Le langage et ses traductions
Axiomatique impérative

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 22 mai 2018

Table des matières

C++ - Le langage et ses traductions

Objectif

Ce module donne la syntaxe et la description sémantique de l'axiomatique impérative dans le langage algorithmique et ses traductions dans divers langages de programmation.

1 Le langage

1.1 Le langage

Règles, Alphabet du langage, Éléments lexicaux

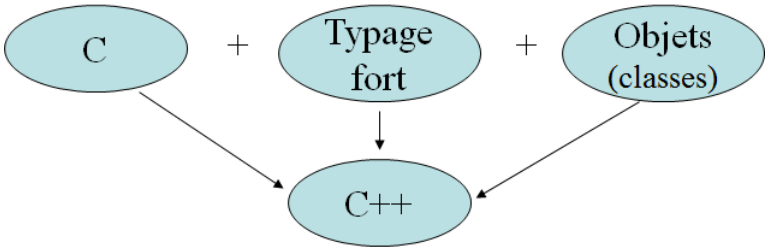
Tout **langage** de programmation possède :

- Des **règles de présentation** (aspect lexical).
- Des **règles d'écriture** (une syntaxe).
- Des **règles d'interprétation** (une sémantique).

L'**alphabet du langage** permet de construire les mots et les expressions. Comme toute phrase d'un langage courant, il est composé de mots appelés **lexèmes** ou **éléments lexicaux** (*tokens*).

Le langage C++

Langage de programmation issu de C, *orienté objet* (OO), *compilé* (impératif) et *fortement typé*.



C/C++

Alphabet du langage

C'est celui de la langue anglaise (majuscules et minuscules), les chiffres 0-9 et quelques symboles spéciaux (opérateurs et ponctuations). Étant un langage anglophone, les lettres accentuées ou exotiques ne sont pas autorisées dans les lexèmes. Il est en revanche possible de les utiliser dans les chaînes de caractères mais, ces caractères exotiques étant codés différemment selon les systèmes d'exploitation, ces chaînes risquent de ne plus être compréhensibles lors du portage du fichier sur un autre système.

C/C++

La casse

Le langage est *case-sensitif* (sensible à la casse) [1] et les accentués ne sont pas des lettres anglophones. Ainsi `cout`, `Cout` et `COUT` réfèrent trois mots différents et `coût` n'est pas autorisé.

1.2 Les mots-clés

Mots-clés (*keywords*)

Dits aussi **mots réservés**, ils constituent le vocabulaire du langage. Ils forment l'ossature d'un programme en définissant la structure de ses données ou de ses instructions.

1. Il fait la distinction entre minuscules/majuscules.

Propriété
Les mots-clés ont une **signification inchangeable et prédéfinie**, c.-à-d. qu'ils ne peuvent pas être redéfinis.

Liste des mots-clés
(1) Ansi C (2) C++ (3) Représentations alternatives d'opérateurs

```
(1) auto break case char const continue default do double else enum extern float for
    goto if int long register return short signed sizeof static struct switch typedef
    union unsigned void volatile while
(2) asm bool catch class const_cast delete dynamic_cast explicit export false friend
    inline mutable namespace new operator private protected public reinterpret_cast
    static_cast template this throw true try typeid typename using virtual wchar_t
(3) and (&&) and_eq (&=) bitand (&) bitor (|) compl (~) not (!) not_eq (!=) or (||)
    or_eq (|=) xor (^) xor_eq (^=)
```

Environnements IDE
Les mots-clés y sont en gras (ou en couleurs).

1.3 Les identifiants

Identifiant
Dit aussi **identificateur**, c'est un **nom** choisi par le concepteur pour désigner les entités : variable, fonction, type, etc. Cette définition reste valable dans son domaine de validité, à condition qu'elle ne soit pas recouverte par une autre définition. Il **ne doit pas** correspondre à un mot-clé.

Identifiant
Séquence de lettres (A...Z, a...z), de chiffres (0...9) ou du caractère souligné (_, *underscore* en terme anglo-saxon). Il doit commencer par une lettre ou un souligné.

Le souligné est significatif
Ainsi `a`, `a_` et `_a` sont des identifiants différents.

1.4 Séparateurs et espaces blancs

Espace blanc
Suite de un ou plusieurs caractères choisis parmi : espace, tabulation horizontale, tabulation verticale, changement de ligne ou changement de page.

Séparateurs
Comprend les opérateurs (`+`, `-`, `*`, `/`...) et les caractères de « ponctuation ».

Quelques séparateurs

	Symbole	Rôle
`{}`	accolades	délimitent les blocs
`()`	parenthèses	encadrent des expressions
`[]`	crochets	symboles du composant tableau
`.`	point	accès à un composant structurée
`,`	virgule	sépare les éléments d'une liste
`;`	point-virgule	sépare les instructions
`' '`	quotes	encadrent les caractères
`" "`	guillemets	encadrent les chaînes de caractères

1.5 Types intégrés

Types intégrés (*builtins*)

Dits aussi **types de base**, **types fondamentaux** ou encore **types primitifs**, ils correspondent aux données qui peuvent être traitées directement par le langage.

Types intégrés

Domaine	Algorithmique	Équivalent C++
$\mathbb{Z}$	`Entier`	`int`
$\mathbb{R}$	`Réel`	`double`
$\mathbb{B}$	`Booléen`	`bool`
$\mathbb{A}$	`Caractère`	`char`
$\mathbb{T}$	`Chaîne`	`#include <string>` `string`

C++ : Le type Chaîne

Ce n'est pas un type élémentaire.
On utilisera néanmoins le type `string` qui le défini.

1.6 Littéraux

Littéral

Valeur explicite au sein d'un programme.
Il possède *un type* déterminé par sa valeur (entier, réel, caractère...).

Notation scientifique d'un réel

Expression « m e p » qui signifie $m \cdot 10^p$ (où `m` est une suite de chiffres décimaux et `p` un entier). Ainsi, $12.4e-5 \equiv 12.4 \cdot 10^{-5} \equiv 0.000124$. Elle permet d'exprimer de très petits nombres (ex : 2.5e-201) et de très grands nombres (ex : 5e156).

Point décimal

Un réel se distingue d'un entier par l'ajout d'un point (.) à la fin.
Pour la lisibilité, préférez .0 (ex. 5.0 (réel) mais 5 (entier)).

C/C++

Littéraux

- `Entier` : Suite de chiffres éventuellement préfixé par un signe (+ ou −). S'il y a deux chiffres ou plus, il **ne doit pas** commencer par 0.
- `Réel` : S'écrit en notation décimale ou en notation scientifique.
- `Booléen` : Ils identifient le `Vrai` (mot-clé `true`) et le `Faux` (mot-clé `false`).
- `Caractère` : Se place entre quotes (').
- `Chaîne` : Se place entre guillemets (").

2 Structure d'un algorithme

2.1 Structure générale

Structure générale

```
#include <des_trucs_utiles>
using namespace std;
déclaration_des_objets_globaux
déclarations_et_définitions_de_fonctions_utiles
int main()
{
  corps_du_programme
}
```

Explication

Un programme est constitué par :

- Un **en-tête** qui demande au préprocesseur d'inclure les fichiers indiqués.
- Un **corps** lequel contient la fonction particulière `main()` (« principale ») nécessaire pour produire du code machine *exécutable*.

Le programme commence son exécution sur l'accolade ouvrante de la fonction `main`, se déroule séquentiellement et se termine sur son accolade fermante.

C/C++ : La fonction main

Quelques règles :

- Respectez la casse (m minuscule) ainsi que les parenthèses.
- Chaque programme possède une fonction `main()`.
- En l'absence de fonction `main()`, le programme ne démarre pas.

Conseil

On veillera à ce qu'un programme tienne sur une vingtaine de lignes (donc, en pratique, sur un écran de 40 x 80 caractères ou une page). Ceci implique que, si votre programme devait être plus long, il faudra le découper, comme nous le verrons plus loin.

2.2 Commentaire

Commentaire (narratif)

Texte qui n'est **ni lu, ni exécuté** par la machine. Il est essentiel pour rendre plus lisible et surtout plus compréhensible un programme par un être humain.

C99/C++ : Commentaire orienté ligne

```
... // rend le reste de la ligne non-exécutable
```

Commentaire orienté bloc

```
/*
rend le code entouré non exécutable...
(hérité du C)
*/
```

2.3 Inclure un fichier

C++ : Inclure un fichier

```
#include "NomFichier"
```

Explication

Insère le fichier `NomFichier` à la place de la directive `#include`. Par défaut, l'extension du fichier est ".hpp".

3 Déclarations

3.1 Déclaration de variables

Déclaration de variables

Consiste à associer un type de données à une ou un groupe de variables. Toute variable doit impérativement avoir été déclarée avant de pouvoir figurer dans une instruction exécutable.

C/C++

Déclaration de variables

```
TypeVar nomVar;
TypeVar nomVar1, nomVar2, ...;
```

Explication

Déclare des variables d'identifiants `nomVarI` (le nom) de type `TypeVar`.

3.2 Définition de constante

Constante

Littéral à lequel est associé un **identifiant** (par convention, écrit en MAJUSCULES) afin de rendre plus lisible et simplifier la maintenance d'un programme. C'est donc une information pour laquelle nom, type et valeur sont figés.

Définition de constante

```
const TypeConst nomConst = expression; // notation impérative
const TypeConst nomConst(expression); // notation objet
```

Explication

Définit la constante d'identifiant `nomConst` de type `TypeConst` et lui affecte une valeur (littéral ou expression) spécifiée.

Valeur immuable fixée à la déclaration

Toute tentative de modification est rejetée par tout compilateur qui signalera une erreur (ou un avertissement).

3.3 Définition d'une énumération

Type énuméré

Définit un ensemble de constantes entières associées une à une à des identifiants ou *énumérateurs*. Deux avantages :

- Une indication claire des possibilités de la variable lors de la déclaration.
- Une lisibilité du code grâce à l'utilisation des valeurs explicites.

Définition d'une énumération

```
enum NomType { nomVal1 [= valeur1], nomVal2 [= valeur2] ... };
```

Explication

Introduit `NomType` dont les valeurs discrètes sont `nomVal1`, `nomVal2`, etc. Par défaut, `valeur1` vaut 0 et les suivantes sont incrémentées de 1 par rapport à la précédente.

Remarque

Un type énuméré ne peut ni être saisi, ni affiché directement par les fonctions standards du langage ; en revanche, il peut être affiché sous une forme entière.

Quid des langages de programmation ?

Chaque langage de programmation propose sa propre technique de conversion de valeurs.

- Certains langages (comme Java) proposent un type énuméré complet.
- D'autres (comme C et C++) proposent un type énuméré incomplet mais qui permet néanmoins une écriture comme celle ci-dessus.
- D'autres langages ne proposent rien. Pour ces derniers, l'astuce est de définir des constantes entières qui vont permettre une écriture proche de celle ci-dessus (mais sans une déclaration explicite).

3.4 Synonyme de type

Synonyme de type

Alias d'un type existant (lorsqu'un nom de type est trop long ou est difficile à manipuler).

Synonyme de type

```
using TypeAbrege = TypeExistant;
```

Explication

Désigne l'identifiant `TypeAbrege` comme étant un synonyme du type `TypeExistant`.

Typedef = Définition

`using` N'introduit pas de nouveau type mais un **nouveau nom** pour le type.

4 Instructions élémentaires

4.1 Primitives

Bibliothèque
Ensemble de fonctionnalités ajoutées à un langage de programmation.
Chaque bibliothèque décrit un thème.

Pour les utiliser

```
#include <nomBiblio>
using namespace std;
```

Primitives
Noms de fonctions (`abs`, `log`, `sin`...), d'opérateurs (`div`, `mod`...) ou de traitement (`afficher`, `saisir`...). Elles acceptent un ou plusieurs paramètres et jouent le même rôle syntaxique qu'un identifiant.

Appel d'une primitive

```
P(x,...); // procédure
r = F(x,...) // fonction
```

Explication
Appelle (on dit aussi **invoque**) la procédure `P` ou la fonction `F` avec les arguments `x`... Dans le cas de fonction, la valeur retournée peut être utilisée en tant que macro-expression.

4.2 Saisie de données

Saisie de données

```
cin >> nomVar1 >> nomVar2 >> ... >> nomVarN;
```

Explication
Ordonne à la machine de lire des valeurs `valI` depuis le clavier et de les stocker dans les variables `nomVarI` (qui doivent exister c.-à-d. déclarées). L'identifiant `cin` désigne la variable associée à l'« entrée standard ».
Prononcez : c-in (console input).

Remarque
Par défaut, ce qui est tapé au clavier est envoyé à l'écran et temporairement placé dans un tampon pour permettre la correction d'erreurs de frappe. On peut donc se servir des touches [←] et [Suppr] pour effacer un caractère erroné ainsi que les flèches [<] et [>] pour se déplacer dans le texte.

C/C++ : Nature de la donnée saisie
Le langage se met en boucle si la nature de la donnée saisie ne correspond pas à celle de la variable affectée.

C/C++ : Caractère séparateur
S'il y a plusieurs valeurs à saisir, séparez-les par un espace ou une tabulation. Les valeurs entrées ne sont affectées aux variables que lorsque l'usager appuie sur la touche [Entrée].

4.3 Affichage de résultats

Affichage de résultats

```
cout << expr1 << expr2 << ... << exprN; // SANS retour de ligne
cout << expr1 << ... << exprN << endl;  // AVEC retour de ligne
```

Explication
Ordonne à la machine d'afficher les **valeurs** des expressions `exprI`. Par défaut, elles ne sont pas séparées par des espaces. Ajoutez le(s) délimiteur(s) si nécessaire.
L'identifiant `cout` désigne la variable associée à la « sortie standard ».
Prononcez : c-out (console output)

C++ : Le manipulateur endl
Il provoque le passage à la ligne suivante et ordonne le vidage du tampon d'affichage.

Variables devant être initialisées
On ne peut *afficher* que des expressions dont les variables qui la composent ont été affectées préalablement.

4.4 Formats d'édition

Format d'édition d'une expression
Indique de quelle façon doit être cadrée l'expression à afficher. Il s'applique aux valeurs de type `Chaîne`, `Entier` ou `Réel`.

Formats d'édition

```
#include <iomanip>
cout << setw(largeur1) << exprChaine;
cout << setw(largeur1) << exprEntier;
cout << fixed << setprecision(largeur2) << setw(largeur1) << exprReel;
```

Explication
Définit le format d'édition. L'entier `largeur1` indique sur combien de caractères doit être écrite l'expression. L'entier `largeur2` précise le nombre de chiffres après le point décimal des réels. Les manipulateurs `fixed` (format point-décimal) et `setprecision` (nombre de décimales) sont définis dans la bibliothèque `<iomanip>`.

Règle d'affichage
Si le format est :

- **Égal** à la longueur nécessaire à l'édition de la valeur : la valeur est écrite telle quelle.
- **Inférieur** à la longueur : il est ignoré et la valeur est écrite sur la longueur nécessaire.
- **Supérieur** à la longueur : le système effectue un cadrage de la valeur à afficher à l'intérieur du format qui lui a été spécifié. Les données numériques sont cadrées à droite sur le point décimal, et les données alphanumériques cadrées à gauche.

4.5 Affectation interne

Affectation interne
Opération qui **fixe une valeur** à une variable.

Affectation interne

```
nomVar = expression;
```

Explication
Place la valeur de l'`expression` dans la zone mémoire de la variable de nom `nomVar`. En algorithmique, le symbole <- (qui se lit « devient ») indique le sens du mouvement : de l'expression située à droite **vers** la variable à gauche.

Rappel
Toutes les variables apparaissant dans l'`expression` doivent avoir été affectés préalablement. Le contraire provoquerait un arrêt de l'algorithme.

Conversions implicites
Il est de règle que le résultat de l'expression à droite du signe d'affectation soit de même type que la variable à sa gauche. En C/C++, on tolère certaines exceptions.

- `Réel --> Entier` : Le contenu de la variable sera la valeur **tronquée** de l'expression réelle.
- `Entier --> Réel` : Conversion d'un entier en réel.
- `Caractère --> Chaîne` : L'expression caractère est transformée en une chaîne de taille 1. Le contraire n'est évidemment pas accepté.

4.6 Instruction composée

Instruction

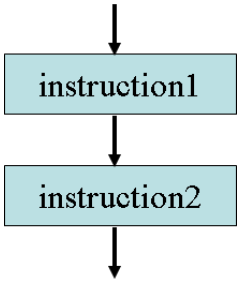
Ordre donné à l'ordinateur qui a pour effet de changer l'état de la mémoire ou le déroulement du programme ou bien de communiquer avec les unités périphériques (clavier, écran, imprimante, etc.).

Instruction composée ou Bloc

Regroupement syntaxique de 0, 1 ou plusieurs instructions (et déclarations) comme une unique instruction.

Séquentialité

Les algorithmes et programmes présentés sont exclusivement séquentiels : l'`instruction2` ne sera traité qu'une fois l'exécution de l'`instruction1` achevée.



C/C++ **Bloc**

```
{
  instruction1;
  instruction2;
  ...
}
```

Conventions usuelles

Savoir présenter un programme, c'est montrer que l'on a compris son exécution.

- Chaque ligne comporte une seule instruction.

  C/C++ Le point-virgule « ; » est le terminateur d'instructions.
- Les indentations sont nécessaires à sa bonne lisibilité. Ainsi :

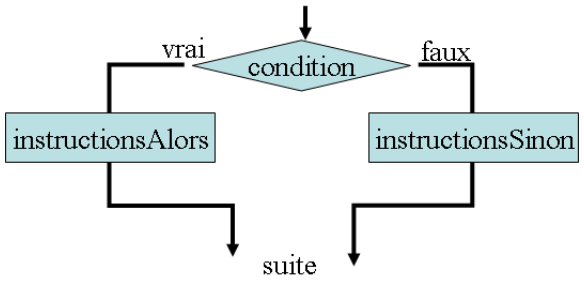
  C/C++ Alignez les accolades de début `{` et fin `}` de bloc l'une sous l'autre.

5 Structures conditionnelles

5.1 Sélective Si

Sélective Si

Elle traduit : `Si` la `condition` est vraie, exécuter les `instructionsAlors`, `Sinon` exécuter les `instructionsSinon`. Il s'agit d'un choix binaire : **une et une seule** des deux séquences est exécutée.



La `condition` peut être simple ou complexe (avec des parenthèses et/ou des opérateurs logiques `Et`, `Ou`, `Non`).

C/C++ **Sélective Si**

```
if (condition)
{
  instructionsAlors;
}
else
{
  instructionsSinon;
}
```

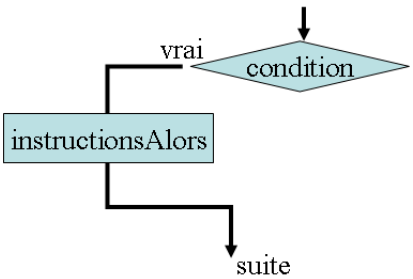
C/C++

Notez l'absence du mot-clé `Alors` d'où l'obligation des parenthèses autour de la condition.

5.2 Sélective Si-Alors

Sélective Si-Alors

Forme restreinte de la structure `Si` (sans clause `Sinon`).



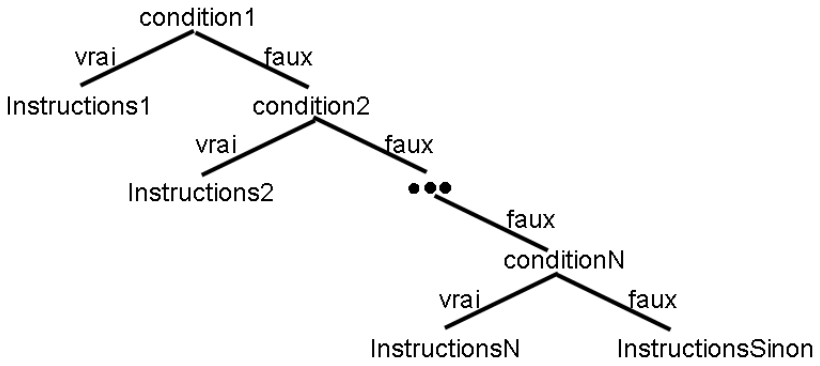
C/C++ **Sélective Si-Alors**

```
if (condition)
{
  instructionsAlors;
}
```

5.3 Sélective Si-Sinon-Si

Sélective Si-Sinon-Si

Elle évalue successivement la `conditionI` et exécute les `instructionsI` si elle est vérifiée. En cas d'échec des `n` conditions, exécute les `instructionsSinon`.



C/C++ **Sélective Si-Sinon-Si**

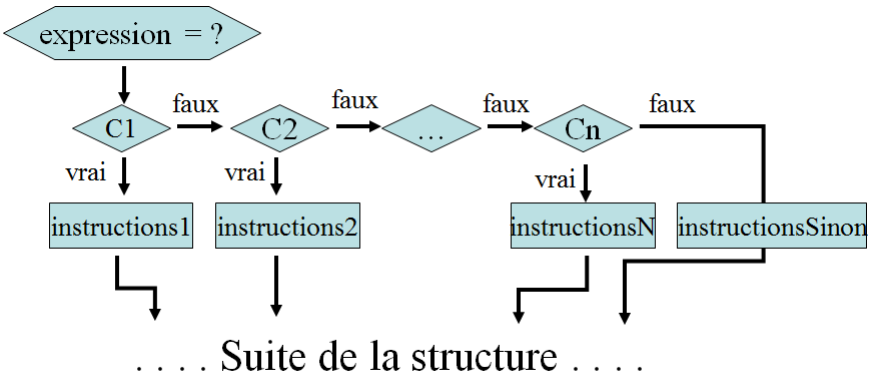
```
if (condition1)
{
  instructionsA1;
}
else if (condition2)
{
  instructionsA2;
}
else if...
...
else if (conditionN)
{
  instructionsAn;
}
else
{
  instructionsSinon;
}
```

5.4 Sélective Selon (listes de valeurs)

Sélective Selon (listes de valeurs)

Elle évalue l'`expression` et n'exécute que les `instructionsI` qui correspondent à la valeur ordinale `Ci` (c.-à-d. de type entier ou caractère). La clause `Cas Autre` est facultative et

permet de traiter tous les cas non traités précédemment. Il s'agit de l'instruction multi-conditionnelle classique des langages.



Sélective Selon (listes de valeurs)

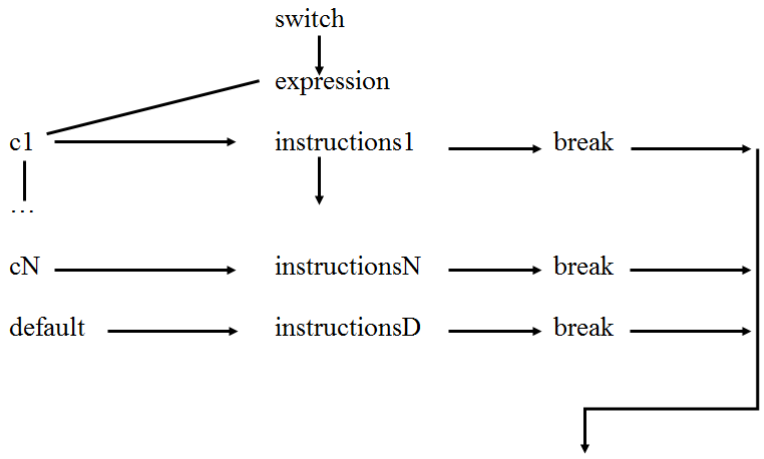
```
switch(expression)
{
  case C1:
    instructions1;
    break;
  ...
  case Cn:
    instructionsN;
    break;
  default:
    instructionsD
}
```

Remarque

Veillez à ne pas faire apparaitre une même valeur dans plusieurs listes.

C/C++ : Rupture

L'achèvement d'un énoncé n'est pas automatique : il faut l'expliciter à l'aide de l'instruction `break`.



Selon v.s. Si

Le `Selon` est moins général que le `Si` :

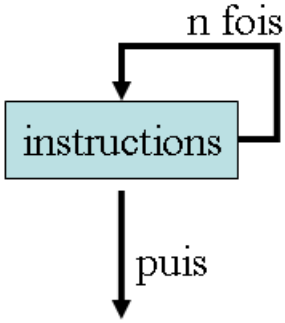
- L'expression doit être une valeur discrète (`Entier` ou `Caractère`).
- Les cas doivent être des *constantes* (pas de variables).

Si ces règles sont vérifiées, le `Selon` est plus efficace qu'une série de `Si` en cascade (car l'expression du `Selon` n'est évaluée qu'une seule fois et non en chacun des `Si`).

6 Structures répétitives

6.1 Répétitive Itérer

Répétitive Itérer
Elle traduit : Exécuter `n` fois les `instructions`, avec `n` un entier positif.
Finitude assurée.

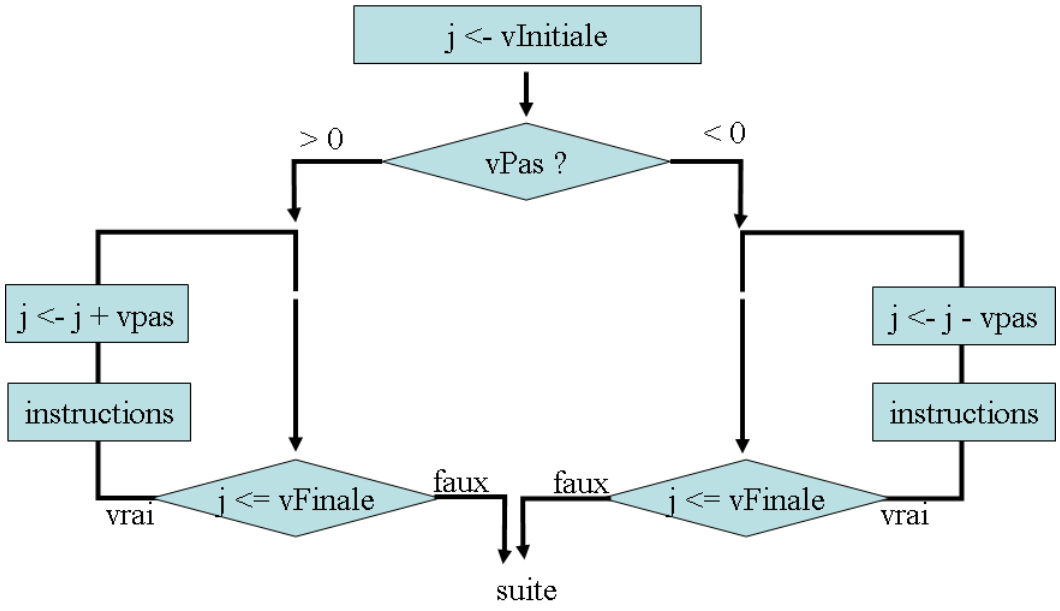


Répétitive Itérer

```
for (int j = 1; j <= n; ++j)
{
  instructions;
}
```

6.2 Répétitive Pour

Répétitive Pour
Elle traduit : Exécuter les `instructions`, `Pour` une variable de boucle `nomVar` (entière ou réelle) dont le contenu varie de la valeur initiale `valDeb` à la valeur finale `valFin` par pas de `valPas` (par défaut de 1). Finitude assurée.



Terminologie
La variable utilisée dans la boucle `Pour` s'appelle la **variable de boucle**, **variable de contrôle**, **indice d'itération** ou **compteur de boucle**. En général, son nom se réduit simplement à une lettre, par exemple `j`.

Répétitive Pour

```
for (initialisations ; conditions ; crementations)
{
  instructions;
}
```

En cas d'initialisations ou de crémentations multiples, séparez-les par des virgules (elles sont évaluées de la gauche vers la droite).

Pas croissant +1 ou décroissant -1

```
for (int j = valDeb ; j <= valFin ; ++j)
{
  instructions;
}

// Attention au sens des inégalités
for (int j = valDeb ; j >= valFin ; --j)
{
  instructions;
}
```

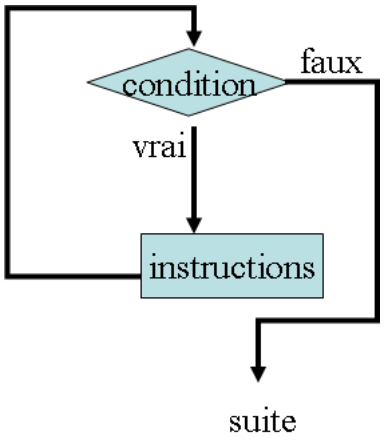
Pas positif ou négatif différent de 1

```
//Pas positif
for (int j = valDeb ; j <= valFin ; j += valPas)
{
  instructions ;
}

//Pas négatif
for (int j = valDeb ; j >= valFin ; j -= valPas)
{
  instructions;
}
```

6.3 Répétitive TantQue

Répétitive TantQue (répétition a-priori)
Elle traduit : `TantQue` la `condition` est vraie, exécuter les `instructions`.
Si la `condition` est fausse dès le début, la tâche n'est jamais exécutée.



Boucle infinie

La séquence `instructions` **doit** modifier la condition de telle manière qu'elle puisse devenir **fausse**. Dans le cas contraire, la boucle va tourner sans fin (condition indéfiniment vraie) : c'est ce qu'on appelle une **boucle infinie**.

Répétitive TantQue

```
while (condition)
{
  instructions;
}
```

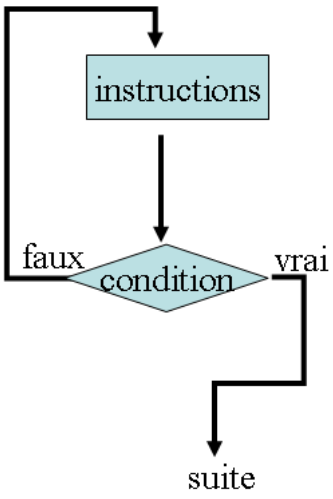
C/C++

Notez l'absence du mot-clé `Faire` d'où l'obligation des parenthèses autour de la condition.

6.4 Répétitive Répéter

Répétitive Répéter (répétition a-posteriori)

Elle traduit : Exécuter les `instructions` `Jusqu'à` ce que la `condition` est vraie.



Boucle infinie
La séquence `instructions` **doit** modifier la condition de telle manière qu'elle puisse devenir **vraie** pour arrêter l'itération.

C/C++ **Répétitive Répéter**

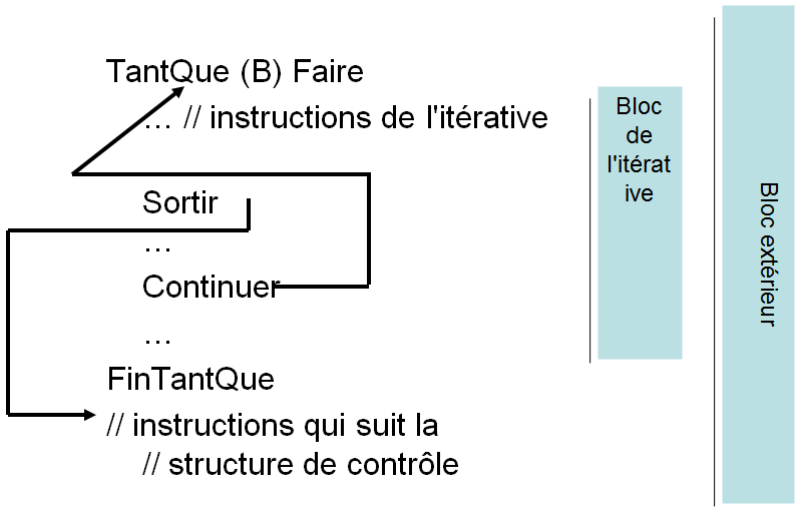
```
do {
  instructions;
} while (!condition); //<- point-virgule
```

C/C++
Notez que la condition d'arrêt de la répétitive `do-while` est l'inverse de celle de la définition `répéter-jusqu'à`.

6.5 Ruptures de séquence (de bloc)

Ruptures de séquence (de bloc)
On en distingue deux :

- `Sortir` : Interrompt l'exécution de la structure de contrôle en provoquant un saut vers l'instruction qui suit la structure de contrôle.
- `Continuer` : Interrompt l'exécution des instructions du bloc d'une **répétitive** et provoque la ré-évaluation de la condition de continuation afin de déterminer si l'exécution de l'itérative doit être poursuivie (avec une nouvelle itération).



Utilisez-les avec parcinomie
Car elles ne permettent pas de réaliser une preuve formelle d'un algorithme (cf. @[Preuve et Notations asymptotiques]).

C/C++ **Rupture Sortir**

```
break;
```

C/C++ **Rupture Continuer**

```
continue;
```

7 Fonction

7.1 Profil de fonction

Profil de fonction
Constitué par le nom de la fonction, la liste des types des paramètres d'entrée et le type du résultat.

C/C++ **Profil de fonction**

```
TypeRes nomFcn(T1 param1, ..., Tn paramN)
```

Explication
Définit le **profil** de la fonction d'identifiant `nomFcn` ayant pour paramètres formels les `paramI` de type correspondants `Ti`. Le **type** de la valeur renvoyée est `TypeRes`. La liste est vide si la fonction n'a pas besoin de paramètres.

Nature de l'information retournée
Précisez toujours `TypeRes` même si le langage définit le type `int` par défaut.

Remarque
En théorie, le type de la valeur retournée peut être un type simple (entier, réel, booléen...), un type structuré, un tableau ou même un objet (ces types seront vus dans les modules suivants). En pratique il conviendra de s'en tenir aux limitations du langage utilisé.

Remarque
Les paramètres formels deviennent automatiquement des variables locales (cf. plus bas, @[Variable locale]) du module.

7.2 Instruction de retour

C/C++ **Instruction de retour**

```
return expression;
```

Explication
Renvoie (retourne) au module appelant le résultat de l'`expression` placée à la suite du mot-clé.

C/C++ : return
L'instruction provoque la terminaison de la fonction.

Dans une fonction

Il doit **toujours** y avoir l'exécution d'une primitive `return`, et ceci quelles que soient les situations (conditions).

En effet, si dans un cas particulier, la fonction s'exécute sans être passée par cette primitive, ceci révèle une incohérence dans la conception de votre fonction car celle-ci aura une valeur inconnue et aléatoire.

7.3 Schéma d'une fonction

C/C++ **Schéma d'une fonction**

```
TypeRes nomFcn(TypeParam1 param1,TypeParam2 param2,...)
{
  TypeRes resultat = valeurInitiale;
  calcul_du_resultat;
  return resultat;
}
```

Explication

Pour des questions de lisibilité et de preuve de programme, il vous est fortement recommandé d'adopter le schéma ci-dessus.

C/C++ : Pas de fonctions imbriquées

Contrairement à d'autres langages (comme Pascal ou Python), il n'est pas possible en C/C++ de définir de fonctions imbriquées : elles doivent toutes se trouver au « premier niveau » du programme, indépendamment de leur place dans la hiérarchie de décomposition de l'algorithme. Ceci explique la structure générale d'un programme C/C++ : celui-ci est composé de fonctions – dont une seule a pour nom `main()` : c'est le point d'entrée du programme – et de variables globales, ces deux éléments pouvant étant répartis dans différents fichiers sources (le résultat de leurs compilations se regroupant dans un exécutable).

C/C++ **Expression fonctionnelle**

```
TypeRes nomFcn(TypeParam1 param1,TypeParam2 param2,...)
{
  return expression;
}
```

Explication

Dans le cas d'une expression calculable directement, on peut regrouper le tout : on parle alors d'**expression fonctionnelle**.

7.4 Appel d'une fonction

Appel d'une fonction

```
v = nomFcn( a1, ..., aN )
```

Explication

Appelle (on dit aussi *invoque*) la fonction `nomFcn` avec les (éventuels) arguments `aI`. La valeur retournée peut être utilisée en tant que macro-expression.

Fonction = macro-expression

Une fonction retourne **toujours** une information à l'algorithme appelant. C'est pourquoi l'appel d'une fonction **ne se fait jamais** à gauche du signe d'affectation.

8 Procédure

8.1 Profil de procédure

Profil de procédure

```
void nomSsp(parametres)
```

Explication

Définit le **profil** de la procédure de nom `nomSsp` ayant pour paramètres formels les `parametres` lesquels décrivent pour chaque paramètre, son nom, son type et sa caractéristique.

C/C++ : Mot-clé void

Le mot-clé `void` (« vide ») spécifie une fonction **sans valeur de retour**.

8.2 Schéma d'une procédure

Schéma d'une procédure

```
void nomSsp(
D1   d1, ..., // les données
R1 & r1, ..., // les résultats
M1 & m1, ...) // les modifiés
{
  // Corps de la procédure:
  // sur une donnée D : passage par valeur
  // sur un résultat R ou un modifié M : passage par référence (&)
}
```

Explication

Définit la procédure de nom `nomSsp`.

8.3 Paramètres formels

Paramètres Entrants/Sortants/Mixtes

Les paramètres **entrants** ou *données* :

- Ont une **valeur à l'entrée** du module.
- Et seront **consultés à l'intérieur** du module.

Les paramètres **sortants** ou *résultats* :

- Ont une **valeur indéterminée à l'entrée** du module.
- Et seront **utilisables après l'appel** du module.

Les paramètres **mixtes** ou *modifiés* :

- Ont une **valeur à l'entrée** du module.
- Et seront éventuellement **modifiés à l'intérieur** de celui-ci.

Paramètres formels

```
void nomSsp(D1 d1, ..., R1 & r1, ... M1 & m1, ...)
TypeRes nomFcn(D1 d1, ..., R1 & r1, ... M1 & m1, ...)
```

Explication

Les `D` sont des paramètres **d**onnées, les `R` des **r**ésultats et `M` des **m**odifiés.
Le symbole `&` (entre le type et le nom du paramètre) définit le passage par référence. (Voir @[Compléments] pour les explications.)

C/C++

Chaque paramètre possède son type et son mode de passage.

8.4 Appel d'une procédure

Appel d'une procédure

```
nomSsp(d1, ..., r1, ..., m1, ...);
```

Explication

Appelle la procédure `nomSsp` avec les (éventuels) arguments figurant entre les parenthèses.

Procédure = macro-instruction

Une procédure étant une macro-instruction, un appel de procédure se fait obligatoirement **en dehors de toute expression de calcul**.

9 Tableaux

9.1 Déclaration et initialisation d'un tableau

C/C++ **Déclaration d'un tableau**

```
TypeElement nomTab[taille];
```

Explication

Déclare une variable dimensionnée. Avec : `TypeElement` le type (simple ou non) des éléments constitutifs du tableau, `nomTab` l'identifiant et `taille` son nombre d'éléments. La taille doit être une valeur entière positive (littéraux ou expressions constantes).

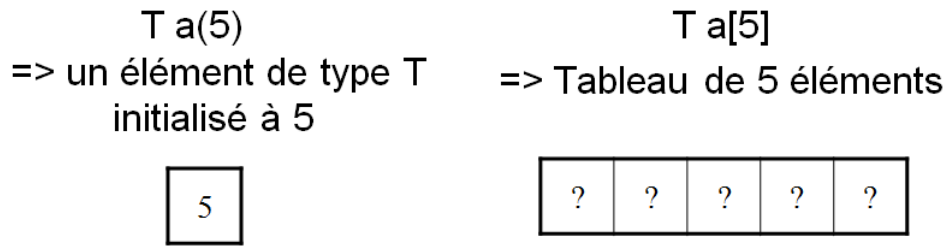
C/C++ **Déclaration et initialisation**

```
TypeElement nomTab [taille] = {val1, ..., valN}; // taille explicite
TypeElement nomTab [] = {val1, ..., valN}; // taille de la liste
```

Explication

Déclare (voir supra) et initialise un tableau. Les `valI` sont des valeurs littérales ou expressions constantes de type compatible `TypeElement` initialisant séquentiellement chacun des éléments du tableau. Dans le cas (2) la longueur de la liste détermine le nombre d'éléments. Dans le cas (1), si la liste d'initialisation contient moins d'éléments que la `taille` spécifiée, les éléments manquants seront initialisés par défaut au zéro du type `TypeElement`.

Distinguer T a(5) et T a(.5.)



9.2 Accès indiciel

C/C++ **Accès indiciel**

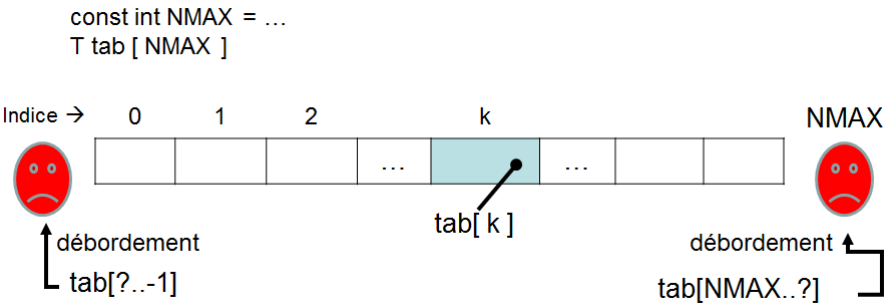
```
tab[k]
```

Explication
Accède à la case d'indice `k` d'un tableau `tab`.
Le temps d'accès à l'élément est fixe.

Numérotation des cases
Chaque langage de programmation possède sa propre convention.

- **alg :** Les cases sont numérotées de 1 (par défaut) à `TMAX` (taille du tableau).
- **C/C++, Java, Python :** Ils commencent à indicer un tableau à partir de 0. Ce principe est dit l'**indexation en base 0**.
- **Basic :** Il débute la numérotation à partir de 1 ou 0.
- **Ada :** Il permet de numéroter les cases à partir d'une valeur quelconque.

Dépassement des bornes
Les langages contrôlent le débordement des bornes d'un tableau et déclenchent une erreur qui généralement arrête le programme. A l'exception du langage C/C++ qui n'effectue aucun contrôle.



9.3 Déclaration et initialisation d'un k-tableau

C/C++ **Déclaration d'un k-tableau**

```
T nomTab[taille1][taille2][...][tailleK];
```

Explication
Déclare un tableau k-dimensionnel de nom `nomTab` d'éléments de type `T`. Les `tailleI` sont des valeurs entières positives (littérales ou expressions constantes).

Cas particulier d'un tableau bidimensionnel

```
T nomTab[taille1][taille2];
```

C/C++ **Déclaration et initialisation**

```
T nomTab[taille1]...[tailleK] = {{{v111, ..., v11k}, ...}; // dim explicite
T nomTab[ ]...[ ] = {{{v111, ..., v11k}, ...}; // dim implicite
```

Explication
Déclare et initialise un tableau k-dimensionnel. Dans le cas (2), la dimension de la composante est la longueur de la liste.

9.4 Accès indiciel

Accès indiciel

```
tab[k1][k2][...]
```

Explication
Accède à la case indice `(k1, k2, ...)` d'un tableau multidimensionnel `tab`. Il faut spécifier autant d'indices qu'il y a de dimensions dans le tableau.

Crochet de crochets
Un tableau k-dimensionnel est vu comme un tableau à une dimension dont chacun des éléments est lui-même un tableau à $(k-1)$-dimension.

Rappel
Les indices sont relatifs à 0.

10 Types complexes

10.1 Définition d'un type structuré

Structure, Champ

Une **structure** permet de regrouper une ou plusieurs variables, de n'importe quel type (structure de données **hétérogène**), dans une entité unique et de la manipuler comme un tout. Chaque élément, appelé **champ** de la structure, possède un nom unique.

Définition d'un type structuré

```
struct TypeStruct
{
  Type1 nomChamp1;
  Type2 nomChamp2;
  ...
}; //<- point-virgule
```

Explication

Crée un nouveau type nommé `TypeStruct` à partir d'autres types élémentaires ou composés déjà définis `TypeI` et d'identifiants respectifs `nomChampI`.

C/C++

- Une erreur fréquente est d'oublier le point-virgule (« ; ») terminal de la définition (ce qui aura pour effet de provoquer le plus souvent une avalanche de messages d'erreurs de la part du compilateur).
- Une autre erreur est de croire que l'on déclare ainsi une variable. La syntaxe supra ne fait qu'annoncer un nouveau type mais aucun emplacement mémoire n'est encore réservé.

10.2 Déclaration et initialisation d'une variable structurée

Propriété

La déclaration de variables d'un type structuré défini est identique à celle des variables simples.

Déclaration d'une variable structurée

```
TypeStruct nomVar;
```

Explication

Déclare une variable structurée `nomVar` de type `TypeStruct` (qui doit être défini).

C/C++ **Déclaration et initialisation**

```
TypeStruct nomVar = { valChamp1, valChamp2, ... };
```

Explication

Déclare et initialise une variable structurée `nomVar`. Chaque `valChampI` est la valeur de type `TypeI` définis dans le type `TypeStruct`.

10.3 Accès aux champs d'une structure

C/C++ **Accès aux champs d'une structure**

```
nomVar.nomChamp
```

Explication

Accède au champ `nomChamp` d'une variable structurée `nomVar` (notation « pointée »).

11 Fichiers

11.1 Déclaration de fichier

Déclaration de fichier

```
#include <fstream>
ifstream f; // fichier en lecture (Input)
ofstream f; // fichier en écriture (Ouput)
```

Explication

Déclare une variable `f` de type `Fichier`.

11.2 Ouverture d'un canal d'entrée/sortie

Ouverture d'un canal

```
f.open(nomFich);
```

Déclaration et Ouverture d'un canal

```
#include <fstream>
ifstream f(nomFich); // fichier en Lecture
ofstream f(nomFich); // fichier en Ecriture
```

Explication

Associe un canal d'entrées/sorties à un fichier. La variable `f` sera utilisée pour toutes les opérations sur ce fichier jusqu'à sa fermeture. Le `nomFich` est une chaîne de caractères contenant le nom du fichier à ouvrir avec éventuellement le chemin d'accès à savoir le nom du disque et le chemin relatif ou absolu permettant d'atteindre le fichier. A défaut, le fichier doit être dans le dossier courant (habituellement le dossier où est sauvegardé le projet en exécution).

Erreur à l'ouverture

Le système peut être dans l'impossibilité d'ouvrir le fichier spécifié pour une ou l'autre des raisons suivantes :

- Tentative d'ouvrir un fichier inexistant en mode lecture.
- Tentative d'ouvrir un fichier qui est déjà ouvert.
- Tentative d'ouvrir un fichier sur un canal d'entrées/sorties invalide.
- Le nom du fichier est invalide : ceci peut être dû au dossier inexistant, au nom du fichier contenant des caractères interdits par le système d'exploitation ou à l'unité de stockage défectueuse ou non disponible.

Dans ce cas le système provoque l'interruption du programme et affiche un message d'erreur précisant la cause de l'erreur.

Fichier en mode écriture
L'ouverture efface automatiquement son contenu s'il existe déjà.

11.3 Fermeture d'un canal d'entrées/sorties

Fermeture d'un canal

```
varFich.close();
```

Explication
Ferme le canal d'entrées/sorties `f` précédemment ouvert et purge toutes les mémoires tampon. Dans le cas où le fichier a été ouvert en écriture, cette primitive place la marque spéciale de fin de fichier dans l'élément courant. Une fois le fichier fermé, il n'est plus permis de l'utiliser.

Remarque
Un fichier créé et non refermé risque de contenir des données aléatoires et invalides.

11.4 Lecture depuis un canal d'entrée

Lecture depuis un canal d'entrée

```
f >> nomVar1 >> nomVar2...;
```

Explication
Lit dans le fichier référencé par la variable `f`, ouvert en lecture, des valeurs qui seront affectées aux variables `nomVarI`. Cette opération peut échouer si la fin de fichier est atteinte. Ceci est décelable grâce à la fonction `FinDeFichier`.

Remarque
Le canal d'entrées/sorties doit obligatoirement être associé à un fichier ouvert en mode `Lecture`. Toute tentative de lecture visant un canal d'entrées/sorties associé à un fichier ouvert en mode `Ecriture` ou `Ajout` cause l'arrêt d'exécution de l'algorithme.

11.5 Écriture sur un canal de sortie

Écriture sur un canal de sortie

```
f << expr1 << expr2...;
```

Explication
Rajoute des données dans le fichier référencé par la variable `f`, ouvert en écriture, les valeurs des expressions `exprI`. Cette opération peut échouer si le support utilisé pour le fichier est plein.

Remarque
Le canal d'entrées/sorties doit obligatoirement être associé à un document ouvert en mode `Ecriture` ou `Ajout`. Toute tentative d'écriture visant un canal d'entrées/sorties associé à un document ouvert en mode `Lecture` provoque l'arrêt d'exécution de l'algorithme.

11.6 Détection de fin de contenu

Détection de fin de contenu

```
f.eof()
```

« eof » signifie « EndOfFile ».

Explication
Renvoie la valeur `Vrai` si le pointeur de lecture référencé par `f` détecte la **fin de fichier**, `Faux` sinon.

Attention
La primitive n'est applicable qu'aux canaux associés en mode `Lecture`. Toute invocation de la primitive sur un canal associé à un document ouvert en mode `Ecriture` ou `Ajout` cause l'arrêt d'exécution de l'algorithme.

12 Opérateurs

12.1 Opérateurs arithmétiques

Opérateurs arithmétiques

Dits aussi **opérateurs algébriques**, ils agissent sur des opérandes de type numérique.

C/C++ **Opérateurs arithmétiques**

Opérateur Mathématique	Signification	Équivalent C/C++
+	(unaire) valeur	`+a`
-	(unaire) opposé	`-a`
+	addition	`a + b`
-	soustraction	`a - b`
*	multiplication	`a * b`
/	division décimale	`a / b`
div	division entière	`a / b`
mod	modulo (reste de la division entière)	`a % b`

C/C++ : Élévation à la puissance

Il est nécessaire de faire appel :

- Soit à des produits successifs pour des puissances entières pas trop grandes (par exemple, on calculera x^3 comme `x*x*x`).
- Soit à la fonction `pow` de la bibliothèque standard.

C/C++ : Que vaut a / b ?

La division s'effectue sur :

- $\mathbb{N}$: si a **et** b sont entiers (division entière)
- $\mathbb{R}$: si a **ou** b sont réels (division réelle)

Division euclidienne, cas des négatifs

Il n'y a pas unicité du quotient et du reste lorsque le dividende ou le diviseur sont négatifs. Si a et b sont deux entiers relatifs dont l'un au moins est négatif, il y a plusieurs couples (q, r) tels que $a = b \times q + r$ avec $|r| < |b|$. Par exemple, si $a = -17$ et $b = 5$ alors $(q = -3, r = -2)$ ou $(q = -4, r = 3)$ sont deux solutions possibles. Habituellement, q et r sont choisis comme le quotient et le reste de la division entière de $|a|$ et $|b|$ affectés du signe approprié (celui permettant de vérifier $a = b \times q + r$ avec $|r| < |b|$). Dans l'exemple ci-avant, la solution retenue serait $(q = -3, r = -2)$. Par contre, la norme impose que la valeur de $(a \text{ div } b) * b + a \bmod b$ soit égale à la valeur de a.

Pas d'overflow sur les entiers

Si le résultat d'une opération dépasse la capacité d'un entier, le résultat est un entier négatif ou inférieur : il n'y a pas d'erreur de **dépassement de capacité**.

Division par zéro
Tout emploi de la division devra être accompagné d'une réflexion sur la valeur du dénominateur, une division par 0 entraînant toujours l'arrêt d'un programme.

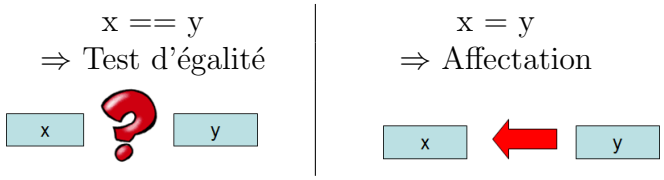
12.2 Opérateurs de comparaison

Opérateurs de comparaison
Dits aussi **opérateurs relationnels** ou **comparateurs**, ils agissent généralement sur des variables numériques ou des chaînes et donnent un résultat booléen. Pour les caractères et chaînes, c'est l'ordre alphabétique qui détermine le résultat.

C/C++ **Opérateurs de comparaison**

Opérateur Mathématique	Signification	Équivalent Algorithmique	Équivalent C/C++
$<$	(strictement) inférieur	`a < b`	`a < b`
$\leq$	inférieur ou égal	`a <= b`	`a <= b`
$>$	(strictement) supérieur	`a > b`	`a > b`
$\geq$	supérieur ou égal	`a >= b`	`a >= b`
$=$	égalité	`a = b`	`a == b`
$\neq$	différent de (ou inégalité)	`a <> b`	`a != b`

Distinguer == et =



A gauche Compare la valeur de x à celle de y et rend true si elles sont égales, false sinon (et donc **ne modifie pas** la valeur de x)

A droite Affecte à la variable x la valeur de y (et donc **modifie** la valeur de x)

12.3 Opérateurs logiques

Opérateurs logiques
Dits aussi **connecteurs logiques** ou **opérateurs booléens**, ils agissent sur des expressions booléennes (variables ou expressions à valeurs booléennes) pour donner un résultat du même type. Ils peuvent être enchaînés.

Opérateurs logiques

Opérateur Mathématique	Signification	Équivalent Algorithmique	Équivalent C/C++
¬	négation (unaire)	`Non a`	`!a`
∧	conjonction logique	`a Et b`	`a && b`
∨	disjonction logique (ou inclusif)	`a Ou b`	`a || b`

Opérateurs logiques

Opérateur Mathématique	Signification	Équivalent Algorithmique	Équivalent C++
¬	négation (unaire)	`Non a`	`not a`
∧	conjonction logique	`a Et b`	`a and b`
∨	disjonction logique (ou inclusif)	`a Ou b`	`a or b`

Opérateur Ou-exclusif

Il n'y a pas d'opérateur OU-exclusif (`xor`) logique.

12.4 Opérateur Si-expression

Opérateur Si-expression

```
exprBool ? exprAlors : exprSinon
```

Explication

Évalue l'expression logique `exprBool` et si elle est vérifiée, effectue l'expression `exprAlors`, sinon l'expression `exprSinon`. Les `exprAlors` et `exprSinon` doivent être du même type.

Remarque

Cette syntaxe très raccourcie doit être réservée à de petits tests.

12.5 Priorité des opérateurs

C/C++ : Priorité des opérateurs

Les opérateurs de même priorité sont regroupés sur une même ligne.

Priorité	Opérateur Algorithmique	Opérateur C/C++	Signification
La plus élevée	- (unaire)	- (unaire)	Négation algébrique
	^	(aucun)	Puissance
	* / div mod	* / / %	Multiplication, division, div. entière, modulo
	+ -	+ -	Addition et soustraction
	< <= > >=	< <= > >=	Opérateurs de comparaison
	= <>	== !=	Opérateurs d'égalité
	Non	!	Négation logique
	Et	&&	Et logique
La plus basse	Ou	||	Ou logique

Cas de combinaisons de Et et de Ou

Mettez des parenthèses :

`(cond1 Et cond2) Ou cond3`

est différent de

`cond1 Et (cond2 Ou cond3)`

En l'absence de parenthèses, le `Et` est prioritaire sur le `Ou`.

13 Fonctions prédéfinies

13.1 Fonctions mathématiques

Fonctions mathématiques
Elles agissent sur des paramètres à valeurs réelles et donnent un résultat réel.

Pour les utiliser

```
#include <cmath>
```

Quelques Fonctions mathématiques

Fonctions Mathématiques	Signification	Équivalent C/C++
$\text{acos}(x)$	Cosinus inverse (radians)	`acos(x)`
$\text{asin}(x)$	Sinus inverse (radians)	`asin(x)`
$\text{atan}(x)$	Tangente inverse (radians)	`atan(x)`
$\lceil x \rceil$	Réel de l'entier supérieur	`ceil(x)`
$\cos(x)$	Cosinus de x (radians)	`cos(x)`
e^x	Exponentielle de x (base e)	`exp(x)`
$\lvert x \rvert$	Valeur Absolue de x	`fabs(x)`
$\lfloor x \rfloor$	Réel de l'entier inférieur	`floor(x)`
$\log(x)$	Logarithme naturel (base $e = 2.718281$)	`log(x)`
$\log_{10}(x)$	Logarithme en base 10	`log10(x)`
x^y	Puissance	`pow(x,y)`
$\sin(x)$	Sinus de x (radians)	`sin(x)`
$\sqrt{x}$	Racine carrée	`sqrt(x)`
$\tan(x)$	Tangente de x (radians)	`tan(x)`

Racine carrée
Attention de ne l'utiliser qu'avec un radicant positif.

13.2 Fonctions caractère

Pour les utiliser

```
#include <cctype>
```

Fonctions caractère

```
int isalpha(char); // caractère alphabétique?
int isupper(char); // majuscule?
int islower(char); // minuscule?
int isdigit(char); // chiffre?
int isxdigit(char); // chiffre + lettres?
```

```
int isspace(char); // espace?
int iscntrl(char); // caractère de contrôle? (ASCII 0..31 et 127)
int ispunct(char); // caractère de ponctuation?
int isalnum(char); // chiffre ou lettres?
int isprint(char); // caractère imprimable?
int isgraph(char); // isalpha ou isdigit ou ispunct
int isascii(char); // ASCII 0..127?

char tolower(char); // renvoie la minuscule
char toupper(char); // renvoie la majuscule
```

13.3 Opérations de chaînes

Longueur d'une chaîne
Nombre de caractères dans la chaîne.

Concaténation de deux chaînes
Consiste à prendre ces deux chaînes et à les coller bout-à-bout.

Sous-chaîne d'une chaîne
Suite consécutive de caractères de la chaîne : c'est une partie (un morceau) de cette chaîne.

Comparer deux chaînes
C'est déterminer laquelle des deux précède l'autre pour l'ordre alphabétique des dictionnaires (encore appelé **ordre lexicographique**) où la chaîne vide `""` est avant toutes les autres et où il faut tenir compte des lettres minuscules et majuscules ainsi que des caractères spéciaux. La comparaison de deux chaînes s'effectue **caractère par caractère de gauche à droite** jusqu'à rencontrer la fin d'une des deux chaînes ou une différence. La chaîne de caractères qui **précède** l'autre est celle qui, la première, a un caractère qui **précède** le caractère correspondant de l'autre chaîne. En cas d'égalité permanente, la chaîne la plus courte **précède** la chaîne la plus longue.

Comparaison de chaînes
Les opérateurs usuels `==`, `!=`, `<`, `>`, `<=` et `>=` servent à comparer deux chaînes.

Opérations de chaînes

- Affectation : `s = s2; s = "valeur"; s = 'c';` // 1 seul
- Concaténation : `s = s2 + s3; s = s2 + "valeur"; s = s2 + 'c';`
- Accès au caractère `k` : `s[k]` (avec $k = 0..$)

Remarque
L'« addition » de chaînes n'est pas une opération commutative car la chaîne `x + y` n'est pas identique à la chaîne `y + x`.

Attention
Les positions dans les chaînes commencent à 0.

Quelques méthodes
Les `s| sont des \lstinline`string@ ; l'entier `p` (avec $p = 0..$) désigne une position ; l'entier positif `n` désigne un nombre de caractères :

- Taille (nombre de caractères) : `s.length()`
- Insertion de `s2` dans `s` en `p` : `s.insert(p,s2)`
- Remplacement en `p` par `n` caractères de `s2` : `s.replace(p,n,s2)`
- Suppression : `s.replace(p,n,"")`
- Recherche avant : `s.find(s2)`
- Recherche arrière : `s.rfind(s2)`
- Extraction de `n| caractères à partir de \lstinline`p@ : `s.substr(p,n)`
- Conversion en chaîne « à-la-C » : `s.c_str()`

Les méthodes `find` et `rfind` retournent la première, resp. la dernière, position où la sous-chaîne `s2` recherchée est trouvée, la constante `string::npos` en cas d'échec.