

- ☞ Les *déclarations* doivent obligatoirement être faites **avant** toute utilisation de la variable;
- ☞ les *initialisations* doivent impérativement être faites **avant** toute utilisation de la variable[2].

En C++, la syntaxe de la déclaration d'une variable est:

```
<type> <identificateur> ;
```

Où *type* est l'identificateur du **type** de la variable déclarée, et *identificateur* est l'identificateur de la variable.

Exemple:

```
int i;
float exemple_de_variable
```

2. Le compilateur n'interdit pas l'utilisation d'une variable non initialisée, mais il exclut l'utilisation de variables non déclarées.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Déclaration

Les principaux ***types élémentaires*** définis en C++ sont:

`bool` : les valeurs logiques «vraie» (true) et «fausse» (false)

`int` : les nombres entiers

`float` : les nombres réels

`char` : les caractères (alphanumériques)

Un identificateur de variable ou de constante peut être n'importe quelle séquence[3] composée de lettres, de chiffres ou du caractère souligné («_»), et débutant par une lettre ou le caractère souligné.

Exemple:

```
x
b3
TailleMax
nb_etudiants
```

3. Remarquons toutefois qu'il existe certaines séquences «interdites», correspondant à des *mots réservés* du langage.

Valeurs littérales

- Valeurs littérales de type entier: `1, 12, ...`
- Valeurs littérales de type réel: `0.0, 1.23, ...`
  Remarquons que:
  `0.123` peut aussi s'écrire `.123`
  `12.3e4` correspond à $12.3 \cdot 10^{4}$ (soit 1230000)
  `12.e-4` correspond à $12.3 \cdot 10^{-4}$ (soit 0.00123)
- Valeurs littérales de type caractères: `'a', '!', ...`
  Remarques:
  le caractère «`'`» est représenté par `'\''`
  le caractère «\» est représenté par `'\\'`[4]

Remarque:

la valeur littérale 0 est une valeur d'initialisation qui peut être affectée à une variable de n'importe quel type élémentaire.

4. Le caractère *backslash* «\» est ainsi appelé un caractère *d'échappement*.

Déclarations et initialisation

Il est possible, en C++, d'affecter une valeur à une variable au moment de sa déclaration.

La syntaxe de la **déclaration-initialisation** est:

*<type> <identificateur>***(***<valeur>***);**

Où *valeur* est n'importe quel **littéral** ou **expression**[5] du type indiqué.

Exemple:

```
int i(2);
float exemple_de_variable(3.1415);
char c('a');
int j(i);
float x(2.0*exemple_de_variable);
```

5. La notion d'expression sera définie en détail dans la suite du cours.

Variables v.s. constantes

En C++, la nature **modifiable** (variable) ou **non modifiable** (constante) d'une donnée peut être définie lors de la déclaration-initialisation, au moyen du mot réservé `const`

Exemple:

```
const int j(2);
```

La constante `j` de type entier est déclarée, et initialisée à la valeur 2.
Cette valeur *ne pourra plus être modifiée par le programme*, plus exactement toute tentative de modification sera rejetée par le compilateur, qui signalera une erreur.

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Déclaration et initialisation de variables

En résumé, en C++ une donnée est donc un élément informatique caractérisé par:

- ⇨ son **type**
- ⇨ son **identificateur**

} définis lors de la **déclaration**

- ⇨ sa **valeur** → définie lors de **l'initialisation**

Exemple:

```
int i(3);
```

déclare une donnée de type entier, d'identificateur `i`
et l'initialise à la valeur 3

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Premiers exemples de traitements

```
#include <cmath>
#include <iostream>

void main()
{
   float b(0.0);
   float c(0.0);
   float delta(0.0);
   cin >> b >> c;
   delta = b*b - 4*c;
   if (delta < 0.0){
      cout << "Pas de solutions réelles !" << endl;
   }
   else if (delta == 0.0) {
      cout << "Une solution unique: " << -b/2.0 << endl;
   }
   else {
      cout  << "Deux solutions: [" << (-b-sqrt(delta))/2.0
            << ", " << (-b+sqrt(delta))/2.0 << ']' << endl;
   }
}
```

Une instruction simple: l’affectation

☞ L’instruction d’**affectation** consiste à **attribuer** (*affecter*) une **valeur** à une **variable**.

En C++, la syntaxe d’une affectation est:

```
<identificateur> = <valeur> ;
```

où `valeur` est une valeur **littéral** ou une **expression** de même type que la variable référencée par `identificateur`.

Exemple:

```
i = 3;
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Interagir avec l'extérieur: entrées/sorties

Les **instructions d'entrées/sorties** (E/S) permettent les interactions du programme avec son environnement (extérieur).

Les mécanismes d'E/S doivent en particulier donner au programmeur la possibilité:

De prendre en compte, au sein du programme, des données issues de l'extérieur, comme des informations saisies au clavier ou lues dans des fichiers.

⇨ requiert des fonctions spécifiques, appelées **fonctions d'entrées**, permettant d'associer des informations externes aux variables définies dans le programme.

D'afficher des données à l'écran

⇨ requiert des fonctions spécifiques, appelées **fonctions de sorties**, permettant de visualiser des valeurs, en spécifiant éventuellement un format d'affichage.

De sauvegarder les résultats produits pour d'éventuelles réutilisations.

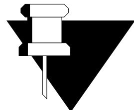
⇨ requiert des fonctions spécifiques, appelées **fonctions de lecture/écriture**, permettant de stocker les données produites par les programmes dans des *fichiers*, et de récupérer ces données par la suite (notion de *persistence* des données).

Instructions d'entrées/sorties (1)

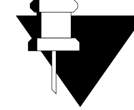
Instructions élémentaires pour l'affichage à l'écran:

cout << *<expression*$_1$*>* **<<** *<expression*$_2$*>* **<< ... <<** *<expression*$_n$*>* **;**

affiche à l'écran[6] les valeurs des expressions *<expression*$_i$*>*.



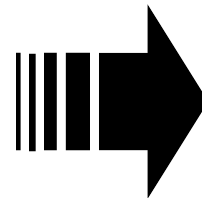
Les chaînes de caractères doivent être indiquées entre guillemets anglais « " », tandis que les caractères simples sont eux indiqués entre apostrophes « ' ».



Les valeurs affichées ne sont, pas délimitées par défaut (pas d'espaces séparateurs). Les délimiteurs sont donc à indiquer explicitement, par exemple par `' '` ou `" "`.

Exemple

```
int a(1);
cout  << '[' << -a << ", "
      << a << ']' << endl;
```



```
[-1, 1]
```

6. En réalité sur la *sortie standard* (c.f. plus tard dans le cours).

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Instructions d'entrées/sorties (2)

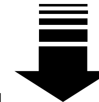
Exemple d'utilisation de `cout`:

Avec l'initialisation:

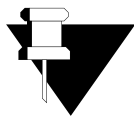
```
int i(2);
float x(3.14);
```

l'instruction:

```
cout << "=> " << 2*i+5 << ", " << x << endl;
```



produira à l'écran l'affichage suivant: `=> 9, 3.14`



Le manipulateur `endl` représente le *retour de ligne*.
Il permet donc de «passer à la ligne» suivante lors de l'affichage.

Instructions d’entrées/sorties (3)

Instructions élémentaires pour la lecture au clavier:

cin << <*var*$_1$> **<<** <*var*$_2$> **<< . . . <<** <*var*$_n$>**;**

permet à l’utilisateur de saisir au clavier[7] une liste de valeurs val_1, val_2, ..., val_n qui seront stockées dans les variables <*var*$_i$>.

Exemple

Avec l’initialisation:

```
int i;
double x;
```

l’instruction:

```
cin >> i >> x;
```

permettra de lire au clavier un entier suivi d’un réel, et affectera l’entier lu à la variable `i` et le réel à la variable `x`.

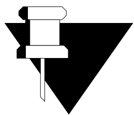
7. En réalité sur l’ *entrée standard* (c.f. plus tard dans le cours).

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Instructions d'entrées/sorties (4)

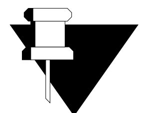
Exemple de programme:

```
#include <iostream>
void main()
{
   int i;
   double x;
   cout << "Valeurs pour i et x: " << flush;
   cin >> i >> x;
   cout << "=> " << i << ", " << x << endl;
}
```



Pour pouvoir utiliser les instructions d'E/S, il faut inclure, en début de programme, le fichier *header* `iostream`, au moyen de la directive:

`#include <iostream>`



Le manipulateur `flush` permet d'assurer l'affichage de la ligne courante, en l'absence de saut de ligne (il ordonne le vidage du tampon d'affichage)

Exemples d'expressions

```
#include <cmath>
#include <iostream>

void main()
{
   float b(0.0);
   float c(0.0);
   float delta(0.0);
   cin >> b >> c;
   delta = b*b - 4*c;
   if (delta < 0.0){
      cout << "Pas de solutions réelles !" << endl;
   }
   else if (delta == 0.0) {
      cout << "Une solution unique: " << -b/2.0 << endl;
   }
   else {
      cout  << "Deux solutions: [" << (-b-sqrt(delta))/2.0
            << ", " << (-b+sqrt(delta))/2.0 << ']' << endl;
   }
}
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Opérateurs et expressions (1)

☞ En plus des instructions, tout langage de programmation fournit des **opérateurs** permettant de manipuler les éléments prédéfinis.
Les définitions des opérateurs sont souvent étroitement associées au type des éléments sur lesquels ils sont capables d'opérer.

Ainsi, les ***opérateurs arithmétiques*** (`+, -, *, /, ...`) sont définis pour les types numériques (entiers et réels par exemple[8]), et les ***opérateurs logiques*** (`!`, `&&`, `||`) sont définis eux (en particulier) pour les types booléens.

Les **expressions** sont des séquences bien formées combinant (éventuellement à l'aide de parenthèses) des opérateurs et des arguments (qui peuvent être des valeurs littérales, des variables ou des expressions).

Par exemple: `(2*(13-3)/(1+4))` est une expression numérique bien formée, alors que `)3+)5*(-2` ne l'est pas.

8. En fait, et on le verra par la suite, il est également possible de définir les opérateurs arithmétiques pour des types non-élémentaires, comme par exemple les nombres complexes.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Opérateurs et expressions (2)

Les ***opérateurs arithmétiques*** sont (dans leur ordre de priorité d'évaluation):

`*`	multiplication
`/`	division
`%`	modulo
`+`	addition
`-`	soustraction

Le *modulo* est le reste de la division entière.

Ces opérateurs sont tous ***associatifs à gauche***

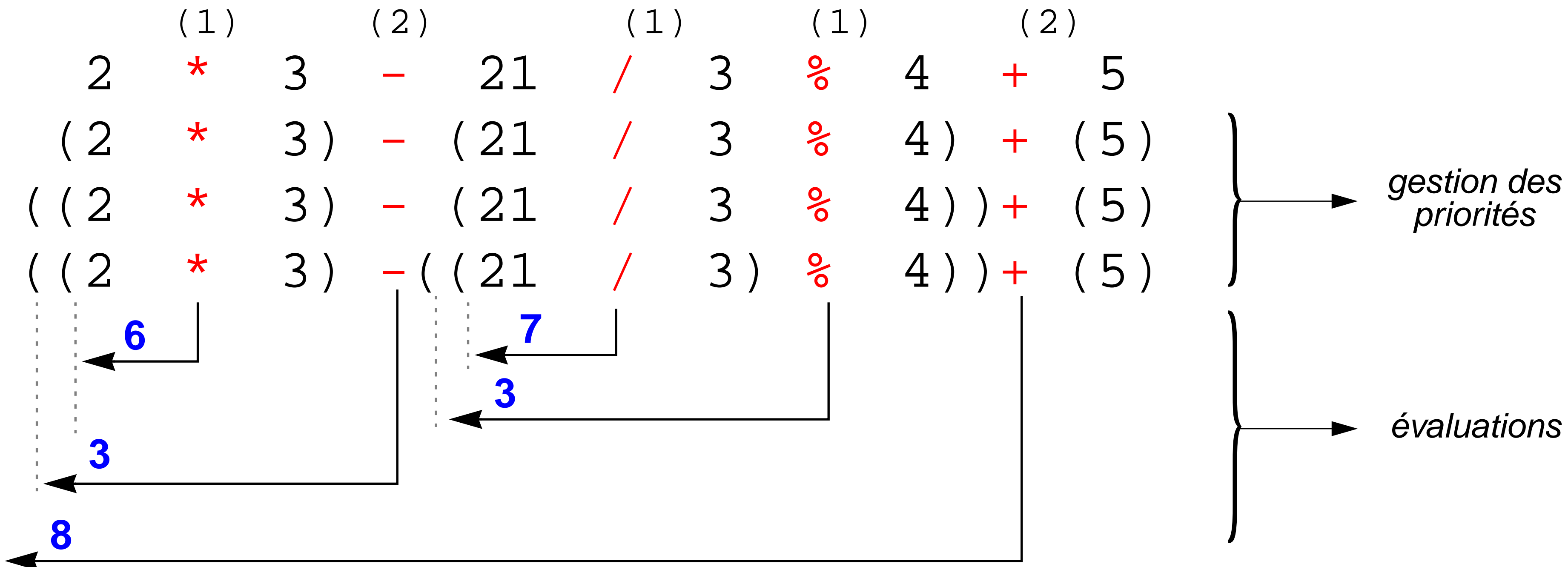
Du fait que les opérateurs ont non seulement une ***syntaxe***, mais aussi une ***définition opératoire***, une expression peut être **évaluée**, c'est-à-dire associée à la ***valeur littérale*** correspondant au ***résultat*** de l'expression.

Ainsi, l'évaluation de l'expression `(2*(13-3)/(1+4))` correspond à la valeur littérale de type entier `4`.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Opérateurs et expressions (3)

Fonctionnement de l'ordre de priorité des opérateurs:



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Opérateurs et expressions (4)

Exemple d'utilisation d'expressions pour l'affectation:

- `Z = (x+3) % y;`
- `Z = (3*x+y)/10;`
- `Z = (x = 3*y)+2;`

Remarque:

C++ (tout comme C) fournit un certain nombre de **notations abrégées** pour des affectations particulières:

Affectation	*Notation abrégée*
`x = x + y` idem pour {`-`,`*`,`/`,`%`}	`x += y`
`x = x + 1` idem pour {`-`}	`++ x`[9]

9. Ou `x++`, la différence entre les deux notations sera expliquée dans les mini-références.

Opérateurs et expressions (4)

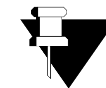
Remarques à propos de l'opérateur de division en C++:

➯ Si `a` **et** `b` sont *entiers*, `a/b` est le quotient de la division entière de `a` par `b`.

Exemple: `5/2 = 2`

et `a%b` est le reste de cette division.

Exemple: `5 % 2 = 1`



donc, si `a` et `b` sont *entiers*, on a:

`(a/b)*b+(a%b) = a`

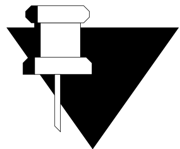
➯ Si `a` **et/ou** `b` est *réel*, `a/b` est le résultat de la division de `a` par `b`.

Exemple: `5.0/2 = 2.5`

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Expressions logiques

Au sens strict, une **expression logique** (ou ***condition***) est une **expression booléenne** (i.e. de type *booléen*), c'est-à-dire une expression qui peut avoit comme résultat : soit la valeur «vraie» (`true`) – dans ce cas on dit que la condition est *vérifiée* – soit la valeur «fausse» (`false`) – et dans ce cas, la condition n'est *pas vérifiée*.



En fait, la définition du langage C++ permet de considérer comme expression logique une expression de n'importe quel type, avec la convention suivante:
si l'évaluation de l'expression est une ***valeur nulle***, elle sera considérée comme une condition **fausse**, sinon elle sera considérée comme une condition **vraie**.

Les valeurs nulles sont[10]:

- les zéros numériques: `0` et `0.`
- la valeur booléenne `false`

Exemples d'expressions simples:

- `true`, `1`, `(2+2)`
  sont des expressions évaluées comme *vraies*.
- `false`, `(16%2)`, `0.0`
  sont des expressions évaluées comme *fausses*.

10. Il existe une valeur nulle supplémentaire: la valeur *void*; nous la verrons plus tard dans le cours, lorsque nous aborderons la notion de pointeurs.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Opérateurs de comparaison

Des conditions logiques plus complexes
peuvent être construites, à l’aide d’opérateurs spécifiques:
les **opérateurs de comparaison** et les **opérateurs logiques**.

Les **opérateurs de comparaisons** (également appelés ***opérateurs relationnels***)
sont (dans leur ordre de priorité d’évaluation): .

Opérateur	*Opération*
< <= > >=	strictement inférieur inférieur ou égal strictement supérieur supérieur ou égal
== !=	égalité différence (non-égalité)

Exemples de conditions correspondants à des expressions relationnelles:

```
(x >= y)
(x+y == 4)
((x > y) == (y > z))
```

Opérateurs logiques (1)

Les **opérateurs logiques** permettent de combiner plusieurs conditions entre-elles.

Opérateur	*Opération*
`!`	NON logique (négation)
`&&`	ET logique (conjonction)
`||`	OU logique

Toujours selon leur ordre de priorité d'évaluation.

Exemples de conditions complexes:

```
  • ((z != 0) && (2*(x-y)/z < 3))
≡ • ((z) && (2*(x-y)/z < 3))

  • ((x >= 0) || ((x*y > 0) && !(y*z > 0)))
```

≡ *indique 2 conditions logiquement équivalentes*

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Opérateurs logiques (2)

Les opérateurs logiques ET, OU et NON sont définis par les tables de vérités usuelles:

X	***Y***	***non X***	***x ET y***	***x OU y***
`v`	`v`	`f`	`v`	`v`
`v`	`f`	`f`	`f`	`v`
`f`	`v`	`v`	`f`	`v`
`f`	`f`	`v`	`f`	`f`

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Evaluation paresseuse (1)

☞ Les opérateurs logiques `&&` et `||` effectuent une **évaluation paresseuse** (*lazy evaluation*) de leur arguments:

⇨ l'évaluation des arguments se fait de la **gauche vers la droite**, et ne sont évalués que les arguments **strictement nécessaires** à la détermination de la valeur logique de l'expression.

Plus précisément:

- dans l'expression conditionnelle: (X_1 `&&` X_2 `&&` ... `&&` X_n),
  l'argument X_i n'est évalué que si les arguments X_j ($j<i$) sont tous *vrais:*
  l'évaluation se termine au **premier X_j** ***faux*** rencontré, auquel cas l'expression toute entière est *fausse* (si un tel X_j n'existe pas, l'expression est naturellement *vraie*)

- dans l'expression conditionnelle: (X_1 `||` X_2 `||` ... `||` X_n),
  l'argument X_i n'est évalué que si les arguments X_j ($j<i$) sont tous *faux:*
  l'évaluation se termine au **premier X_j** ***vrai*** rencontré, auquel cas l'expression toute entière est *vraie* (si un tel X_j n'existe pas, l'expression est naturellement *fausse*)

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Evaluation paresseuse (2)

La notion d'évaluation paresseuse est très utile pour construire des expressions logiques dans lesquelles l'évaluabilité de certains arguments peut être conditionnée par les arguments précédents dans l'expression.

Exemple:

```
(x != 0) && 4/x > 3)
```

Si l'évaluation n'était pas paresseuse, la détermination de l'expression conduirait à une erreur lorsque x vaut 0 (division par 0)[11]

```
(x <= 0 || log(x) == 4)
```

Dans ce cas également, une évaluation complète conduirait à une erreur lors du calcul de `log(x)` pour tout `x` inférieur ou égal à 0.

D'une manière générale, il est toujours bon d'économiser du temps

11. Notez que dans ce cas particulier, on pourrait contourner l'obstacle en testant l'expression équivalente:

```
((x>0) && (4 > 3*x)) || ((x<0) && (4 < 3*x))
```

Structures de contrôle (1)

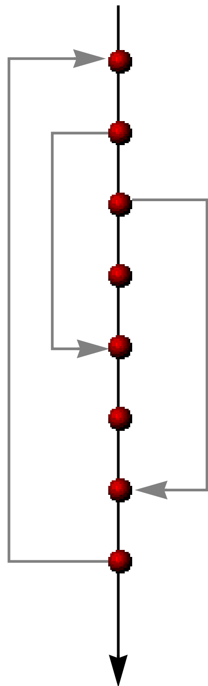
```
#include <cmath>
#include <iostream>

void main()
{
   float b(0.0);
   float c(0.0);
   float delta(0.0);
   cin >> b >> c;
   delta = b*b - 4*c;
   if (delta < 0.0){
      cout << "Pas de solutions réelles !" << endl;
   }
   else if (delta == 0.0) {
      cout << "Une solution unique: " << -b/2.0 << endl;
   }
   else {
      cout  << "Deux solutions: [" << (-b-sqrt(delta))/2.0
            << ", " << (-b+sqrt(delta))/2.0 << ']' << endl;
   }
}
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Structures de contrôle (2)

☞ Une ***structure de contrôle*** sert à modifier l'ordre linéaire normal d'exécution (i.e. le *flux d'exécution*) des instructions d'un programme.

Les principales **structures de contrôle** sont:



- Le *branchement* conditionnel: `if`
- La *sélection* à choix multiples: `switch`
- La *boucle*: `while`
- L'*itération*: `for`

Toutes ces structures vont de plus utiliser la notion [importante] de ***bloc d'instructions***.

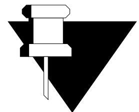
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Séquence ou bloc

Pour pouvoir être utilisée, les instructions d'un programmes C++ doivent être regroupées en entités[1] appelées **séquences d'instructions** ou ***blocs***.

Ces séquences sont explicitement délimitée par les accolades «{» et «}»:

```
{
   int tmp(a); // échange du contenu
   a = b;      // de deux variables
   b = tmp;
}
```



Par convention, on alignera les accolades l'une sous l'autre[2],
et on indentera la séquence d'instructions correspondant au bloc.

1. Parfois imbriquées les unes dans les autres
2. Il existe cependant de nombreux cas où il paraît judicieux de ne se plier à cette convention. Ainsi, dans la suite du cours elle ne sera parfois pas respectée, pour des raisons de présentation (l'espace disponible sur un transparent étant passablement restreint).

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Branchement conditionnel: `if` (1)

Le **branchement conditionel** est une structure de contrôle qui permet la ***mise en œuvre de traitements variables***, définit par des conditions d'applications spécifiques (i.e. le traitement *d'alternatives*).

La syntaxe générale d'un **branchement conditionnel** est:

if (*<condition>*) { *<instructions1: si condition vérifiée>* }
[**else** { *<instructions2: si condition non-vérifiée>* } *]*

la notation [. . .]
indique une partie optionnelle

L'expression *<condition>* est tout d'abord évaluée.
Si la condition est vérifiée, la séquence d'instructions *<instructions1:...>* est exécutée; sinon, c'est l'éventuelle séquence alternative *<instructions2:...>*, introduite par le mot réservé `else`, qui sera exécutée.

Exemple

```
if (x != 0) {cout << 1/x << endl;}
else {cout << "Erreur! (Infinity)" << endl;}
```

Branchement conditionnel: `if` (2)

Il est également possible d'enchaîner plusieurs conditions:

```
if (<condition 1>)
{
    <instructions 1: si condition 1 vérifiée>
}
else if (<condition 2>)
{
    <instructions 2: si condition 2 vérifiée>
}
...
else if (<condition N>)
{
    <instructions N: si condition N vérifiée>
}
[else
{
    <instructions par défaut>
}]
```

- L'expression *<condition 1>* est tout d'abord évaluée; si elle est vérifiée, le bloc *<instructions 1:...>* est exécuté, et les alternatives ne sont pas examinée.
- Sinon (1ère condition non vérifiée), c'est l'expression *<condition 2>* qui est évaluée, et ainsi de suite.
- Si aucune des N conditions n'est vérifiée, et que la partie optionnelle `else` est présente, c'est le bloc *<instructions par défaut>* associé qui sera finalement exécuté.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Sélection à choix multiples: `switch` (1)

La **sélection à choix multiples** permet parfois de remplacer avantageusement un enchaînement de branchements conditionnels, en permettant de sélectionner une instruction en fonction des valeurs possible d'une expression de type entier ou caractère.

La syntaxe générale d'une **sélection à choix multiples** est:

```
switch (<expression>)
{
    case <Constante_1> : <instructions_1> ;
    ...
    case <Constante_n> : <instructions_n>;
[   default : <instructions> ; ]
}
```

L'expression *<expression>* est évaluée, puis comparée successivement à chacune des valeurs *<Constante*$_i$*>*[3] introduites par le mot réservé `case`. En cas d'égalité, le bloc (ou l'instruction) associée est exécutée. Dans le cas où il n'y a aucune *<Constante*$_i$*>* égale à la valeur de l'expression, l'éventuel bloc introduit par le mot réservé `default` est exécuté.

3. Pour spécifier ces valeurs, on ne peut uniquement utiliser des expressions constituées de valeurs littérales ou de constantes (l'expression doit être évaluable lors de la compilation)

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Sélection à choix multiples: `switch` (2)

Exemple de sélection

```
switch (a+b)
{
   case 0:  a = b;    // exécuté uniquement lorsque
            break;    // (a+b) vaut 0
   case 2:
   case 3:  b = a;    // lorsque (a+b) vaut 2 ou 3
   case 4:
   case 5:  a = 0;    // lorsque (a+b) vaut 2,3,4 ou 5
            break;
   default: a = b = 0; // dans tous les autres cas.
}
```

Lorsque l'exécution des instructions d'un `case` est terminée, le contrôle ne passe pas à la fin du `switch`, mais continue l'exécution des instructions suivantes (même si leur valeur associée n'est pas égale à l'expression de sélection). Pour provoquer la terminaison de l'instruction `switch`, il est nécessaire de placer explicitement une instruction **`break`**.

![EPFL logo]

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Boucles: `while` (1)

Les **boucles** permettent la mise en œuvre répétitive d'un traitement, contrôlé *a posteriori* ou *a priori* par une condition de continuation.

La syntaxe générale d'une boucle avec condition de continuation *a posteriori* est:

```
do
{
    <actions>
} while (<condition>);
```

La séquence d'instructions *<actions>* est exécutée **tant que** l'expression de continuation *a posteriori* *<condition>* est vérifiée.

Exemple

```
do
{
   cout << "valeur de i (>=0): ";
   cin >> i;
   cout << "=> " << i << endl;
}
while (i<0);
```

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Boucles: `while` (2)

La syntaxe générale d'une boucle avec condition de continuation *a priori* est:

```
while (<condition>)
{
    <actions>
}
```

Tant que l'expression de continuation *a priori* *<condition>* est vérifiée,
la séquence d'instructions *<actions>* est exécutée.

Exemple

```
while (x>0)
{
   cout << x;
   x /= 2;
}
```

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

L'itération: `for` (1)

Les itérations permettent l'application itérative d'un traitement, contrôlée par une opération d'initialisation, une condition d'arrêt, et une opération de mise à jour des variables de contrôle de l'itération.

La syntaxe générale d'une **itération** est:

```
for (<initialisation> ; <condition> ; <mise à jour>)
{
    <actions>
}
```

Une itération `for` est équivalente à la boucle `while` suivante:

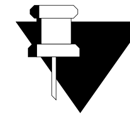
```
{
    <initialisation> ;
    while (<condition>)
    {
        <actions>
        <mise à jour> ;
    }
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

L'itération: `for` (2)

Exemple: affichage du carré des nombres entre 0 et 10

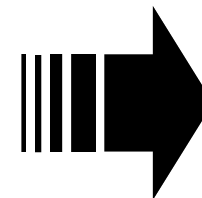
```
for (int i(0); i<=10; ++i)
{
   cout << i*i << endl;
}
```



Il est à noter que la variable de contrôle de la boucle (`i`) est déclarée et initialisée dans la clause `for`.

Lorsque plusieurs instructions d'initialisation ou de mise à jour sont nécessaires, elles sont séparées par des **virgules** (et sont évaluées de la gauche vers la droite)

```
for (int i(0),s(0); i<6; s+=i,++i)
{
   cout << i << s << endl;
}
```



```
0  0
1  0
2  1
3  3
4  6
5 10
```

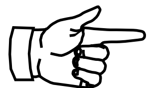
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Ruptures de séquence (1)

C++ fournit également des instructions de *rupture de séquence*, permettant de contrôler de façon plus fine le déroulement des structures de contrôles.

Parmis ces instructions, on trouve **`break`** et **`continue`**.

L'instruction `break`:
Elle ne peut apparaître qu'au sein d'une boucle ou d'une clause de sélection.
Elle permet d'interrompre le déroulement de la boucle (quelque soit l'état de la condition de continuation) ou de l'instruction de sélection, en provoquant un saut vers l'instruction suivant la structure de contrôle.



L'instruction `continue`:
Elle ne peut apparaître qu'au sein d'une boucle.
Elle interrompt l'exécution des instructions du bloc, et provoque la ré-évaluation de la condition de continuation, afin de déterminer si l'exécution de la boucle doit être poursuivie (avec une nouvelle itération).

Ruptures de séquence (2)

```
while (condition)
{
    ...
    /* actions de la boucle */
    ...
    break
    ...
    continue
    ...
}
/* actions suivant la
   structure de contrôle
*/
```

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Ruptures de séquence (3)

Utilisation de `break`:

```
/*
   une façon de simuler une boucle
   avec condition d'arrêt:
*/
while (true)
{
   /* actions */
   if ( <condition d'arrêt> )
   {
      break;
   }
}
```

Utilisation de `continue`:

```
for (int x = 0; x<100; ++x)
{
   /* actions exécutées pour
      tout x entre 0 et 99 incl */
   if (x%2)
   {
      continue;
   }
   /* action exécutées pour
      tout x impair
   /*
}
```

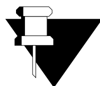
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Portée: variables locales/globales (1)

Les blocs ont une grande autonomie en C++.
Ils peuvent contenir leurs propres déclarations (et initialisations) de variables:

les variables déclarées à l'intérieur d'un bloc seront appelées ***variables locales*** au bloc; elles ne sont accessibles (référençables) qu'à l'intérieur de ce bloc;

les variables déclarées en dehors de tout bloc seront appelées ***variables globales*** au programme; elles sont accessibles dans l'ensemble du programme.



La portion de code dans laquelle une variable est utilisable est appelée la ***portée*** (ou *visibilité*[4]) de la variable, et s'étend usuellement de la déclaration de la variable jusqu'à la fin du bloc dans lequel est faite la déclaration

4. La visibilité effective d'une variable sera conditionnée par des règles de masquage qui seront vues plus loin dans le cours.

Portée: variables locales/globales (2)

```
// déclarations globales
int z(0);
...
void main()
{
   // déclaration locales
   int x,y;
   ...
   {
      // déclaration locales
      int y;
      ... x ...
      ... y ...
      ... z ...
   }
   ... y ...
   ... z ...
}
```

portée globale

portée locale **fonction** *main*

portée locale **bloc** interne

```
// début du programme
int z(0);
...
void main()
{
   // bloc niveau 1
   int x,y;
   ...
   {
      // niveau 2
      int y;
      ... x ...
      ... y ...
      ... z ...
   }
   ... y ...
   ... z ...
}
```

Portée: variables locales/globales (3)

En particulier, la déclaration d'une variable **à l'intérieur d'une boucle** est une déclaration **locale** au bloc associé à la structure de boucle.

Ainsi dans la structure de boucle

```
while (<condition>)
{
    int i(0);
    ...
}
```

la variable i est **locale** à la boucle

Et de même dans l'itération

```
for (int i(0); i<10; ++i)
{
    int j(0);
    cout << i << j << endl;
}
```

≡

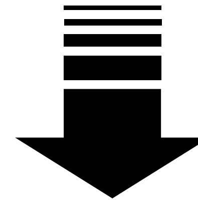
```
{
    int i(0);
    while (i<10)
    {
        int j(0);
        cout << i << j << endl;
        ++i;
    }
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Portée: variables locales/globales (4)

Exemple: le programme

```
const int MAX(5);
void main()
{
    int i(120);
    for (int i(1); i<MAX; ++i)
    {
        cout << i << endl;
    }
    cout << i << endl;
}
```



donne en sortie:

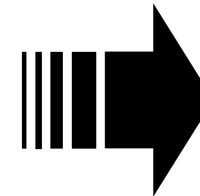
```
1
2
3
4
120
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Portée: variables locales/globales (5)

Autre exemple: le programme

```
const int MAX(5);
void main()
{
   int i(120);
   int j(0);
   while (j<MAX)
   {
      int i(0);
      ++j;
      cout << j << ',' << i << endl;
      i=j;
   }
   cout << j << ',' << i << endl;
}
```



donne en sortie:

```
1,0
2,0
3,0
4,0
5,0
5,120
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Notion de réutilisabilité (1)

A ce point du cours,
un programme n'est rien de plus qu'une
séquence bien formée d'instructions simples ou
composées (i.e. comprenant des structures de contrôle).

Exemple: soit P le programme d'affichage suivant[1]

```
cout << "*************************" << endl;
cout << "Ceci est  un petit texte" << endl
     << "       permettant       " << endl
     << "d'afficher quelque chose" << endl
     << "       à  l'écran       " << endl;
cout << "*************************" << endl;
```

Si la tâche réalisée par le programme P doit être exécutée plusieurs fois,
une possibilité est de recopier P autant de fois que nécessaire...

... mais c'est [évidemment] une **très mauvaise** solution !

1. On pourrait également songer à non pas un programme d'affichage, mais à une fonction mathématique, telle que *SinC(x)*, soit $f(x) = \frac{sin(x)}{x}$

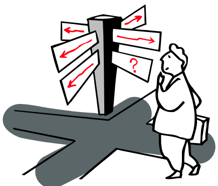
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Notion de réutilisabilité (2)

La duplication de larges portions identiques de code est à **poscrire**, car:

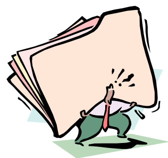
⇨ Cela rend la **mise à jour** du programme **fastidieuse**: il faut [manuellement] répercuter chaque modification de `P` dans chacune de ses copies

☞ c'est par conséquent une **source supplémentaire d'erreurs**, et cela dégrade donc la fiabilité globale du programme.



⇨ Cela **réduit la lisibilité** globale du programme: la taille du programme source est plus importante, ce qui concourt à rendre les modifications et la traque des erreurs plus difficiles.

⇨ Cela **augmente** (de manière injustifiée[2]) **la taille** – donc l'occupation mémoire – du programme objet résultant.



Un langage de programmation doit donc fournir des moyens plus efficaces pour permettre la réutilisation de portions existantes de programmes.

2. Cette augmentation de taille est injustifiée **car** il existe une alternative à la duplication manuelle du code.

Réutilisabilité: les fonctions (1)

Pour mettre en œuvre la **réutilisabilité**,
la plupart des langages de programmation fournissent des
mécanismes permettant de manipuler des **portions de programmes**.

En C++, ces mécanismes vont s'appuyer sur la notion de ***fonction***

De façon imagée, une fonction est assimilable à une expression,
dont l'évaluation (on dit l'**appel**) correspond à l'exécution de la portion
de programme qui lui est associée, et produit une valeur (résultat de l'évaluation),
la ***valeur de retour*** de la fonction.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Réutilisabilité: les fonctions (2)

Une **fonction** est donc une **portion réutilisable** de programme, associée à la définition d'une **interface explicite** avec le reste du programme (permettant de définir la manière d'utiliser cette fonction), par le biais d'une **spécification** des **arguments formels** (les «entrées») et de la **valeur de retour** (les «sorties»).

```
val = f(0.1,3,"ab");

f(x,2,"a");
```

Programme

arguments formels

x1

x2

x3

résultat

valeur de retour

interface de communication

```
{
    Portion réutilisable
    de programme

    return ...
}
```

fonction

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Réutilisabilité: les fonctions (3)

Plus précisément, une fonction est
un élément informatique
caractérisé par:

- un **identificateur**, qui n'est autre qu'une référence particulière à l'élément «fonction» lui-même;
- un **ensemble d'arguments formels**, qui correspondent en fait à un ensemble de références formelles à des entités définies à l'extérieur de la fonction, mais dont les valeurs seront potentiellement utilisées par la fonction;
- un **type de retour**, qui permet de préciser la nature du résultat retourné par la fonction
- un **corps**, qui correspond à un bloc (la portion de code réutilisable qui justifie la création de la fonction) et qui induit également une portée –locale– pour les éléments (variables, constantes) définis au sein de la fonction.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Prototypage (1)

Tout comme dans le cas des variables, la mise en œuvre d'une fonction se fait en 2 étapes:

- le ***prototypage*** ou *spécification* (équivalent de la déclaration des variables)
- la ***définition*** ou *implémentation* (à peu près équivalent de l'initialisation)

La syntaxe d'un ***prototypage*** est:

```
<type> <identificateur>(<arguments>);
```

Où *type* est le type de la valeur de retour renvoyée par la fonction[3], *identificateur* est le nom donné à la fonction, et *arguments* est la liste des arguments formels, de la forme:

$type_1\ id\text{-}arg_1,\ type_2\ id\text{-}arg_2,\ \ldots,\ type_n\ id\text{-}arg_n$

Exemple:

```
float puissance(const float base, float exposant);
```

3. On dit souvent (par extension), le «type de la fonction»

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Prototypage (2)

Dans les *prototypes* de fonctions, les identificateurs des arguments sont optionnels, mais servent généralement à rendre plus lisible le *prototype*.

Ainsi, la fonction `puissance` spécifiée précédemment pourrait être prototypée (de manière équivalente pour le compilateur, mais nettement moins lisible) par:

```
float puissance(const float, float);
```

Comme dans le cas des variables, les fonctions ont une portée (et une visibilité). Ceci implique que toute fonction doit être prototypée avant d'être utilisée.

Les prototypes des fonctions seront ainsi toujours placés avant le corps de la fonction *`main`*.

Exemple:

```
float puissance(const float base, float exposant);
void main()
{
    ...
    x = puissance(8.0,3.);
    ...
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Définition (1)

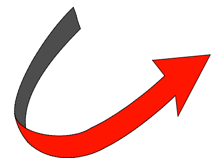
La ***définition*** permet «l'affectation» d'un corps (bloc d'instructions) au prototype de la fonction.

La syntaxe d'une ***définition*** est:

```
<type> <identificateur>(<arguments>)
{
   <corps de la fonction>
   [return <valeur>]
}
```

Où `corps de la fonction` est le **bloc** définissant la fonction, et dans lequel les arguments formels jouent le rôle de **variables ou constantes locales**, initialisées avec la valeurs des arguments employés lors de *l'appel* de la fonction.

Une instruction de rupture de séquence particulière, **`return`**, permet de terminer la fonction en retournant comme résultat de l'exécution la valeur spécifiée par `valeur`.



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Définition (2)

Exemple:

```
#include <iostream>

// Prototypages ..........................

float moyenne(const float x1, const float x2);

// Définitions ...............................

void main()
{
   cout << moyenne(8.0,3.0) << endl;
}


float moyenne(const float x1, const float x2)
{
   return (x1+x2)/2.0;
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Définition (3)

Comme dans le cas des variables, il est possible de réaliser le *prototypage* et la *définition* **en une seule étape** (dans ce cas la syntaxe est simplement celle de la définition)

On peut donc directement écrire:

```
float moyenne(const float x1, const float x2)
{
   return (x1+x2)/2.0;
}
void main()
{
   cout << moyenne(8.0,3.0);
}
```

Cependant, si pour les autres types de données il est recommandé de toujours initialiser lors de la déclaration, il est préférable, pour les fonctions, de réaliser le *prototypage* et la *définition* en 2 opérations distinctes.[4]

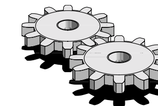
4. L'élément prédominant est de rendre le code le plus lisible possible. A cet fin, il est souvent préférable de regrouper les différentes déclarations en un endroit unique (typiquement en début de programme), et disposer ainsi d'une sorte d'index dans lequel il est aisé de retrouver le prototype d'une fonction donnée.

Evaluation d'un appel fonctionnel (1)

Pour une fonction *f* définie par: `int f(t1 x1, t2 x2, ..., tn xn) { ... }`
l'évaluation de ***l'appel fonctionnel*** `f(arg1, arg2, ..., argn)`
s'effectue de la façon suivante:

(1) les arguments $arg_1, arg_2, \ldots, arg_n$ sont évalués (de la gauche vers la droite), et les valeurs produites sont liées avec les arguments formels $x_1, x_2, \ldots, x_n$ de la fonction *f* (cela revient concrètement à réaliser les affectations: $x_1 = val_1$, $x_2 = val_2$, …, $x_n = val_n$), où $val_1, val_2, \ldots, val_n$ sont le résultat de l'évaluation des arguments ($val_i \Leftarrow (arg_i)$);

(2) le bloc correspondant au corps de la fonction *f* est exécuté;

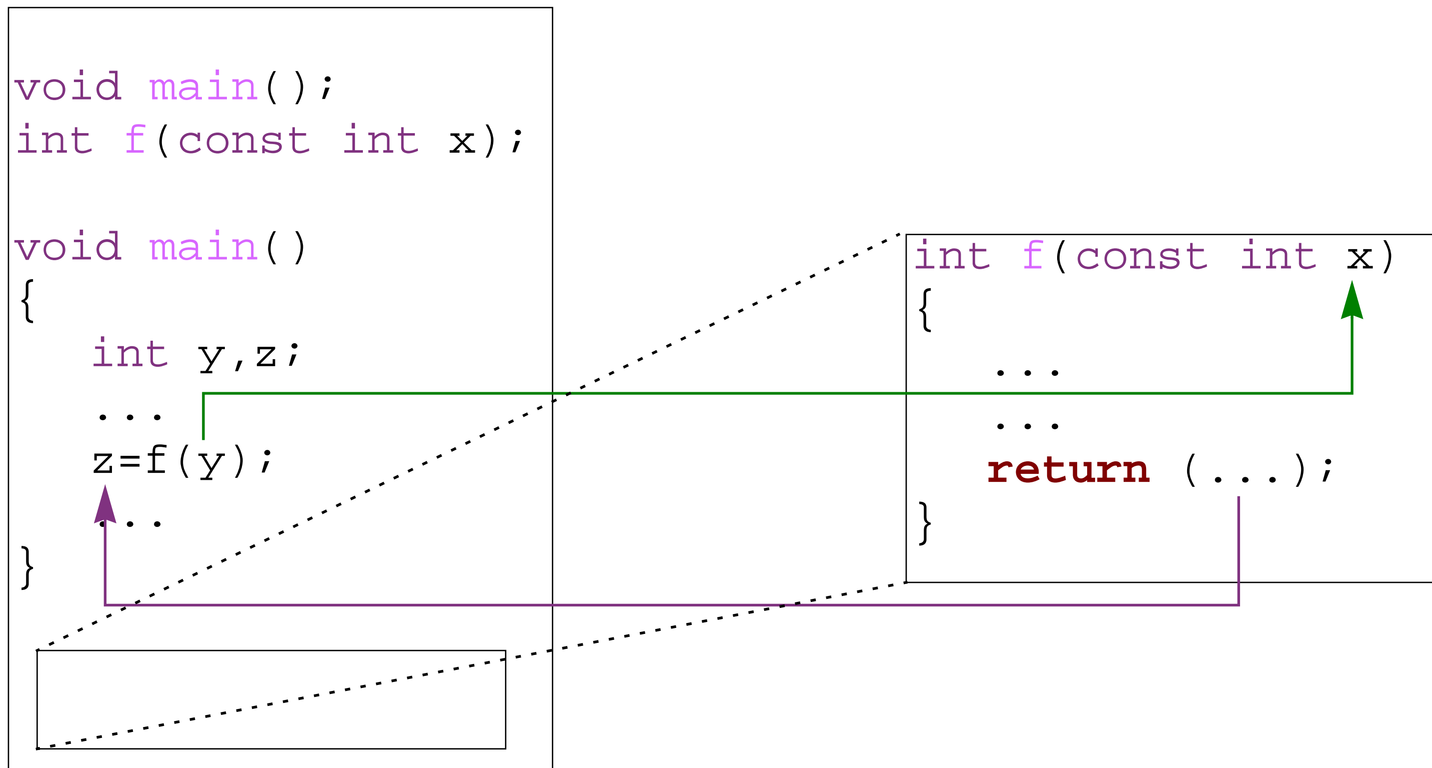


(3) l'expression (entière dans notre cas) définie par le mot clef `return` est évaluée, et est retournée comme résultat de l'évaluation de l'appel fonctionnel.

The EPFL logo (ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE) at top left

Evaluation d'un appel fonctionnel (2)

L'évaluation de l'appel fonctionnel peut être schématisé de la façon suivante:



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Evaluation d'un appel fonctionnel (3)

Pour la fonction `moyenne()` précédemment définie, soit l'appel fonctionnel:

`moyenne((sqrt(2*3*4)+1), 12%5)`

1 Evaluation des arguments et liaison avec les arguments formels:
```
x1 = (sqrt(2*3*4)+1) => x1 = 5.89897948557
x2 = 12%5 => x2 = 2.0
```

2 Exécution du corps de la fonction:
```
moyenne = (x1+x2)/2.0 => moyenne = 3.94948974278
```

3 La valeur de retour, correspondant à l'évaluation de l'expression «appel fonctionnel», est donc: `3.94948974278`

L'appel est alors équivalent à l'exécution du bloc:

```
float moyenne;
{
   const float x1(sqrt(2*3*4)+1);
   const float x2(12%5);
   moyenne = (x1+x2)/2.0;
}
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Fonctions sans valeur de retours

Il est également possible de définir des fonctions **sans valeur de retour** (i.e. des fonctions qui ne renvoient rien, et donc à peu près équivalentes à de simple sous-programme).

Cette absence de valeur de retour sera indiquée, dans le prototype et la définition de la fonction, par un type de retour particulier, le type **`void`**

Dans ce cas, le mot réservé `return` n'est pas requis, l'exécution de la fonction se terminant à la fin du bloc constituant son corps. Il est toutefois possible l'instruction `return` (dans ce cas sans indication d'expression de retour), afin de provoquer explicitement (et généralement prématurément) la terminaison de la fonction.

Exemple:

```
void afficheEntiers(const int n)
{
   for (int i(0); i<n; ++i)
      cout << i << endl;
   return;
}
```

```
void main()
{
   int n(10);
   afficheEntiers(n+1);
}
```

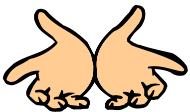
Fonctions sans arguments

De même, il est possible de définir des fonctions **sans arguments**.

Dans ce cas, la liste des arguments du prototype et de la définition est tout simplement vide (mais les parenthèses délimitant cette liste vide doivent être présentes)

Exemple:

```
int saisieEntiers();

void main()
{
   int val = saisieEntiers();
   cout << val << endl;
}

int saisieEntiers()
{
   int i;
   cout << "Entrez un entier: ";
   cin >> i;
   return i;
}
```



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

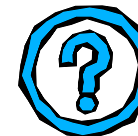
Notion de portée locale (1)

Les notions de *portée* ***locale*** v.s. ***globale*** définies pour les blocs sont également valides dans le cadre des fonctions[5].

Comme dans le cas des blocs,
la *portée* permet de résoudre les problèmes de conflits
de référençage entre entités ***externes*** (i.e. définies à l'extérieur du corps
de la fonction) et entités ***internes*** (i.e. définies dans le corps-même de la fonction).

Exemple:

```
int i(2);
void f(int a);
void main {
   f(i);
}
void f(int a) {
   int i(10);
   cout << a*i << endl;
}
```



Quelle variable est référencée par le `i` de l'instruction `cout << a*i << endl` ?

5. Cela est bien naturel, puisque le corps d'une fonction n'est autre qu'un bloc.

Notion de portée locale (2)

Les règles utilisées en C++ sont:

☞ En cas de conflit de référence entre une entité interne et une entité externe, **l'entité interne est systématiquement choisie**; on dit que les **références internes sont de portée locale**, i.e. limitées aux entités internes de la fonction. On dit également que les entités internes ***masquent*** les entités externes.

☞ Les **arguments formels** d'une fonction constituent des **entités internes** à sa définition.

☞ Les références (non ambiguës) à des entités externes sont possibles[6]. Il est néanmoins préférables de les éviter, en **explicitant systématiquement** toute référence à une entité externe **par le biais d'un argument formel**.

6. Une fonction modifiant des variables externes sera dite «*à effet de bord*», ce qui est fortement déconseillé.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Notion de portée locale (3)

Schématiquement, ces règles peuvent se représenter par:

```
void main();
int f(const int x);
int z;

void main()
{
   int x,y;
   ...
   ...
   z=f(y);
   ...
}
```

```
int f(const int x)
{
   int y;
...
...
   ... x ...
   ... y ...
   ... z ...
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Le passage des arguments

En C++, on distingue pour les fonctions
2 types de **passage d'arguments**:

☞ le *passage par **valeur***:

dans lequel la fonction travaille sur
des **copies locales** des arguments transmis

☞ et le *passage par **référence***:
dans lequel la fonction travaille effectivement
l'argument **transmis** lui-même**.**

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Passage par valeur (1)

La variable locale associée à un argument formel ***passé par valeur*** correspond à une ***copie locale*** de l'argument utilisé lors de l'appel de la fonction.

C'est donc une entité distincte, mais de même valeur littérale, et en conséquence, les modifications éventuelles apportées à l'argument formel à l'intérieur de la fonction n'affectent pas la variable originale:

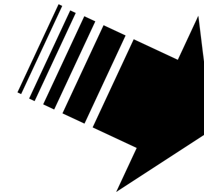
> Les modifications effectuées à l'intérieur de la fonction **ne sont pas répercutées à l'extérieur**.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Passage par valeur (2)

Exemple

```
void f(int x)
{
   x = x + 2;
   cout << "'x' vaut: " << x << endl;
}

void main()
{
   int val(0);
   f(val);
   cout << "'val' vaut: " << val << endl;
}
```



```
'x' vaut: 2
'val' vaut: 0
```

La modification de l'argument formel `x` effectuée dans la fonction `f()` laisse inchangée la valeur de la variable `val` utilisée comme argument (associé à `x`) lors de l'appel de la fonction..

De ce fait, il est tout à fait possible d'utiliser des valeurs littérales comme argument lors des appels aux fonctions [pour les arguments passés par valeur].

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Passage par référence (1)

La variable locale associée à un argument formel ***passé par référence*** ne correspond qu'à un autre nom (synonyme local) pour l'argument utilisé lors de l'appel de la fonction[7].

Aucune nouvelle variable n'est crée, et en conséquence, les modifications éventuelles apportées à l'argument formel à l'intérieur de la fonction affectent également l'argument original:

> Les modifications effectuées à l'intérieur de la fonction **sont répercutées à l'extérieur**.

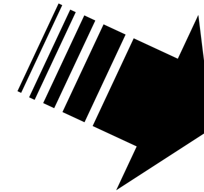
Le passage par référence doit être explicitement indiqué. Pour ce faire, on ajoute le symbole «&» au type des arguments formels concernés (par exemple: `int&`, `bool&`, ...).

7. Ce type de «variable» est appelé ***référence***, d'où le nom de *passage par référence*.

Passage par référence (2)

Exemple

```
void f(int& x) {
   x = x + 2;
   cout << "'x' vaut: " << x << endl;
}
void main() {
   int val(0);
   f(val);
   cout << "'val' vaut: " << val << endl;
}
```



```
'x' vaut: 2
'val' vaut: 2
```

Remarquons que dans le cas d'arguments passés par référence, les valeurs littérales ne peuvent être utilisés comme arguments lors de l'appel à la fonction que pour des arguments formels déclarés constants.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Entités de type «référence à»

De manière générale, les entités de type «*référence à*» (appelée plus simplement des ***références***) permettent de définir de nouveaux noms (**synonymes** ou ***alias***) pour des variables ou constantes existantes.

Leur déclaration se fait de la même manière qu'en paramètre des fonctions; la syntaxe est:

```
[const] <type>& <alias>(<entité>);
```

Où *entité* est une variable, une constante ou une référence de type *type*.[8]

Exemple:

```
char c('a');
const int i(2);
...
char& c_alias(c);                   // un simple synonyme de «c»
const int& i_alias(i);              // un synonyme (obligatoirement) constant de la constante «i»
const char& c_alias2(c_alias);      // un synonyme constant de la variable «c»
```

Une référence ne peut pas être ré-assignée. Une fois déclarée, elle restera donc toujours associée au même élément.

8. Il est possible de déclarer des références *constantes* à des variables, mais il n'est pas permis de déclarer des références *simples* à des constantes.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Arguments par défaut (1)

Lors de sa définition,
une fonction peut être munie d'**arguments avec valeur par défaut**,
pour lesquels il n'est alors pas obligatoire de fournir de valeur
lors de l'appel de la fonction.

La syntaxe pour définir des arguments
avec **valeur par défaut** est:

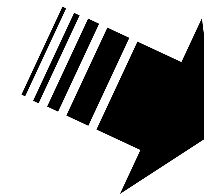
```
<type> <argument> = <valeur>
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Arguments par défaut (2)

Exemple

```
void afficheLigne(const char c, const int n = 5)
{
   for (int i(0); i<n; cout << c, ++i);
   cout << endl;
}

void main()
{
   afficheLigne('*');
   afficheLigne('+',8);
}
```



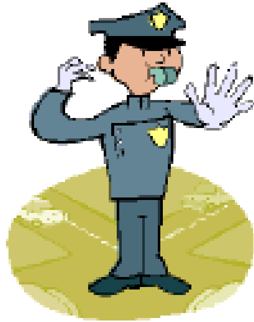
```
*****
++++++++
```

Dans le cas du premier appel («`afficherLigne('*')`»),
la valeur du second argument n'étant pas spécifiée,
celui-ci prend la valeur par défaut «5».

Par contre, il prend la valeur explicitement spécifiée («8»)
dans le cas du second appel («`afficherLigne('+',8)`»).

Arguments par défaut (3)

Attention



- Dans le cas d'une fonction avec *prototype* et *définition* distincts, la spécification des arguments avec valeur par défaut **ne doit être réalisée que lors du *prototypage***.
- Les arguments avec valeur par défaut doivent apparaître **en dernier** dans la liste des arguments de la fonction.
- Lors de l'appel d'une telle fonction, avec plusiqurs arguments avec valeur par défaut, les arguments omis doivent être **les derniers** de la liste des arguments optionnels.

Exemple:

```
void f(int x, int y=2, int z=3); // prototype
void f(int x, int y, int z) { ... } // définition
...
f(1) <=> f(1,2,3)
f(0,1) <=> f(0,1,3)
```

Surcharge des fonctions (1)

On appel «***signature*** d'une fonction» le couple constitué de l'identificateur de la fonction et de la liste de types de ses arguments formels.

(le type de retour de la fonction ne fait pas partie de la signature !)

Comme c'est par le biais de leur signature que le compilateur identifie les fonctions utilisées par le programmeur[9], il est de ce fait tout à fait possible de définir **plusieurs fonctions avec un même identificateur** (nom), **mais de listes d'arguments différentes** (soit par le nombre d'arguments, soit par leur type).

Un tel mécanisme, appelé ***surcharge*** [des fonctions], est très utile pour l'écriture de fonctions correspondant à des traitements de même nature, mais s'appliquant à des entités de types différents.

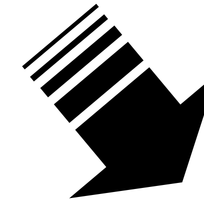
9. La ***signature*** d'une fonction est donc similaire à la notion d'***attribut identifiant*** dans le domaine des bases de données.

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

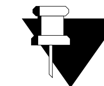
Surcharge des fonctions (2)

Exemple:

```
void affiche(const int x) {
   cout << "entier: " << x << endl;
}
void affiche(const float x) {
   cout << "réel: " << x << endl;
}
void affiche(const int x1, const int x2) {
   cout  << "couple: ("
         << x1 << ',' << x2 << ')' << endl;
}

void main() {
   affiche(1.0);
   affiche(1);
   affiche(5,2);
}
```



```
réel: 1.0
entier: 1
couple: (5,2)
```



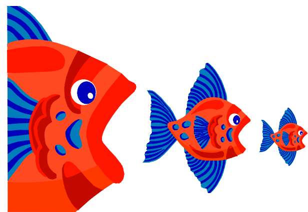
Les arguments avec valeur par défaut **ne font pas** partie de la signature.

Algorithmique et récursion

Pour un problème donné,
par exemple le tri d'une table,
de nombreuses solutions algorithmiques
peuvent être applicables.

Une catégorie particulière de solutions sont
les **résolutions *récursives***.

Le principe de l'approche récursive est de
**ramener le problème à résoudre à un sous-problème
correspondant à** ***une instance «réduite» du problème lui-même***.



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Exemple de récursion appliquée au tri

tri

entrée: table de *n* éléments

sortie: table triée

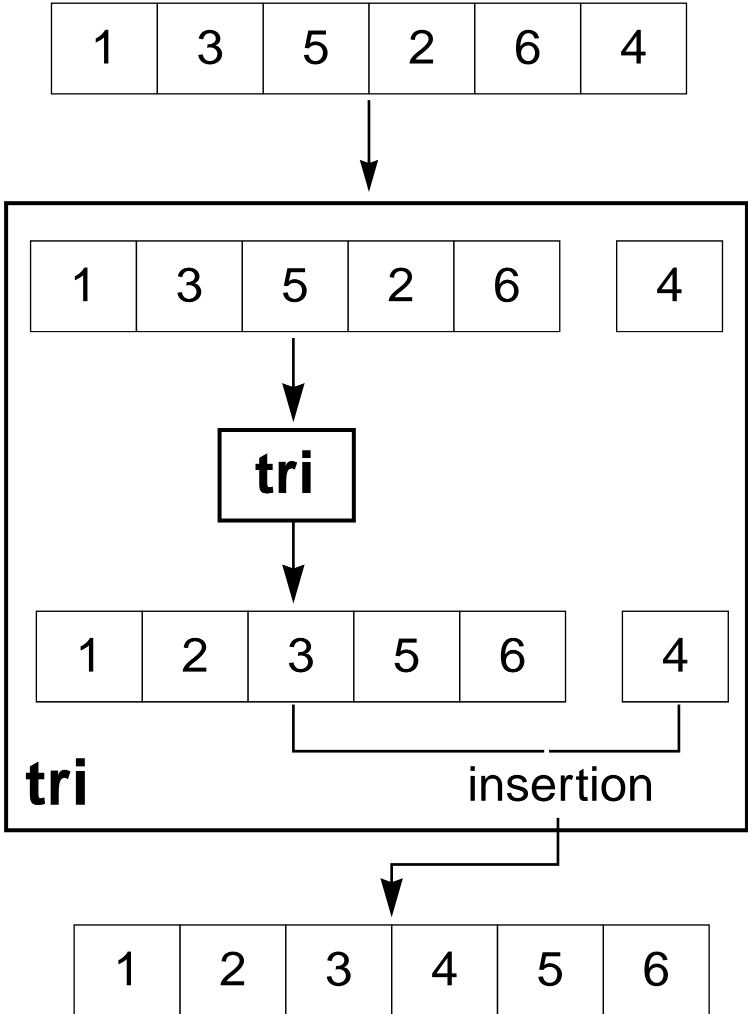
tri (instance réduite du problème)

entrée: table de *n-1* éléments

sortie: table triée

...

insertion: insertion (au bon endroit) du n^e élément dans le tableau trié de *n*-1 éléments



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

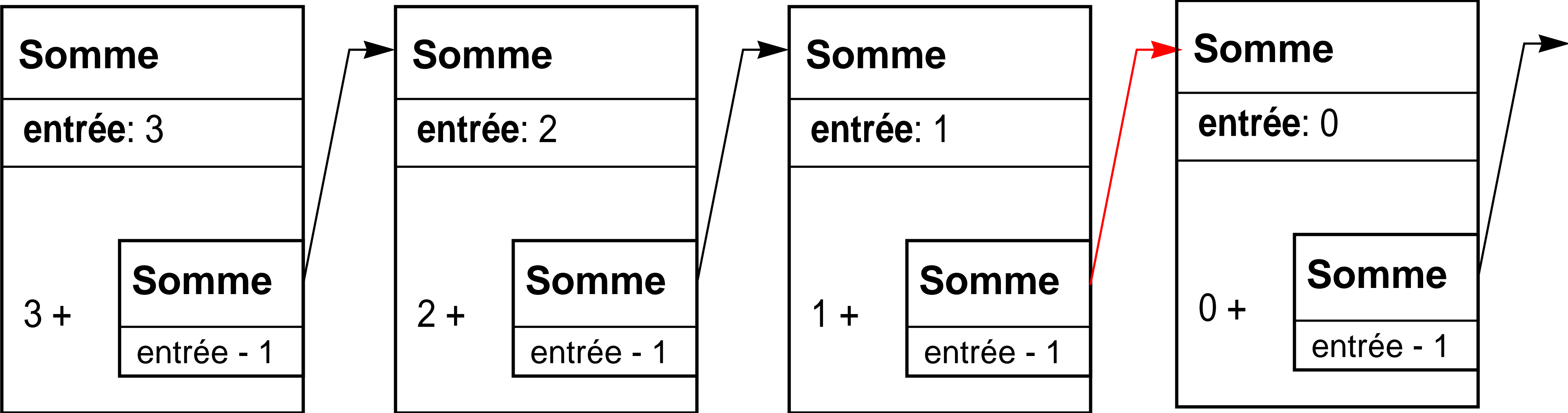
Algorithme récursif (1)

Attention

Pour que la résolution récursive soit correcte,
il faut une **condition de terminaison** portant sur les entrées.

Sinon, on risque fort de boucler indéfiniment !

Exemple: la somme des *n* premiers entiers positifs



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Algorithme récursif (1)

Le schéma général d'un algorithme récursif **correct** est donc:

AlgoRécursif

entrées: `entree`

si condition de terminaison atteinte
    alors retourner une valeur de sortie
sinon

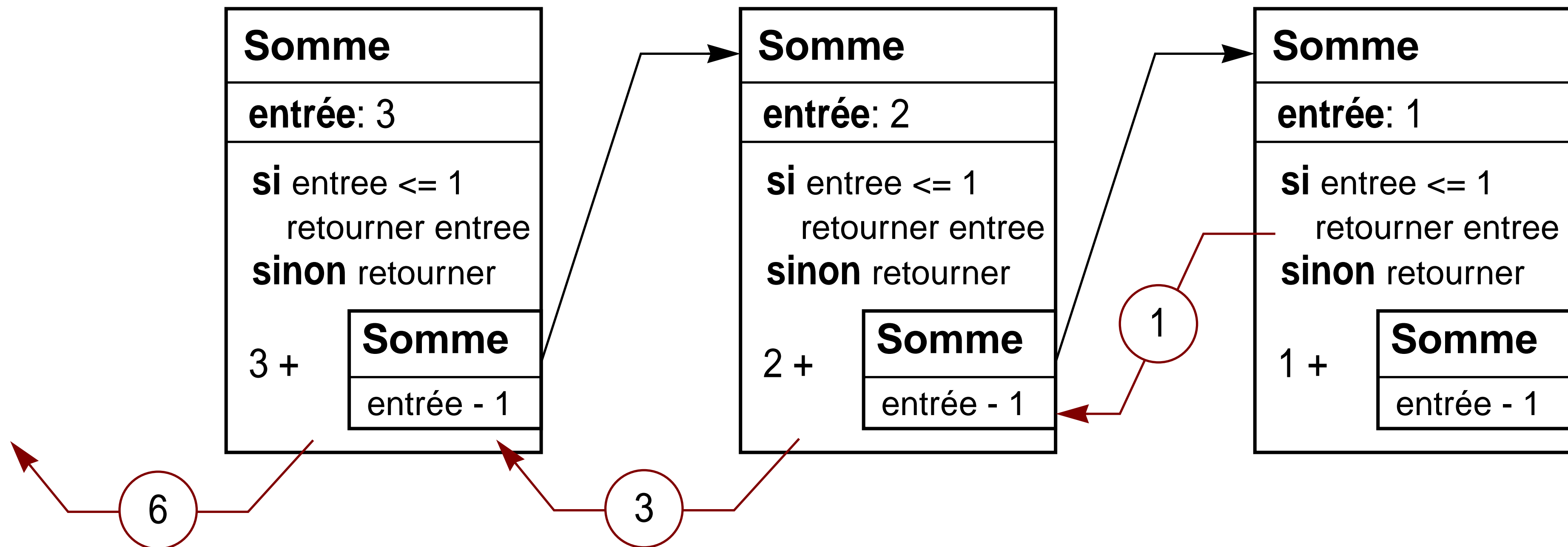
> **AlgoRécursif (instance réduite)**
>
> **entrée**: entrée de l'instance réduite
>
> ...

terminer le traitement et retourner une valeur de sortie.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Algorithme récursif (2)

Appliqué à l'exemple de la somme des n premiers entiers positifs, on obtient l'algorithme correct suivant:



Algorithme récursif (3)

Par exemple, le schéma de l'algorithme récursif `tri_recursif` est:

tri_recursif

---

entrée: tableau de *n* éléments
sortie: tableau triée

---

si taille du tableau <= 1,
le tableau est trié, et on peut le retourner

sinon:

[tri_recursif]
du tableau privé de
son dernier élément

[insertion]
du dernier élément à la bonne
place dans le sous-tableau trié

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Les fonctions récursives (1)

Concrètement, un algorithme récursif sera implémenté par une ***fonction récursive***.

Comme la définition d'une fonction (i.e. son corps) peut faire appel à toute fonction précédemment prototypée dans le même programme, elle peut donc, en particulier, *faire appel à elle-même*.

Une fonction qui, dans sa définition, fait appel à elle-même est appelée une ***fonction récursive***[10]

Pour être effectivement utilisable, une fonction récursive doit posséder une **condition d'arrêt** (conditionnant l'exécution ou non de l'appel récursif)

10. Sur le plan syntaxique, une telle fonction n'est pas différente des autres. La seule différence réside dans le fait que la définition de la fonction induit un cycle, par un appel à elle-même. Remarquons que l'on appelle également récursives des fonctions couplées (A appelle B qui appelle A).

Les fonctions récursives (2)

Le schéma général d'une *fonction récursive*
est donc le suivant:

```
typeret fonctionRec(type1 x1, type2 x2, ...)
{
   typeres result;
   if (terminaison(x1, x2, ...)
   {
      ...
      result = ...
   }
   else
   {
      type1 y1 = ...
      type2 y2 = ...
      ...
      result = ... fonctionRec(y1, y2, ...)
   }
   return result;
}
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Les fonctions récursives (3)

Exemple: la somme des *n* premiers entiers positifs

```
// prototypage
int somme(const int n);

// définition
int somme(const int n)
{
   if (n >= 1) // condition d'arrêt
   {
      return n;
   }
   else
   {
      return (n + somme(n-1)); // appel récursif
   }
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Les fonctions récursives (4)

Second exemple: on reprend l'algorithme de tri proposé précedemment:

On suppose en outre disposer des fonctions suivantes:

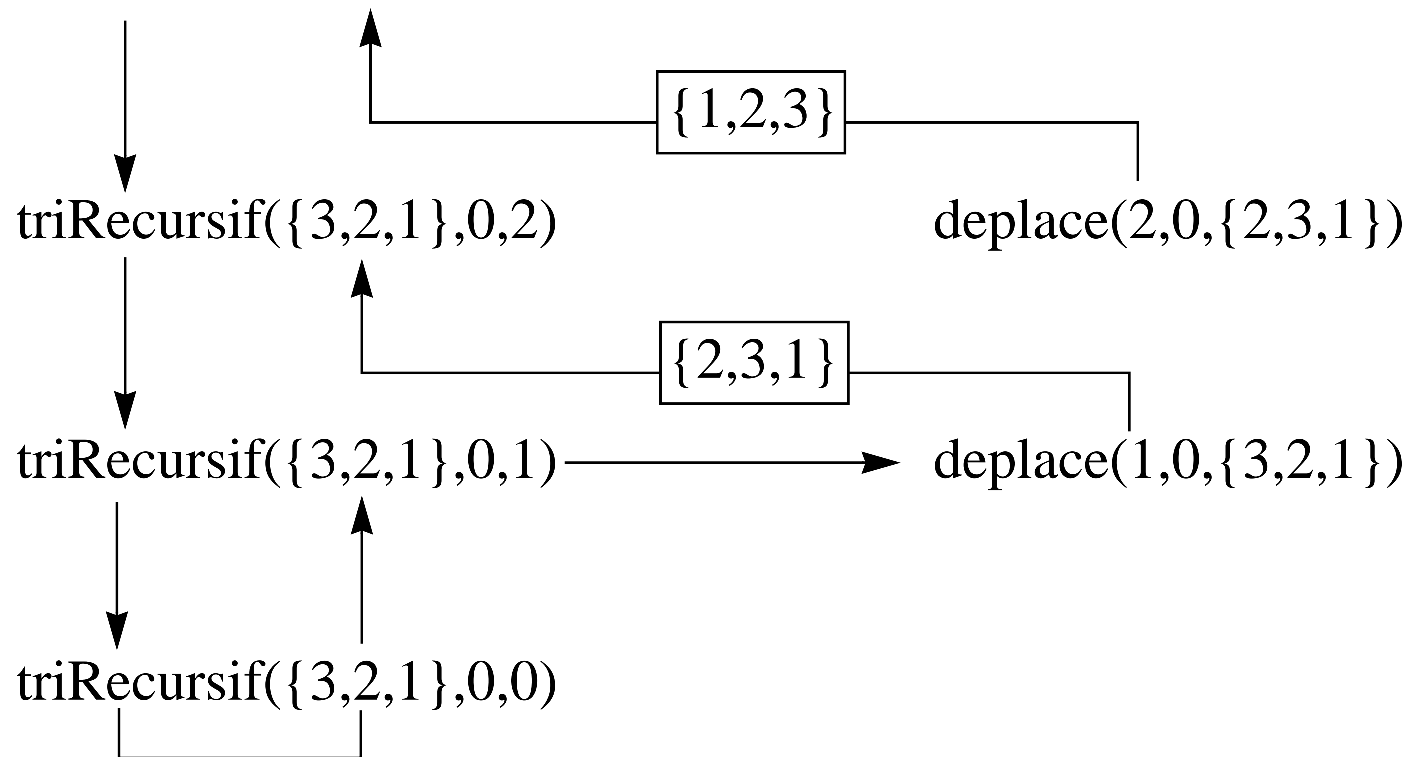
- `int bonnePlace(const int index, vector<int>& table)`
  qui retourne la «bonne place» de l'élément d'indice `index` dans le tableau `table`
- `void deplace(const int from, const int to, vector<int>& table)`
  qui déplace l'élément d'indice `from` vers l'indice `to`, en décalant les éléments de `table` existant entre `from` et `to`

```
void triRecursif(vector<int>& table, const int from, const int to)
{
   if (from == to) // un seul élément -> (sous-)table triée
      return;
   else
   {
      triRecursif(table, from, to-1); // appel récursif
      deplace(to, bonnePlace(to, table), table);
   }
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Les fonctions récursives (5)

Le scéma des appels récursifs pour
un appel tel que: `triRecursif({3,2,1},0,2)`:



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Structures de données: types composés

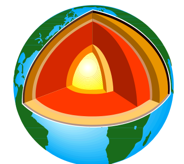
Les **types élémentaires** (nombres entiers, réels, caractères, booléens, ...) permettent de représenter des **concepts simples** du monde réel: des dimensions, des sommes, des tailles, ...

Les identificateurs associés permettent de donner une signification aux divers entités manipulées (pour cette raison, il faut toujours utiliser des noms aussi explicites que possible)

Exemples:

```
age = 18; poids = 62.5; taille = 1.78; ...
```

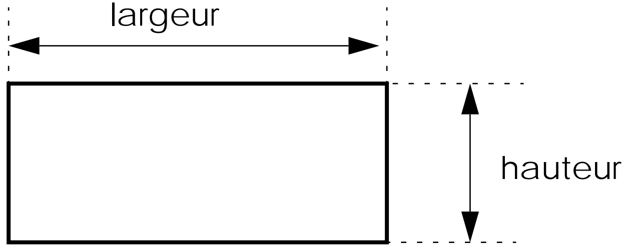
Cependant, de nombreux concepts plus sophistiqués ne se réduisent pas à une seule entité élémentaire...



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Exemple de type composé

Un rectangle par exemple peut être définis par 2 grandeurs élémentaires: sa «largeur» et sa «hauteur».



Si l'on ne dispose que de type élémentaires, ces grandeurs seront représentée indépendamment l'une de l'autre, et leur relation devra être indiquée par le programmeur, par exemple au moyen d'identificateurs tels que: `rectangle_largeur` et `rectangle_hauteur`.

`Rectangle`

longueur	***hauteur***
20	1.75

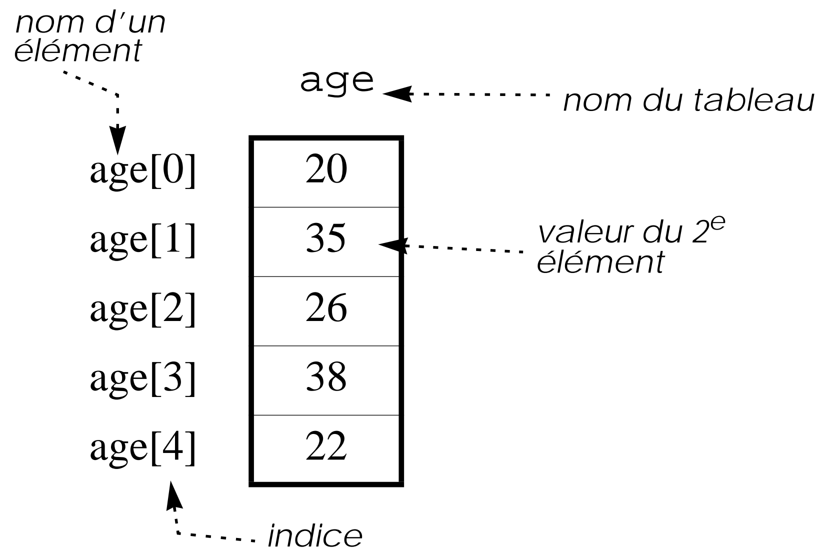
Afin d'éviter au programmeur d'avoir à assurer la cohésion des entités composites, un langage de programmation doit fournir le moyen de combiner les type élémentaires pour construire et manipuler des objets informatiques plus complexes.

Ces types non élémentaires sont généralement appelés **types composés**.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

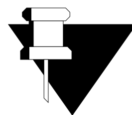
Types composés: les tableaux

Une première façon de composer des types élémentaires est de regrouper une collection d'entités (de même type) dans une structure tabulaire: le ***tableau***



En C++, les tableaux sont des ensembles d'un **nombre fixe** d'éléments:

- de même type, appelé ***type de base*** (le tableau est donc une structure *homogène*);
- référencés par un même identificateur, **l'identificateur du tableau**;
- individuellement accessibles par le biais d'un **indice**.



Le type de base est quelconque: il peut s'agir de n'importe quel type, élémentaire ou composé. On pourra donc définir des tableaux d'entiers, de réels, de caractères, ... et de tableaux.

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Tableaux: déclaration

Un tableau de taille fixe peut être déclaré selon la syntaxe suivante:

```
<type> <identificateur>[<taille>];
```

Avec *type* n'importe quel type, élémentaire ou non,
et *taille* une valeur entière littérale ou une expression entière constante
(i.e. évaluable lors de la compilation)

Cette instruction déclare un tableau désigné par *identificateur*,
comprenant `taille` éléments de type *type*.

Exemple:

Ainsi, le tableau *age* vu précédemment
peut être déclaré par:

```
int age[5];
```

age	
age[0]	20
age[1]	35
age[2]	26
age[3]	38
age[4]	22

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Tableaux: *déclaration-initialisation*

Naturellement, un tableau de taille fixe
peut être **initialisé** directement, lors de sa déclaration.

La syntaxe est alors:[1]

```
<type> <identificateur>[<taille>] = {<val_1>,...,<val_n>};
```

Exemple:

```
int age[5] = {20,35,26,38,22};
```

Remarquons que dans le cas d'une *déclaration-initialisation*,
la spécification de la taille du tableau est **optionnelle**.

Ainsi, la déclaration-initialisation précédente peut également être écrite:

```
int age[] = {20,35,26,38,22};
```

1. Il est également possible de n'initialiser que les premiers élements du tableau, comme dans: «`int age[5] = {20,35}`».

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Tableaux: accès aux éléments

Chaque élément du tableau est **accessible individuellement**.

Pour cela, on utilise un ***indice***[2] **placé entre crochets** «[ ]», indiquant le rang de l'élément dans le tableau[3].

Les éléments d'un tableau de taille *n* sont sont numérotés de 0 à *n*-1.

Il n'y a pas de contrôle de débordement du tableau !

Exemple:

L'accès au premier élément du tableau `age` s'obtient au moyen de l'expression: «`age[0]`», et l'accès au dernier élément par «`age[4]`», comme dans:

```
age[0] = 3;
age[4] = age[1+2]; // soit age[4] = age[3]
```

2. Indice qui peut être une expression numérique.
3. Pour être tout à fait précis, on n'accède pas aux élements en spécifiant un rang, mais un déplacement (une distance, dont l'unité est l'élément/exprimée en nombre d'éléments), relativement au premier élément du tableau. C'est pour cela que l'indiçage débute à 0 et non à 1.

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Tableaux: tableaux de caractères (1)

Les tableaux de caractères forment une catégorie particulière de tableaux. Ils sont en particulier utilisés pour représenter les **chaînes de caractères** (délimitées par les guillemets anglais `""`).

Ainsi, une définition tel que:

```
char mot[7] = {'b','o','n','j','o','u','r'};
```

pourrait être utilisée pour représenter la chaîne `"bonjour"`.

Cependant, comme une chaîne de caractères étant un élément de taille variable, il faut lui associer une information permettant d'en retrouver la longueur.

A l'inverse de langages comme *Pascal*, C++ ne code pas explicitement cette longueur, mais délimite la chaîne par un caractère spécial – «`'\0'`» –, marquant la fin de la chaîne.

La chaîne `"bonjour"` sera donc implémentée par:

```
char mot[8] = {'b','o','n','j','o','u','r','\0'};
```

et sa taille sera de **8 caractères**, et non pas 7 !

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Tableaux: tableaux de caractères (2)

La syntaxe de C++ autorise cependant une initialisation **plus pratique** des «tableaux-chaîne de caractères»:

```
char mot[] = "bonjour";
```
[4]

Une telle représentation des chaînes, héritée du langage C, à comme inconvénient de **fixer la taille de la chaîne** de caractères.

Cette manière de procéder impose un certains nombre de contraintes lors d'opérations sur les chaînes de caractères (par exemple l'affectation, la concaténation de deux chaînes, etc.)

Ceci non seulement diminue l'efficacité de ces traitements, mais surtout oblige les programmeurs à effectuer eux-mêmes un certain nombre de manipulations dans le cas d'opérations sur les chaînes qui ne sont pas fournies dans les librairies standards.

Pour pallier ce défaut, on pourra utiliser, comme nous le verrons plus loin dans le cours, un type issu de l'approche objet de C++, le type **`string`**

4. Cette instruction est effectivement équivalente à celle vue précedemment (incluant le caractère `'\0'`)

Tableaux: tableaux multidimensionnels (1)

Le type de base d'un tableau est peut être un type quelconque, y compris composé. En particulier, le type de base d'un tableau peut être lui-même un tableau.

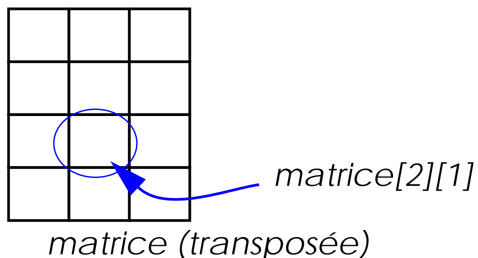
La syntaxe est alors:

<type> *<identificateur>*[*<*dim_1*>*][*<*dim_2*>*][...][*<*dim_n*>*];

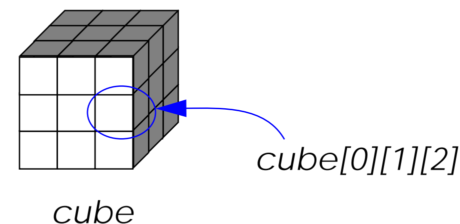
Les entités d'un tel type sont alors des tableaux de tableaux, soit des tableaux multidimensionnels[5].

Exemple:

```
int matrice[4][3];
```




```
float cube[3][3][3];
```



5. En fait, les tableaux informatiques correspondent aux vecteurs mathématiques, et les tableaux de tableaux à des matrices.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Tableaux: tableaux multidimensionnels (2)

Les tableaux multidimensionnels sont naturellement initialisables lors de leur déclaration, et chaque éléments est accessible individuellement.

Exemple:

```
int matrice[4][3] = {{0,1,2},{3,4,5},{6,7,8},{9,10,11}};
matrice[2][1] = 70;
```

matrice[ ][j] (colonnes) — *matrice[i][ ]* (lignes)

0	1	2
3	4	5
6	70	8
9	10	11

Dans les deux cas, il faut spécifier autant d'indices qu'il y a de dimensions dans le tableau.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Tableaux: sizeof

C++ fournit un opérateur unaire (i.e. à 1 argument), **`sizeof`**, qui retourne la taille de son opérande, comptée en octets.

`sizeof(``<`*`opérande`*`>`**`)`**

opérande pouvant être un type quelconque[6], notamment un tableau, il est possible d'utiliser cet opérateur pour déterminer automatiquement le rang du dernier élément d'un tableau, et éviter ainsi les ennuis causés par un dépassement des limites du tableau:

```
typedef float vecteur;
vecteur v[5] = {0.1, 0.1, 0.3, 0.0, -0.3};

// Affiche le contenu du vecteur 'v', du dernier élément au
// premier, en les séparant par un espace.

for (int i(sizeof(v)/sizeof(float)); i>0; --i)
{
   cout << v[i] << ' ';
}
```

6. Il peut également s'agir d'une variable ou une constante, ainsi que d'une expression.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Tableaux dynamiques: *vecteurs*

Pour pallier les défauts inhérents à la rigidité des tableaux de taille fixe (*built-in array*), la librairie (générique) standard[7] de C++ fournit un type de donnée[8] dénommée ***vector*** (*vecteur*), offrant au programmeur un moyen très efficace pour construire des structures de données permettant de représenter des **tableaux de tailles variables** (i.e. *tableaux dynamiques*)[9].

La taille de ces «tableaux» n'est pas obligatoirement prédéfinie, et peut donc varier en cours d'utilisation.

Pour pouvoir utiliser ces *vecteurs* dans un programme, il faut, comme dans le cas des entrées-sorties, importer les prototypes et définitions contenus dans la librairie, au moyen de la directive d'inclusion:

```
#include <vector>
```

7. Le nom officiel de cette librairie est ***STL*** (*Standard Template Library*)
8. En fait de type, il s'agit en réalité d'un *chablon* de *classe* (***template classe***), c'est-à-dire une définition **générique** (valide et réutilisable pour n'importe quel type, de base ou complexe).
9. Pour être exact, les *vectors* sont plus que de simples tableaux dynamiques; ils s'inscrivent dans un famille plus générale d'éléments, utilisés comme briques de bases pour les structures de données complexes, éléments que l'on appelle ***conteneurs*** (*containers*) ou ***collections***, et pour lesquels un ensembe de caractéristiques et contraintes très précises sont définies, comme par exemples les performances minimales des algorithmes d'accès et de recherche.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Vecteur: déclaration (1)

Un *vecteur* peut être déclaré selon la syntaxe suivante:

```
vector<«type»> «identificateur»;
```

Avec *type* n'importe quel type, élémentaire ou non, et correspondant au type de base du tableau.

Exemple:

```
#include <vector>
...
vector<int> age;
```

Il s'agit d'une déclaration de variable («age») tout à fait traditionnelle, dans laquelle la séquence «vector<int>» correspond à l'indication du type de la variable, en l'occurence un tableau dynamique (*vecteur*) d'entiers.

On voit dans ce cas clairement ressortir la nature composite du type.

Vecteur: déclaration (2)

Le fait que l'on s'intéresse ici à des *collections* d'un nombre potentiellement variable d'éléments explique que la déclaration puisse ne comporter aucune indication sur la taille initiale du tableau. Une variable ainsi déclarée correspond alors tout simplement à un tableau vide.

Cependant, une taille initiale peut, si nécessaire, être indiquée;
la syntaxe de la déclaration est alors:

```
vector<«type»> «identificateur»(«taille»);
```

Un tableau nommé *identificateur* comportant *taille* éléments de type *type* sera créé, chacun des éléments ayant comme valeur la valeur par défaut de *type* généralement une valeur dérivée de l'expression (0).

Exemple:

```
vector<int> age(5);
```

Correspond à la déclaration d'un tableau d'entiers, initialement composé de 5 éléments valant 0.

	age
age[0]	0
age[1]	0
age[2]	0
age[3]	0
age[4]	0

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Vecteur: déclaration avec initialisation

La déclaration d'un ***vecteur*** peut être associée à une **initialisation explicite** des éléments initiaux; cependant, cette initialisation ne pourra consister qu'en (a) une duplication d'un même élément, ou (b) en une duplication d'un *vecteur* pré-existant:[10]

(a) **vector<**«*type*»**>** «*identificateur*»**(**«*taille*»**,**«*valeur*»**);**
où *valeur* est une expression de type *type*, dont le résultat sera pris comme valeur intiale des *taille* éléments du tableau *identificateur*.

Exemple:

```
vector<int> vect1(5,8);
```

déclare le *vecteur* d'entiers vect1 avec un contenu initial de 5 entiers valant «8»

(b) **vector<**«*type*»**>** «*identificateur*»**(**«*id-vector*»**);**
où *id-vector* est un identificateur de *vecteur* de type de base *type*.

Exemple:

```
vector<int> vect2(vect1);
```

déclare le *vecteur* d'entiers vect2, avec un contenu initial identique au contenu de vect1 (duplication).

10. Contrairement au cas des tableaux de taille fixe, il n'existe pas de moyen simple pour exprimer la valeur littérale d'un *vecteur* dont les éléments n'ont pas tous la même valeur.

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Vecteur: constante

Comme pour tous les autres types, il est possible de déclarer des **constantes** de type *vecteur*

La syntaxe est identique à celle des autres déclarations de constantes:

```
const vector<«type»> «identificateur»(«initialisation»);
```

identificateur correspond alors à un *vecteur* dont tant le nombre d'éléments que la valeur de chacun de ces éléments sont fixes (et ne peuvent donc êtres modifiés).

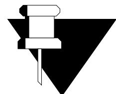
Exemple:

```
const vector<int> age;
```

Correspond à la déclaration d'un *vecteur* constant vide (ne contenant aucun élément) et auquel aucun élément ne pourra être ajouté[11]

```
const vector<int> vect2(vect1);
```

Correspond à la déclaration d'une copie figée (*snapshot*) du vecteur `vect1`.



Il n'est pas possible de déclarer des vecteurs de constantes.
Ainsi, la syntaxe `vector<const «type»> «identificateur»` bien que licite en soit, n'est en pratique pas utilisable.

11. Cette déclaration est bien sûr totalement inutile.

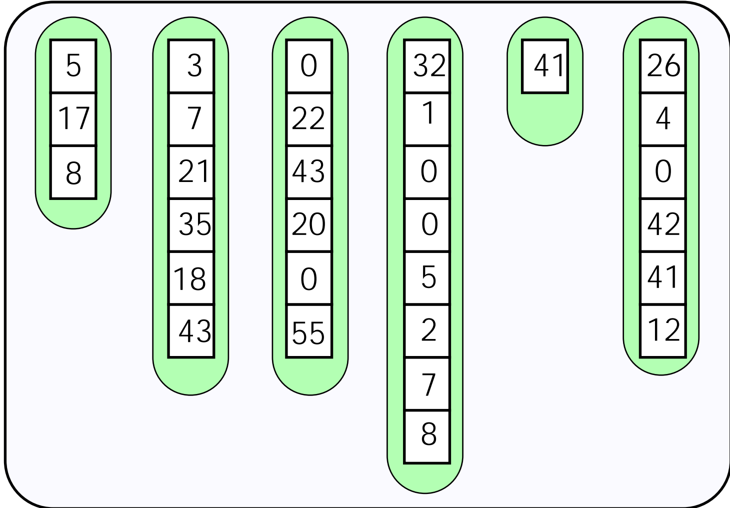
Vecteur: vecteur de vecteur

Le type de base d'un *vecteur* est peut être un type quelconque, y compris composé. En particulier, le type de base d'un *vecteur* peut être lui-même un *vecteur*.

```
vector<vector<int> > matrice;
```

Attention, dans le cas d'une déclaration directe d'une telle structure, il est obligatoire de séparer les «>», présents en fin de définition du type, par un espace[12].

D'un point de vue sémantique, les *vecteurs* de *vecteurs* ne correspondent pas (nécessairement) à des matrices, mais simplement à des ensembles d'ensembles d'éléments.



vector<vector<int> >

12. Cette contrainte est en fait une convention adoptée pour distinguer ce typage de l'opérateur «>>»

Vecteur: affectation

Toute **variable**[13] de type *vector* peut être modifiée (globalement) par affectation:

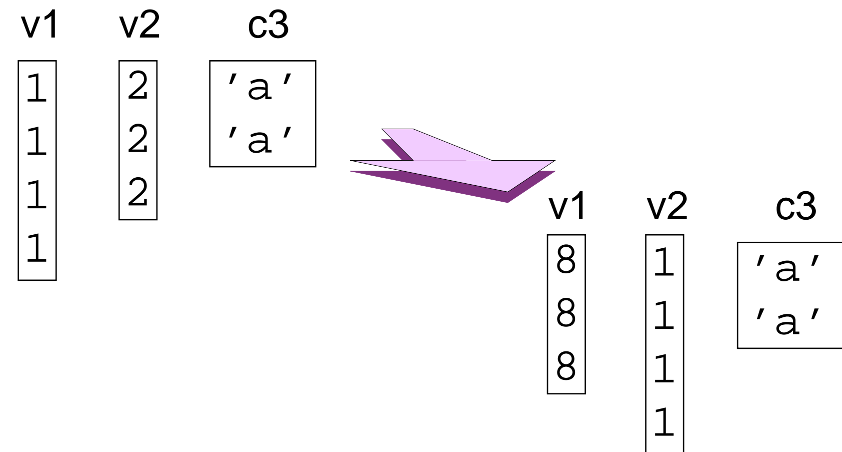
«*identificateur*» **=** «*valeur*»**;**[14]

où *valeur* est une expression de même type qu'*identificateur*,
notamment en ce qui concerne le type de base.

Dans ce cas, la modification va porter sur l'ensemble des éléments de la structure:

Exemple:

```
// 1) Déclarations-initialisations
vector<int> v1(4,1);
vector<int> v2(3,2);
vector<char> c3(2,'a');
// 2) Affectations .........
v2 = v1;
v1 = vector<int>(3,8);
c3 = vector<int>(4,7); // ILLEGAL
```



13. Sauf [naturellement] les constantes.

14. Parmis les opérateurs définis pour les *vecteurs*, on trouve en effet celui d'*affectation*, i.e. « = ».

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Vecteur: accès aux éléments

Chaque élément d'un vecteur est **accessible individuellement**.

Différents moyens permettent d'accéder aux éléments,
et notamment l'indexation comme dans le cas des tableaux de taille fixe:

L'***indice***[15]**, placé entre crochets** «[ ]», indique le rang de l'élément dans le tableau.

Les éléments d'un vecteur de taille n sont sont numérotés de 0 à n-1.
Dans le cas d'accès via l'opérateur «[ ]»,
Il n'y a pas de contrôle de débordement du tableau !

15. Indice qui peut être une expression numérique.

Vecteur: opérateurs relationnels

Les **opérateurs relationnels** suivant
sont définis pour les vecteurs:

Opérateur	*Opération*	
<	strictement inférieur	comparaison lexicographique des éléments.
<=	inférieur ou égal	
>	strictement supérieur	
>=	supérieur ou égal	
==	égalité	
!=	différence (non-égalité)	

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Vecteur: méthodes (1)

Un certain nombre d'opérations,
directement liées à l'aspect «ensembliste» des vecteurs,
sont définies dans la librairie *STL*.

L'utilisation de ces opérations particulières, appelées ***méthodes***[16],
se fait par le biais de la syntaxe suivante:

```
«id-vector».«id-methode»(«arguments»);
```

Exemple:

```
individus.puch_back(jean);
```

à comme effet d'appliquer la méthode `push_back`, au vecteur `individus`, en prenant `jean` comme paramètre [de la méthode].

Comme les fonction, les méthodes peuvent éventuellement retourner une valeur.

16. Les méthodes sont des éléments informatiques (séquences d'instructions) issus de l'extension «objet» de C++. Pour le moment, il vous suffit de les considérer comme des fonctions ayant une syntaxe d'appel un peu particulière.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Vecteur: méthodes prédicats

Parmi les méthodes disponibles, on trouve[17]:

Prédicats:

- `int` **`size`**`()`: renvoie la taille du *vecteur* (i.e. son nombre d'éléments).

  Une manière usuelle pour parcourir les éléments d'un *vecteur* est donc l'itération `for` suivante:

```
for (int i(0); i<vect.size(); ++i)[18]
{
   // traitements
}
```

- `bool` **`empty`**`()`: indique si le vecteur est vide (ou non).

  On a l'équivalence suivante: «`vect.empty()`» <=> «`(vect.size() == 0)`»

17. Par convention:
    - «`vect`» désignera la variable *vecteur* pour laquelle la méthode est invoquée;
    - «`base-type`» désignera le type de base d'un *vecteur*
18. Dans le cas où le sens de parcours est sans importance, on pourra avantageusement remplacer cette séquence par cette autre, plus efficace:
    `for (int i(vect.size()-1); i >= 0; --i) { ... }`

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Vecteur: méthodes de mise à jour

Modificateurs:

- `void` **`clear`**`()`: vide le vecteur, en supprimant tous ses éléments.
  Après l'invocation de cette méthode, le prédicat «`empty`» est forcément vrai.
- `void` **`pop_back`**`()`: supprime le dernier élément du vecteur.
- `void` **`push_back`**`(const base-type element)`: ajoute `element` à la fin du vecteur; `element` devient donc le nouveau dernier élément du vecteur.

Exemple: La boucle suivante initialise un vecteur d'entiers positifs, de taille 8, en demandant à l'utilisateur de saisir les valeurs initiales.
Lors de la saisie, l'utilisateur a la possibilité d'effacer la valeur précedemment saisie, en indiquant un chiffre négatif, ou d'effacer tout le vecteur, en entrant 0.

```
while (vect.size() < 8) {
   int val;
   cout << "Entrez coefficient " << vect.size() << ':' << flush;
   cin >> val;
   if (val < 0)  {vect.pop_back(); continue;}
   if (val == 0) {vect.clear(); continue;}
   vect.push_back(val);
}
```

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Vecteur: méthodes d'accès

Accès:

- base-type **front**(): renvoie une référence vers le premier élément du vecteur
  Les séquences «vect.front()» et «vect[0]» sont donc équivalentes.
  Précondition: le vecteur n'est pas vide (le prédicat «empty()» est faux).
  Exemple: L'itération suivante a comme effet de diviser tous les éléments d'un vecteur (à l'exception du premier), par le premier élément du vecteur.

```
for (int i(1); i<vect.size(); ++i)
{vect[i] /= vect.front();}
```

- base-type **back**(): renvoie une référence vers le dernier élément du vecteur
  Les séquences «vect.back()» et «vect[vect.size()-1]» sont équivalentes.
  Précondition: le vecteur n'est pas vide (le prédicat «empty()» est faux).
  Exemple: remplir un vecteur d'entiers, en demandant chaque valeur à l'utilisateur; le vecteur est considéré comme plein lorsque l'utilisateur entre la valeur 0.

```
do { vect.push_back(saisirEntier)
} while (vect.back());
```

Commande `typedef`

L'utilisation de variables dans un programme implique généralement que l'on spécifie plusieurs fois le type de ces variables (lors de la déclaration des variables, lors du prototypage des fonctions qui en font usage, lors de conversions, etc...)

Lorsque le type est complexe, sa définition peut être ardue, et est généralement longue, ce qui ne facilite pas la lecture du programme. En outre, les modifications éventuelles à apporter à la définition du type doivent être opérées sur chacune de ses occurences. Comme dans le cas des «blocs d'instructions réutilisables» (i.e. les fonctions), et pour les mêmes raisons, la duplication de la définition d'un type est à éviter.

La commande `typedef` permet pour cela de définir des *synonyme* (***alias***) de types, qu'ils soient fondamentaux ou composés:

```
typedef <type> <alias>;
```

Cette instruction permettra de désigner le type `type` indifféremment par `type`, ou au moyen de l'identificateur `alias` (`typedef` n'introduit pas de nouveau type, mais un nouveau nom pour le type)

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Commande typedef: exemple

```
typedef int longueur;
longeur diametre, rayon;
int nbCercles;
...
void  traceCercles(longueur d,
      longueur r, int nombre)
{ ... }
```

```
// pas de typdef //
int diametre, rayon;
int nbCercles;
...
void  traceCercles(int d,
      int r, int nombre)
{ ... }
```

Dans ces deux exemples, diametre, rayon et nbCercles sont tous de même type (int). Mais, si l'on utilise longueur pour exprimer toutes les longeurs (programme de gauche), et que pour une raison quelconque on est amené à changer la représentations des longueurs (p.ex. par des réels), il suffira d'opérer ce changement dans la définition de l'alias longueur, plutôt qu'à chaque occurence de int représentant une longueur.

```
typedef float longueur;
longeur diametre, rayon;
int nbCercles;
...
void  traceCercles(longueur d,
      longueur r, int nombre)
{ ... }
```

```
// pas de typdef //
float diametre, rayon;
int nbCercles;
...
void  traceCercles(float d,
      float r, int nombre)
{ ... }
```

typedef et types composés

Comme dans pratiquement tous les cas de types composés, la commande «**typedef**» permet de simplifier les déclarations (plus de lisibilité).

Par exemple dans la cas des tableaux (*built-in* ou *vecteurs*), en fournissant un alias pour le couple (*indication de tableau*, type de base):

```
typedef «type-base» «alias»[«taille»]

typedef vector<«type-base»> «alias»
```

Dans le cas des tableaux multidimensionnels, on peut se servir avantageusement de cette commande pour rendre plus explicite les déclarations et les usages ultérieurs:

```
typedef int Vecteur2[2];            // définis le type Vecteur2 comme un tableau de 2 entiers
typedef vector<int> GrilleLoto;     // et GrilleLoto comme un ensemble d'entiers
typedef Vecteur2 Matrice3x2[3];     // définit le type Matrice3x2 comme un tableau de 3 Vecteur2
typedef vector<GrilleLoto> BulletinLoto; // et BulletinLoto un ensemble de GrilleLoto
Matrice3x2 A = {{1,2},{3,4},{5,6}}
BulletinLoto fortune;
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

La classe «string»

Comme dans le cas des tableaux,
le type natif (*built-in*) fourni par le langage pour la représentation des chaînes de caractères (c'est-à-dire un tableau de taille fixe, avec le caractère `'\0'` comme marqueur de fin de chaîne) n'est pas toujours très pratique.

La librairie standard de C++ met donc à disposition du programmeur un nouveau type (en réalité une *classe*) dénommé ***string***, permettant une représentation efficace des chaînes, et offrant une large panoplie de fonctions nécessaires à la manipulation et au travail avec de telles chaînes:

- concaténation de chaînes,
- ajout/retrait d'un ou plusieurs caractères,
- recherche de sous-chaînes,
- substitution de sous-chaînes,
- ...

Pour pouvoir utiliser des *string* dans un programme, il faut importer les prototypes et définitions contenus dans la librairie au moyen de la directive d'inclusion:

```
#include <string>
```

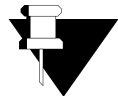
Déclaration d'une chaîne

Comme pour tout les autres types,
les variables correspondant à des chaînes de caractères
`string` doivent être déclarées; la syntaxe est:

`string` *«identificateur»*`;`

Exemple:

```
#include <string>
...
string chaine;
```



Les variables déclarées suivant la syntaxe indiquée ci-dessus sont automatiquement initialisées à une valeur correspondant à la chaîne vide.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Déclaration+initialisation d'une chaîne

La syntaxe d'une déclaration avec initialisation est:

```
string «identificateur»(«valeur»);
```

où *valeur* est:

- ⇨ soit une chaîne de caractères de la forme `"..."`
- ⇨ soit une variable, constante ou référence de type `string`

Exemple:

```
string str("une chaîne");
string chaine(str);
```

La valeur littérale à utiliser pour une initialisation explicite à une chaîne vide est: «`""`».

Affectation d'une chaîne

Toute variable de type `string` peut être modifiée par affectation:

«identificateur» **=** *«valeur»* **;**

où *valeur* est:

- ➯ soit un caractère (par exemple: «'i'»)
- ➯ soit une chaîne de la forme «"..."»
- ➯ soit une référence à une autre entité de type `string`

Exemple:

```
string chaine;                          // chaine  vaut « »
const string chaine2("test");           // chaine2 vaut «test»
chaine = 'c';                           // chaine  vaut «c»
chaine = string("str-temporaire");      // chaine  vaut «str-temporaire»
chaine = "built-in str";                // chaine  vaut «built-in str»
chaine = chaine2;                       // chaine  vaut «test»
```

Dans le cas de l'affectation par un caractère, la valeur affectée à la chaîne est une **chaîne de caractère** réduite à ce caractère.

Opérateurs relationnels entre chaînes

Les **opérateurs relationnels** suivant sont
définis pour les chaînes de type `string`:

Opérateur	*Opération*	
< <= > >=	strictement inférieur inférieur ou égal strictement supérieur supérieur ou égal	La relation d'ordre utilisée pour ces opérateurs est l'**ordre alphabétique**. (plus exactement, ordre **lexicographique**)
== !=	égalité différence (non-égalité)	

L'ordre alphabétique n'est pas respecté dans le cas des caractères accentués, et les majuscules sont considérées comme précédant les minuscules.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Accès aux caractères

Comme dans le cas des tableaux, il est possible d'accéder individuellement à chaque caractère constitutif d'une chaîne:

Si `str` est une chaîne de type `string`, l'expression `str[i]` fait référence au (i+1)$^{\text{ème}}$ caractère de `str`.

Chacun des éléments `str[i]` d'une chaîne `str` de type `string` est de type `char`.

Toujours comme dans le cas des tableaux, les éléments d'une chaîne sont indicés de 0 à (*taille* - 1), où *taille* est la longueur de la chaîne (c'est-à-dire son nombre de caractères).

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Concaténation de chaînes

La ***concaténation*** de chaînes de type `string`
est réalisée par l'opérateur « **+** »

L'expression « `chaine1 + chaine2` » correspond donc à une **nouvelle chaîne**,
dont la valeur littérale est constituée par la concaténation des valeurs
littérales de `chaine1` et de `chaine2`.

Les combinaisons suivantes sont possibles:

- *string* + *string*,
- *string* + *char*,
- *string* + `"..."`,
- *char* + *string*,
- `"..."` + *string*,

Méthodes de `string`: prédicats

Parmi les fonctions (*méthodes*)[19] disponibles, on trouve[20]:

Prédicats:

- `int size()`
  `int length()`

  Toutes deux renvoient la longueur de la chaîne (i.e. le nombre de caractères).

  Une manière usuelle pour parcourir un à un les caractères d'une chaîne est donc l'itération `for` suivante:

```
for (int i(0); i<str.size(); ++i) {
   // traitements avec «str[i]»
}
```

- `bool empty()`: indique si la chaîne est vide (ou non).

  On a l'équivalence suivante: «`str.empty()`» <=> «`(str.size() == 0)`»

19. Pour plus d'informations sur l'appel de tels fonctions, se référer au chapitre présentant les vecteurs.
20. Par convention, «`str`» désignera la variable de type *`string`* pour laquelle la méthode est invoquée

Méthodes de `string`: insertions

- ```
  string& insert(int pos, const char[] s)
  string& insert(int pos, const string& s)
  ```

  Insère, à partir de la position indiçée par `pos`, la chaîne `s`, et renvoie une référence à la chaîne modifiée.

  Exemple: «`string("1234").insert(2,"-xx-")`» utilise le premier des prototypes spécifiés ci-dessus, et renvoie une référence à la chaîne de type `string` et de valeur «`12-xx-34`».

- ```
  string& insert(int pos, const char[] s, int len)
  string& insert(int pos, const string& s, int start=0, int len=npos)
  ```
  [21]

  Insère, à partir de la position indiçée par `pos`, une sous-chaîne de `s` débutant à la position `start` (ou 0 dans le cas du premier prototype), et de longueur `len`. Une référence à la chaîne modifiée est retournée.

  Exemple: «`string("1234").insert(2,string("#$%-!"),3,1)`» utilise le second prototype[22], et renvoie une référence à la chaîne de type `string` et de valeur «`12-34`».

---

21. Une constante particulière, batisée «`string::npos`» permet en effet de représenter la notion «*jusqu'à la fin de la chaîne*».

22. Remarquons que grâce à la convertion automatique (casting) des chaînes de forme «"..."» vers le type `string`, la commande peut également s'écrire: «`string("1234").insert(2,"#$%-!",3,1)`».

Méthodes de `string`: remplacements

- `string& replace(int pos, int long, const char[] s)`
  `string& replace(int pos, int long, const string& s)`

  Substitue `s` au `long` caractères de la chaîne, à partir de la position indicée par `pos`. Renvoie une référence à la chaîne modifiée.

  Exemple: «`cout << string("1234").replace(2,1,"-trois-")`» produira comme résultat l'affichage de la chaîne «`12-trois-4`» : les caractères de l'intervalle [*pos*, *pos*+*long*-1], soit le caractère [2,2], sont remplacés par la chaîne `s`.

- `string& replace(int pos, int long, const char[] s, int len)`
  `string& replace(int pos, int long const string& s, int start, int len)`[23]

  Substitue la sous-chaîne de `long` caractères et débutant à la position `pos`, par la sous-chaîne de `s` de `len` caractères et débutant à la position `start` (ou 0 dans le cas du premier prototype). Une référence à la chaîne modifiée est retournée.

  Exemple: «`cout << string("1234").replace(2,1,"-trois-",1,5)`» produira comme résultat l'affichage de la chaîne «`12trois4`» : les caractères de l'intervalle [*pos*, *pos*+*long*-1], soit le caractère [2,2], sont remplacés par la sous-chaîne de s [*start*, *start*+*len*-1].

23. Avec les mêmes valeurs par défaut pour `start` et `l` que dans le cas de la méthode `insert`.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Méthodes de string: recherches

- ```
  int find(char s, int pos = 0)
  int find(const char[] s, int pos = 0)
  int find(const string& s, int pos = 0)
  ```

  Retourne l'indice du premier caractère de l'occurrence de s **la plus à gauche** dans la chaîne, éventuellement privée de ses pos premiers éléments (la recherche débute à partir de la position pos). Renvoie «string::npos» dans le cas où aucune occurrence n'est trouvée.

  Exemple: «string("baabbaabbaab").find("ba",2)» renverra 4

- ```
  int rfind(char s, int pos = npos)
  int rfind(const char[] s, int pos = npos)
  int rfind(const string& s, int pos = npos)
  ```

  Retourne l'indice du premier caractère de l'occurrence de s **la plus à droite** dans la chaîne, éventuellement privée de ses (*taille*-pos) derniers éléments (la recherche débute à partir de la position pos). Renvoie «string::npos» dans le cas où aucune occurrence n'est trouvée.

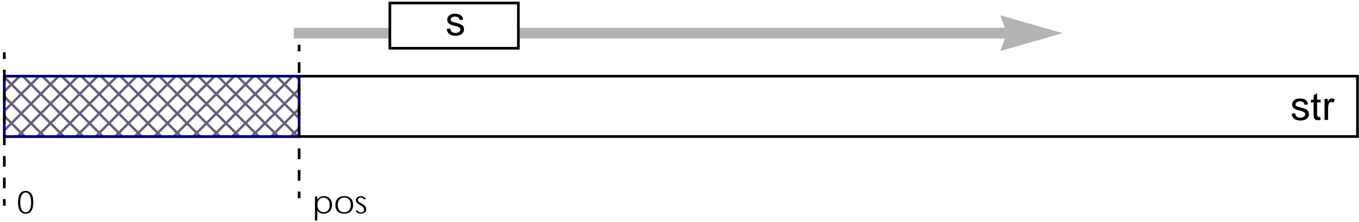
  Exemple: «string("baabbaabbaab").rfind("ba")» renverra 8

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Fonctionnement de `find` et `rfind`

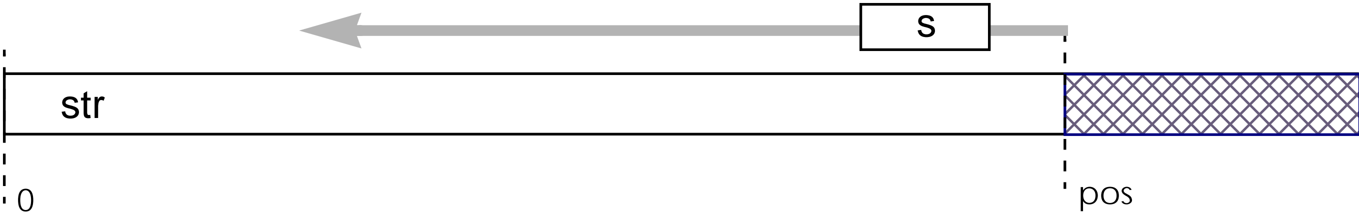
- La méthode «`find`» effectue une recherche du début vers la fin, en prenant comme début la position éventuellement spécifiée comme position de départ.

  Ainsi, «`str.find(s, pos)`» consiste à faire une recherche de `s` dans `str`, privée de ses `pos` premiers éléments:



- La méthode «`rfind`» effectue une recherche de la fin vers le début, en prenant comme fin la position éventuellement spécifiée comme position de départ.

  Ainsi, «`str.rfind(s, pos)`» consiste à faire une recherche «arrière» de `s` dans la sous-chaîne préfixe de `str` et de longueur `pos` + 1:



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Exemple d'utilisation des strings (1)

On désire réaliser une extension automatique des abbréviations dans un texte:

les CFF évoluent, c-à-d progressent.

les Chemin de Fer Fédéraux évoluent, c'est-à-dire progressent.

On utilise pour cela une table de correspondance entre les abbréviations (acronymes) et leur signification:

Acronyme	***Signification***
CFF	Chemin de Fer Fédéraux
c-à-d	c'est-à-dire
...	...

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Exemple d'utilisation des strings (2)

```
#include <string>
#include <vector>

// Définition des composants de la table associative
typedef vector<string> Strings;

// Table associative, globale (externe)
Strings acronymes;
Strings significations;

// Fonction de remplacement:
// utilise les variables externes définissant les acronymes,
// mais uniquement en lecture (pas d'effet de bord).
void expliciteAcronymes(string& str)
{
   int debutAbbrev;
   for (int i(0); i<acronymes.size(); ++i)
      while ((debutAbbrev=str.find(acronymes[i])) != string::npos)
         str.replace(debutAbbrev,acronymes[i].size(),significations[i]);
}
...
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Entrées-sorties en C++

En C++, les entrées-sorties sont gérées par le biais d'un ensembles de type (classes) spécialisés, les ***flots*** (*streams*).

L'idée de base des *flots* est de **séparer l'aspect logique** des entrées-sorties (i.e. leur intégration dans des programmes) **de leur aspect physique** (i.e. leur réalisation par le biais de périphériques particuliers).

Ainsi, un *flot* représente un **flux de données** entre le programme et un dispositif d'entrée-sorties externe (écran, imprimante, ...) ou un fichier.[24]

Comme nous l'avons déjà vu, les opérations de *sorties* sont réalisées à l'aide de l'***opérateur d'insertion*** « `<<` », tandis que celle d'*entrées* le sont à l'aide de l'***opérateur d'extraction***, « `>>` ».

Il existe un certain nombre de variables prédéfinies de type `stream`:

- **`cin`** : flot lié à l'*entrée standard*, qui par défaut est le clavier
- **`cout`** : flot lié à la *sortie standard*, qui par défaut est la console
- **`cerr`** : flot lié à la *sortie d'erreur standard*, qui par défaut est la console.[25]

24. Notons que pour obtenir de meilleures performances, un **tampon mémoire**, par lequel les données transitent, est par défaut associé à chaque flot.

25. Remarquons que `cerr` n'a pas de tampon mémoire: toutes les données introduites sont immédiatement acheminées vers la sortie d'erreur standard. Il n'est donc pas nécessaire, pour ce flot, de demander explicitement le vidage du tampon mémoire au moyen de la commande «`flush`».

Notion de persistance

Pour rendre **disponibles après son exécution** les données produites par un programme, l'affichage à l'écran (seule méthode vue jusqu'à présent) n'est naturellement pas adapté.

Il est par exemple nécessaire:

1) de rendre les données (résultant de l'exécution d'un programme) disponibles en tout temps pour l'utilisateur, et diffusables ou réutilisables par d'autre personnes ou d'autre programmes. Exemple: l'éditeur *XEmacs* rend «persistant» le fruit de vos efforts sous la forme d'un fichier texte, utilisé entre autre par le compilateur qui produira également un résultat réutilisable, sous la forme d'un fichier (binaire) exécutable[26].

2) dans le cas des programmes interactifs, permettre à l'utilisateur d'interrompre son travail (mettre momentanément fin à l'interaction avec le programme) et pouvoir le reprendre plus tard. Exemple: l'éditeur *XEmacs* qui vous permet de sauvegarder votre travail, et le poursuivre (reprendre) la semaine suivante[27].

Pour assurer une telle ***persistance*** des données informatiques, on utilise la plupart du temps le système de fichiers – mis à disposition par le système d'exploitation sous-jacent.

26. Dans ce cas, le format de «sauvegarde» des données (i.e. conventions de représentation, organisation) revêt une importance particulière: il doit soit être humainement compréhensible, soit correctement documenté ou respecter un standard, de manière à pouvoir être interprété par un autre logiciel.
27. Dans ce cas le format de représentation des données importe peu, et il n'est en particulier pas nécessaire de l'expliciter (on parle de *format interne*).

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Entrées sorties avec des fichiers (1)

Le lien entre les fichiers du système d'exploitation
et le programme se fait en C++ par le biais des ***streams***.

Un flot de données de type stream – orienté ou bidirectionnel – peut être *associé à un fichier*, et s'utilise de la même manière que les flots vers ou en provenance de la console (`cin`, `cout`, `cerr`) ou un string (`ostringstream`, `istringstream`):

Entrées sorties avec des fichiers (2)

Pour pouvoir utiliser les streams liés aux fichiers[28],
il faut importer, en début de programme, les prototypes et
définitions contenus dans la librairie, au moyen de la directive d'inclusion:

```
#include <fstream>
```

Deux types (classes) [en particulier] sont alors disponibles:

⇨ **`ifstream`** (*input file stream*)
stream d'entrée associé à un fichier ouvert en mode *lecture*
(similaire à `cin`)

⇨ **`ofstream`** (*output file stream*)
stream de sortie, associé à un fichier ouvert en mode *écriture*
(similaire à `cout`).

28. Les streams de la librairie standard ne véhiculent que des données en mode texte. Pour lire/écrire des fichiers binaires, il faut soit utiliser des fonctions *ad hoc* spécifiques, soit étendre «manuellement» la bibliothèques de streams.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Entrées sorties avec des fichiers (3)

Le mécanisme général pour la mise en œuvre d'entrée-sorties via les fichiers est:

(1) **Création** d'un stream (d'entrée ou de sortie),
par la déclaration d'une variable de type `ifstream` ou `ofstream`.

Exemple:
```
ifstream entree;
ofstream sortie;
```

(2) **Association** de la variable **avec un fichier** physique,
par l'appel de la fonction-méthode `open`, en respectant la syntaxe d'appel définie dans le paragraphe concernant les vecteurs.

Exemple:
```
entree.open("input.txt");
```
qui associe le stream `entree` avec le fichier nommé «input.txt» (présent dans le répertoire courant).

(3) **Utilisation** du stream: par des appels à des fonctions-méthodes, et l'utilisation des opérateurs d'insertion («>>») et d'extraction («<<») de données.

(4) **Fermeture** du stream,
au moyen de la fonction-méthode `close`

Exemple:
```
entree.close();
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Entrées sorties avec des fichiers (4)

Les opérations (1) et (2), à savoir la déclaration de la variable et son association avec un fichier, peuvent se réaliser en une seule étape.

L'initialisation lors de la déclaration se fait alors en spécifiant entre parenthèses le nom du fichier à lier au stream:

Exemple:

```
ifstream entree("input.txt");
```

On peut considérer que l'initialisation fait directement appel à la fonction `open`.

La fonction `open` admet comme argument une chaîne de caractères de type `"..."` (en fait un `char[]`): il n'est donc pas possible d'utiliser directement une chaîne de type `string`[29]. Il faut dans ce cas demander la conversion en type `char[]` du `string`, en faisant appel à la fonction-méthode `c_str`:

Exemple:

```
string str("output.txt");
ifstream entree2;
ofstream sortie(str.c_str());
entree2.open(str.c_str());
```

29. En effet, s'il y a conversion implicite des chaînes de la forme `"..."` vers les *strings* (promotion), l'inverse n'est pas vrai.

Entrées sorties avec des fichiers (5)

L'utilisation dans le programme des variables de type *fstream* pour effectuer des entrées-sorties se fait de la même manière que pour les streams `cin` et `cout`, c'est-à-dire à l'aide des opérateurs d'insertion «>>» et d'extraction «<<»

Exemple:

```
ifstream entree("input.txt");
ofstream sortie("output.txt");
string mot; int n;

entree >> mot >> n;
sortie << "Le mot lu est: '" << mot << "' et le "
       << "nombre est: '" << n << "'." << endl;
```

Remarque:

Une fonction utile pour tester si le lecture d'un fichier est terminée est la fonction-méthode `eof`.

```
if (entree.eof()) {
    cout << "Fin du fichier" << endl;
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Exemple d’écriture dans un fichier

Le programme ci-dessous saisit une phrase via l’entrée standard (clavier), et l’écrit (un mot par ligne) dans un fichier texte nommé «phrase.txt»

```
#include <iostream>
#include <fstream>
#include <string>

void main()
{
   string motSaisit;
   string nomFichier("phrase.txt");
   ofstream sortie(nomFichier.c_str());

   cout << "Entrez une phrase terminée par « .»" << endl;
   do
   {
      cin >> motSaisit;
      sortie << endl << motSaisit;
   } while (motSaisit != ".");
   sortie.close();
}
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Exemple de lecture à partir d'un fichier

Le programme ci-dessous relit le fichier créé par le programme précédant, et affiche la phrase à l'écran, et remettant les mots sur une même ligne.

```
#include <iostream>
#include <fstream>
#include <string>

void main()
{
   string motSaisit;
   ifstream entree("phrase.txt");

   while (!entree.eof())
   {
      entree >> motSaisit;
      cout << motSaisit << ' ';
   }
   cout << endl;
   entree.close();
}
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Robustesse des accès aux fichiers

De même qu'il est conseillé de contrôler la conformité des valeurs entrées par un utilisateur, il est bon de vérifier que l'association d'un stream avec un fichier s'est correctement réalisée (i.e. que le fichier existe, est lisible, etc).

Pour cela, on peut «évaluer le stream», après l'appel à la fonction `open`. Si tout c'est bien passé, le résultat de cette évaluation sera `true`, et il sera `false` en cas de problème:

Exemple:

```
entree.open("fichier-inexistant");
if (!entree) cout << "Oops, le fichier n'est pas lisible!" << endl;
else << cout << "Ok, le fichier existe et est lisible!" << endl;
```

Plus précisément, un certain nombre de ***prédicats*** sont associés aux *streams*, et permettent d'en connaître l'état:[30]

`bool good()`	*le stream est dans un état correct, la prochaine opération (lecture/écriture) sera un succès*
`bool eof()`	*la fin du stream à été atteinte*
`bool fail()`	*la prochaine opération (lecture/écriture) échouera*
`bool bad()`	*le stream est corrompu: des données ont été perdues (et la prochaine opération échouera)*

30. L'«évaluation» d'un *stream* revient en partivulier à tester le prédicat `fail`.

Strings et Streams

Il peut parfois être utile, en vue d'effectuer des traitements particuliers, de disposer d'une **représentation sous forme de chaîne de caractères** des différentes variables d'un programme.

Une telle convertion de représentation est automatiquement réalisée lorsque l'on insère ces éléments dans un *stream* de sortie, tel que `cout`.

Mais peut-on récupérer ce qui est transmis dans le flot ?

A défaut de récupérer ce qui est effectivement transmis, on peut du moins en obtenir l'équivalent sous forme de string, en utilisant des strings hybrides, associés à des streams.

Ces entités hybrides sont désignées « **`stringstream`** ».

Pour pouvoir utiliser les *stringstream* dans un programme, il faut importer les prototypes et définitions contenus dans la librairie, au moyen de la directive d'inclusion:

```
#include <sstream>
```
[31]

31. La version de *gcc* utilisée dans le cadre du cours utilise une ancienne représentation de ces entités, pas tout à fait compatible avec la nouvelle norme. Toutefois, une adaptation a été écrite pour le besoin du cours, et placée dans le répertoire de la librairie standard.

EPFL — ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

stringstream de sortie

Pour obtenir la ***représentation alphanumérique*** d'une variable, il faut utiliser un **`stringstream` de *sortie***, dans lequel on insérera les données désirées (de la même manière qu'avec `cout`) avant d'en extraire l'équivalent sous forme de chaîne de caractère.

Le type (la classe) matérialisant de tels éléments est désigné: **`ostringstream`**, et l'extraction de la représentation chaîne s'obtient au moyen de la méthode:

```
string str()
```

Exemple:

```
#include <sstream>
...
string composeMessage(const int errno, const string& description)
{
   ostringstream ost;
   ost << "Erreur (n°" << errno << "): " << description;
   return ost.str();
}
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

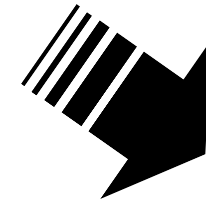
stringstream d'entrée

Il est également possible de réaliser des entrées depuis un `string` (au lieu du clavier, avec `cin`): il suffit d'initialiser un **`stringstream` d'*entrée*** avec la chaîne désirée, et d'utiliser l'opérateur d'extraction sur le *stream* ainsi obtenu.

Le type (la classe) utilisé est: **`istringstream`**.

Exemple:

```
#include <sstream>
...
// Extrait les mots d'un string, et les affiches à
// raison de un par ligne
void wordByWord(const string& str){
   istringstream ist(str);
   string s;
   while (ist >> s) {cout << s << endl;}
}
   // en utilisant la fonction précédante:
   wordByWord(composeMessage(5, "Fichier non trouvé"));
}
```



```
Erreur
(n°5):
Fichier
non
trouvé
```

Sorties formatées (1)

Un certain nombre de paramètres pour le format des sorties peuvent être explicitement spécifiés.

Ces paramètres sont de deux formes:

- ⇨ Soit des ***manipulateurs***, appliqués à l'opérateur « << »
- ⇨ Soit des ***options de configurations***, appliqués directement au stream de sortie

Manipulateurs:

Pour utiliser les manipulateurs, il faut au préalable importer les définitions du fichier `iomanip` au moyen de la directive d'inclusion:

```
#include <iomanip>
```

On utilise les manipulateurs très simplement, en les insérant directement dans le flot, à l'endroit désiré, exactement comme avec les données:

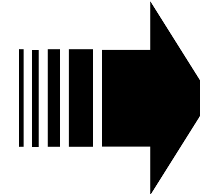
```
cout << «donnée» << «manipulateur» << «donnée» << ...
```

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Sorties formatées (2)

- Longueur de la représentation: `setw(«taille»)`
  la donnée qui suit ce manipulateur est affiché sur (au moins) *taille* caractères, avec (par défaut) un cadrage à droite, fort pratique lors de la représentation en colonnes des nombres.

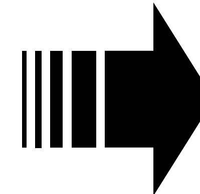
```
cout << setw(10) << "un:"
     << setw(5) << 1 << endl;
cout << setw(10) << "cent deux:"
     << setw(5) << 102 << endl;
cout << setw(10) << "moins 18:"
     << setw(5) << -18 << endl;
```



```
       un:    1
cent deux:  102
 moins 18:  -18
```

- Caractère de remplissage: `setfill(«char»)`
  définit le caractère utilisé pour pour réaliser le remplissage lors d'un alignement (par défaut le caractère espace):

```
cout << setw(10) << "un:"
     << setfill('.') << setw(5)<< 1;
cout << endl << setw(10) << "cent deux:"
     << setfill('.') << setw(5) << 102;
cout << endl << setw(10) << "moins 18:"
     << setfill('.') << setw(5) << -18;
```



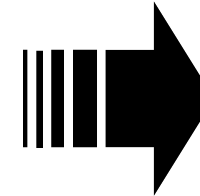
```
       un:....1
cent deux:..102
 moins 18:..-18
```

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Sorties formatées (3)

☞ Bases numérique: `hex, dec, oct`
les données numériques sont affichées ou lues en assumant une base de représentation différente de la base par défaut (décimale).

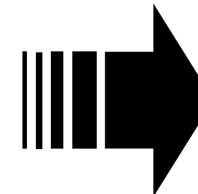
```
cout << "hexa:" << hex << 64 << endl;
cout << "octal:" << oct << 64 << endl;
cout << "decim:" << dec << 64 << endl;
```



```
hexa:40
octal:100
decim:64
```

☞ Précision en virgule flottante: `setprecision(int)`
définit le nombre de chiffres significatifs sur lequel se fera l'affichage des nombre en virgules flottantes (par défaut 6):

```
for (int i(2); i<6; ++i)
{
   cout << setprecision(i) << i
        << ' ' << 314.1592654 << endl;
}
```



```
2 3.1e+02
3 314
4 314.2
5 314.16
```

EPFL ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Sorties formatées (4)

Options de configuration:

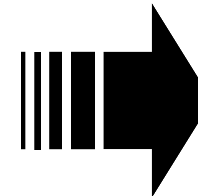
Les options définies pour les streams de sorties sont configurées à l'aide de la fonction (méthode) `setf`, selon la syntaxe:

`«flot».setf(ios::«option»);`

Quelques options:

Infos sur la base de représentation: `showbase`
la base de représentation du nombre est affichée, en suivant les conventions de représentation des nombre littéraux dans les programmes C++:

```
cout.setf(ios::showbase);
cout << "hexa:" << hex << 64 << endl;
cout << "octal:" << oct << 64 << endl;
cout << "decim:" << dec << 64 << endl;
```



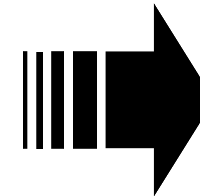
```
hexa:0x40
octal:0100
decim:64
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Sorties formatées (5)

☞ Partie fractionnaire: `showpoint`
la partie fractionnaire des nombres en virgule flottante est explicitement représentée:

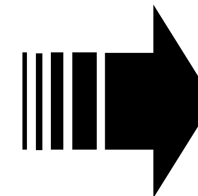
```
cout << 16.000 << endl;
cout.setf(ios::showpoint);
cout << 16.000 << endl;
cout << 16.100 << endl;
cout << 16 << endl;
```



```
16
16.0000
16.1000
16
```

☞ Représentation fixe/scientifique: `fixed, scientific`
détermine un affichage en mode fixe (traditionnel) ou scientifique (en puissance de 10, avec 1 nombre devant la virgule):

```
const float root(sqrt(200));
cout.setf(ios::fixed);
cout << root << endl;
cout.unsetf(ios::fixed);
cout.setf(ios::scientific);
cout << root << endl;
```



```
14.142136
1.414214e+01
```