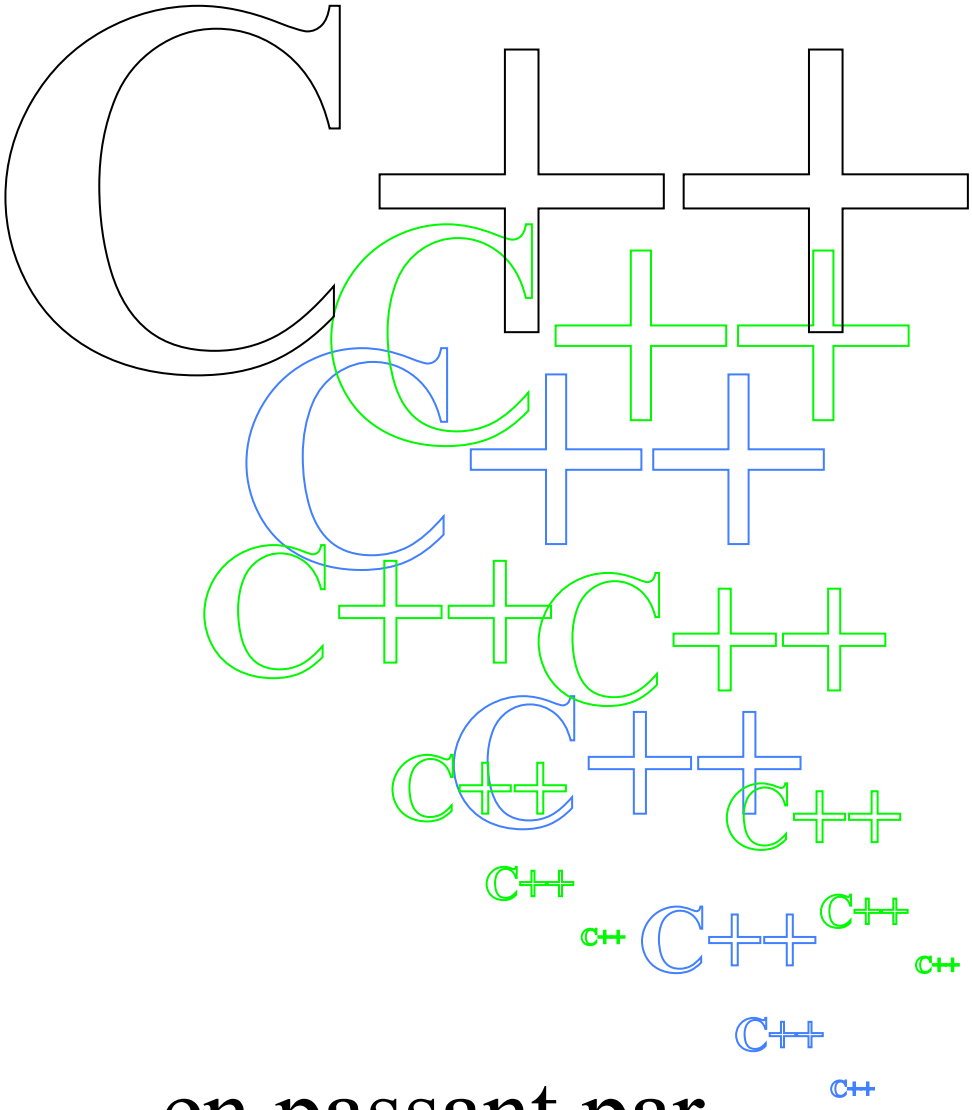


… en passant par

C

Partie 1

Molliet J.-P.

INTRODUCTION

Pour des raisons historiques C++ reste indissociable de C, lui-même l'étant du système Unix.

En 1983 Bjarne Stroustrup des laboratoires Bell crée C++ (C lui date de 1972) qu'il nomme d'ailleurs initialement *C with Classes*.

Le langage est normalisé par l'ISO une première fois en 1998 (ISO/CEI 14882:1998), puis en 2003 (ISO/CEI 14882:2003) et en 2011 (ISO/CEI 14882 :2011).

Le créateur du langage l'a voulu compatible avec C (ceci s'avère en grande partie vrai, mais ne l'est pas à 100%!). Il en améliore, entre autres, la sécurité. On garde l'aspect procédural de C en lui ajoutant essentiellement le paradigme objet inspiré de Simula.

L'objectif de ce cours consiste à présenter la partie procédurale du langage en mettant toujours en évidence les spécificités C++, respectivement C. Nous ne parlerons donc pas (en réalité peu car nous n'échapperons pas à quelques allusions!) d'objet. Ce choix, discutable au demeurant, correspond à une contrainte imposée par le département afin de garder une cohérence avec les autres cours!

Pour illustrer ce polycopié nous utilisons de nombreux exemples; nous attirons fortement votre attention sur le fait que ces exemples sont là pour illustrer un aspect précis de la théorie en cours d'étude, mais qu'en aucun cas ils ne reflètent la manière finale dont il faudra programmer!

Un tout grand merci à mes collègues: messieurs G.-M. Breguet et R. Rentsch pour leur lecture attentive et critique de ce document.

❑ *Le Langage*

Le langage C++ permet à la fois la programmation efficace proche du niveau machine (programmation système) et la programmation à un haut niveau d'abstraction (programmation d'applications). Dans les deux domaines c'est un des langages les plus répandus.

Le créateur de C++, Bjarne Stroustrup, reçut l'idée pour un nouveau langage de programmation à travers ses expériences avec le langage Simula. Ce langage était adapté pour le développement de grands projets de logiciel, mais rendait difficile le développement de programmes performants. Stroustrup eut l'idée d'étendre le langage C. Il choisit C parce que c'était un langage universel qui produisait du code performant et qui était facile à porter sur

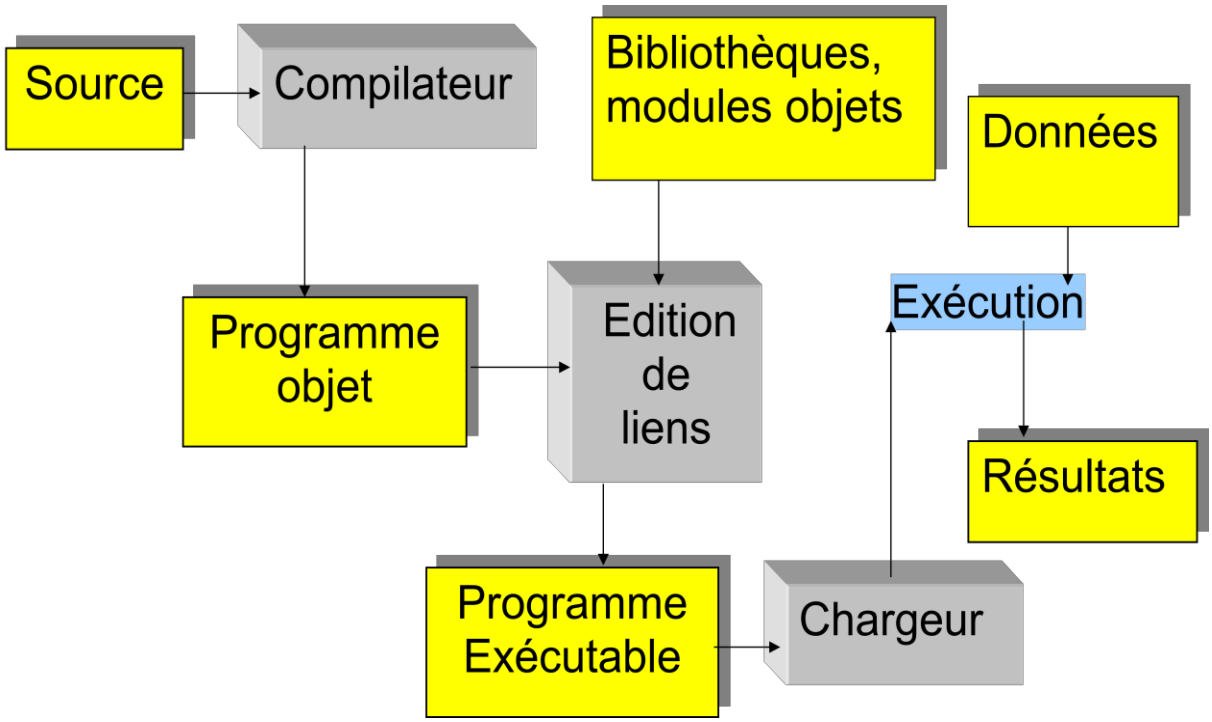
d'autres plateformes.

Parmi les langages qui avaient une influcence sur le développement de C++ on peut compter, hormis des Simula et C déjà cités, Ada et ALGOL 68. C++ lui-même a aussi influencé d'autres langages venus après lui, comme Java, C#, PHP et Perl.

❑ *Les objectifs à atteindre*

Nous l'avons dit, le langage n'est qu'un outil, mais qu'il faut parfaitement le maîtriser pour être productif. Toutefois, avant de penser à coder dans un langage donné, il faut avoir choisi les bons **algorithmes** pour aboutir à une solution optimale. Un algorithme consiste en une suite d'opérations à effectuer pour résoudre un problème donné. Une recette de cuisine représente un algorithme. Le choix du bon algorithme, opération non triviale, représente la base d'une application performante et souvent aussi d'une application facile à maintenir! Après détermination de l'algorithme (des algorithmes pour une application globale), il conviendra de le traduire dans un langage de programmation, pour nous C++/C!

Bien que les environnements de développement masquent les différentes phases, les enchaînant de manière automatique, nous pouvons représenter les opérations nous amenant à la création d'un programme exécutable de la manière suivante:



Durant votre phase de création, même pour de petits exercices simples qui nous aideront par la suite lors de la création d'applications plus complexes, ayez à l'esprit les objectifs suivant à

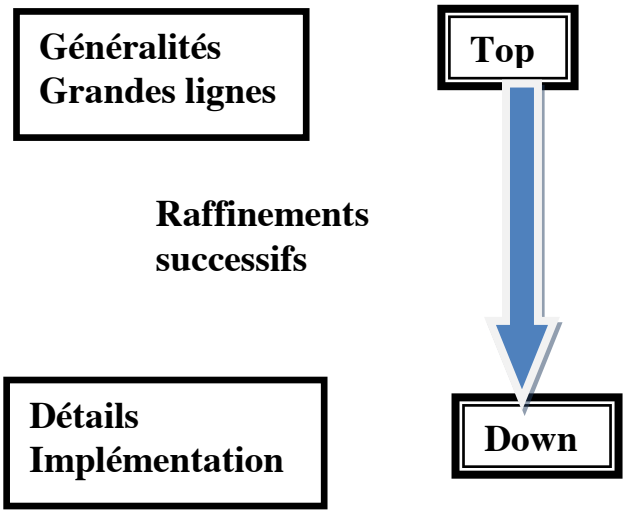
atteindre:

- La **fiabilité**: elle consiste à détecter les erreurs à la compilation ou, tout au moins à l'exécution. Il s'agit certainement là de la qualité première de tout logiciel. On ne devrait jamais se trouver en présence de comportements imprévus.
- La **lisibilité**: elle permet de simplifier la correction des erreurs qui surviendront malheureusement, et de faciliter les modifications/adjonctions futures.
- La **simplicité**: là aussi, plus c'est simple, plus ce sera facile à corriger et maintenir.
- La **cohérence**: utilisez partout dans votre développement la même présentation, la même structuration, la même manière de commenter et de choisir les identificateurs; cela permettra de comprendre plus facilement la logique de vos constructions.
- **L'efficacité**: bien que importante, l'utilisateur n'aimant pas attendre et, comme on dit, "le temps c'est de l'argent", ce ne sera de loin pas notre objectif prioritaire; toutefois ne la négligeons pas, tant que cela ne se fait pas au détriment des points qui précèdent.
- La **portabilité**: le code réalisé devrait demeurer le plus indépendant possible de l'architecture de la machine et des spécificités de l'environnement de développement utilisé. N'utilisez pas les particularités d'une implémentation.

Pour vous en pratique cela signifie:

- Lire attentivement l'énoncé du problème, être convaincu de bien le comprendre avant d'aller plus loin dans la réalisation.
- Réfléchir au problème, déterminer et mettre en évidence les points principaux à résoudre.
- Cherchez le meilleur algorithme possible, ce n'est pas évident!
- Ecrire, dans le formalisme qui vous convient, mais sans coder réellement pour l'instant, l'architecture principale de votre application.
- Finalement, dans votre environnement de développement, coder votre programme. Si celui-ci revêt déjà une certaine importance, n'essayez pas de réaliser tout en une seule fois. Travaillez par étapes et testez chaque étape avant de passer à la suivante.
- Si vous le pouvez, faites tester par une autre personne.

Nous préconisons d'appliquer, de manière générale, une démarche "Top Down", c'est-à-dire que nous partons du général pour aller de plus en plus vers le détail. Cette démarche peut se représenter selon le schéma ci- après.



Pour aboutir à cette démarche, nous travaillerons selon la technique dites des raffinements successifs, méthode basée sur l'idée que, étant donné un problème à résoudre, il faut le décomposer en sous-problèmes de telle manière que:

- Chaque sous-problème constitue une partie du problème donné.
- Chaque sous-problème est plus simple (à résoudre) que le problème donné.
- La réunion de tous les sous-problèmes est équivalente à celle du problème donné.

Par la suite, en règle générale, à chaque niveau de raffinement correspondra un sous-programme.

Elements de base

Premier exemple

Dans le monde C/C++, le premier exemple présenté est toujours du genre de celui ci-dessous. Nous vous le donnons brutalement en guise d'introduction et nous en tirerons nos premières règles.

```
/*
  Premier programme exemple
    SALUT1
*/
#include <iostream>
using namespace std;
int  main ( )
{
  // Juste un affichage
  cout << "salut" << endl << "...et a bientot" << endl;
  return 0;
}
```

Ce qui donne comme résultat:

```
salut
...et a bientot
```

Ce premier exemple de programme[1] va nous permettre de tirer nos premières règles de construction des programmes C++; certaines d'entre-elles seront données brièvement pour nous permettre de travailler, d'autres carrément fournies comme des "recettes de cuisine", mais nous vous promettons qu'elles seront reprises par la suite pour un développement plus approfondi.

[1] Ce programme n'est pas compilable en C. Les formes purement C relatives aux entrées/sorties ne seront présentées qu'ultérieurement.

Les commentaires

Comme nous pouvons le constater, 2 formes de commentaires sont possibles:

- Un commentaire commençant par /* et se termine par */:

```
/* Ceci est un commentaire */
```

Un tel commentaire peut se poursuivre sur plusieurs lignes:

```
/* Ceci en est ...
   ... aussi un!
*/
```

En fait, un commentaire de cette forme pourra venir partout où un séparateur est admis ou nécessaire, en d'autres termes entre chaque unité lexicale, à savoir: les identificateurs, les mots réservés, les opérateurs, les séparateurs, etc.

La seule restriction à mettre en évidence réside dans le fait que les commentaires ne peuvent pas être imbriqués:

```
/* Cette construction /* n'est pas valide
*/ le commentaire s'arrêtant après les 2
 premiers caractères de la ligne qui précède */
```

Il faut commenter ses programmes de manière brève, claire et explicite, mais toutefois sans excès. Le juste équilibre s'avère assez difficile à trouver au début; ayez toutefois à l'esprit le fait qu'un commentaire doit fournir une explication complémentaire par rapport aux instructions, sinon il ne sert à rien!

Ainsi:

```
i = i + 1;       /* Incrementer i de 1 */
```

se révèle typiquement un commentaire inutile, qui n'apporte aucune nouvelle information par rapport à l'instruction elle-même.

Certaines constructions du langage paraîtront suffisamment denses pour que nous ayons encore plus tendance à les documenter par un commentaire.

- La deuxième forme possible pour un commentaire:

```
i = 1;       // Ceci est un commentaire de ligne
```

A partir du « // » tout le reste de la ligne est considéré comme un commentaire, sauf si l'on se trouve dans une chaîne de caractères.

On distingue en général les commentaires de blocs expliquant tout un groupe d'instructions et qui se mettent devant ces instructions et les commentaires d'instruction qui se mettent après celle-ci sur le reste de la ligne. Les premiers seront souvent de la forme /* … */ et les deuxièmes de la forme // …

Important:
Vous devez commenter correctement vos programmes!

Structure générale d'un programme

Un programme principal n'est rien d'autre qu'une fonction (nous y reviendrons!), qui obligatoirement s'appelle: "main" et qui pourra par la suite appeler d'autres fonctions (sous-programmes). Les parenthèses après le "*main*" sont là pour un éventuel passage de paramètres. Le **int** que nous trouvons devant *main* correspond au type du résultat de la fonction donc de notre programme (ici un résultat entier); il reflète l'état de terminaison du programme (l'éventuel code d'erreur!) transmis à l'appelant, soit en général le système d'exploitation.

La ligne d'en-tête:

```
int main ( ) [1]
```

Après la ligne d'en-tête vient le corps du programme qui est en fait un bloc, notion que nous allons préciser quelque peu.

Instruction et bloc

Une instruction (nous en verrons la liste au fur et à mesure de l'avance de ce cours) se termine par un point virgule, il joue donc le rôle de terminateur.

Comme vous pouvez le constater, le corps de notre programme principal se met entre:

```
{... }
```

Ce qui constitue un bloc.

Le plus petit programme C++ que l'on puisse imaginer est donc:

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

Il est compilable, mais ne fait bien sûr pas grand-chose!

[1] Pour un programme C selon la norme C99, la ligne d'en-tête prend la forme: **int** main (**void**) nous reviendrons sur le mot réservé **void** par la suite!

Notre programme principal étant une fonction qui livre un résultat entier, l'exécution de ce programme se termine par:

```
return 0;   //[1]
```

Ceci indique une terminaison normale; la valeur retournée représente un code d'erreur qui permettra éventuellement au système d'exploitation de livrer un message approprié (0 => pas d'erreur!).

Par la suite, partout où une structure de contrôle (test, boucle...) s'applique à une instruction, ce qui est la règle, et que nous voulons l'appliquer à un groupe d'instructions, nous mettrons simplement ces instructions entre accolades:

```
{
   INSTRUCTION_1;
   INSTRUCTION_2;
   .....
   INSTRUCTION_N;
}
```

bloc ou instruction composée

Notez par contre qu'il n'y a pas de ";" après "}" du bloc!

[1] Par la suite nous donnerons une forme plus propre pour fournir la valeur de terminaison car il n'est pas certain que sur tous les systèmes la valeur 0 signifie "pas d'erreur"!

❑ *Identificateurs et mots réservés*

Nous n'allons pas encore entrer dans le détail des éléments constituant un programme, mais nous pouvons déjà signaler que:

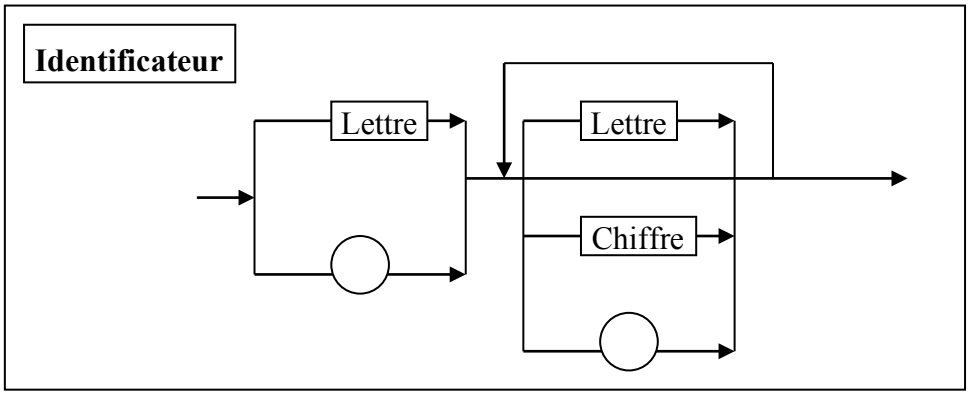
main est l'identificateur (imposé) désignant le programme principal.

cout est un identificateur désignant le flux de sortie (défini dans *iostream*, plus précisément dans *ostream*), dont nous préciserons la sémantique plus tard.

endl est un identificateur indiquant un passage à la ligne (définie dans *iostream*).

return, using et **namespace** sont des mots réservés prédéfinis du langage, vous ne pouvez pas les utiliser dans un autre sens.

En C++, comme dans tous les autres langages de programmation, nous aurons à choisir des identificateurs qui devront respecter les règles syntaxiques suivantes:



Donc un identificateur commence par une lettre[1] ou un symbole souligné et peut se poursuivre par d'autres lettres, chiffres ou symboles soulignés. En principe, sans que cela soit une règle impérative, les identificateurs commençant par un symbole souligné sont réservés pour l'interface avec l'environnement. Vos identificateurs peuvent avoir la longueur que vous désirez.

Important: Majuscules et minuscules sont distinguées; ainsi *TOTO* et *Toto* ne représentent pas le même identificateur.

[1] Les lettres accentuées ne sont normalement pas supportées

Le langage C++ comporte un certain nombre de mots clés qui en fait sont des mots réservés. Ils seront explicités au fur et à mesure de notre avance dans la connaissance du langage. En voici la liste[1]:

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

[1] Les mots de cette liste marqués en italiques sont également des mots réservés en C

Important:

Etant donné qu'il s'agit de mots réservés, nous ne pourrons pas les utiliser dans un autre but que celui prévu par la définition du langage.

Ils doivent impérativement s'écrire en minuscules!

Note relative à C et non valable en C++:

La norme C99 a aussi introduit les mots réservés suivants, non reconnus en C++:

restrict **_Bool** **_Complex** **_Imaginary**

Mise en forme d'un programme

Comme une majorité d'autres langages, C++ laisse une très grande liberté au programmeur quant à la manière de structurer son code source. Cela ne veut pas dire que l'on doit pour autant faire n'importe quoi!

Pensez à:

- "Aérer" vos programmes.
- Choisir des identificateurs significatifs, ni trop longs, ni trop courts.
- Commenter correctement votre code.
- Indenter proprement et systématiquement vos structures de telle sorte que l'aspect visuel du code source reflète sa structure logique.
- Etc.…

Les bases de l'affichage

Il y a encore quelques éléments que nous pouvons déduire de notre exemple introductif, mais nous les développerons dans les chapitres à venir. Signalons simplement:

- `cout ....;`

Représente l'envoi sur le flux de sortie (pour nous l'écran) des éléments qui suivent, symbolisés ici par …. Sa définition vient du fichier *iostream*.

Donnons pour l'instant quelques précisions rudimentaires sur l'utilisation de *cout*. Nous développerons davantage, par la suite lorsque cela deviendra nécessaire, mais ces quelques points nous permettrons déjà de construire nos premiers exemples de programmes.

L'opérateur: << indique qu'il faut "envoyer" l'élément qui suit (il peut s'agir d'une constante, d'une variable ou d'une expression) vers le flux de sortie; exemple avec une simple chaîne de caractères:

```
cout  << "Salut";
```

Relevons déjà au passage qu'une constante chaîne de caractères se met entre guillemets!

L'inclusion du fichier *iostream* nous fournit également la définition du symbole *endl* qui représente une fin de ligne (un passage à la ligne suivante); nous pouvons également envoyer cette indication vers le flux de sortie:

```
cout  << endl;
```

Une même opération *cout* permet d'envoyer plusieurs éléments vers le flux de sortie:

```
cout << "salut" << endl << "...et a bientot" << endl;
```

- `#include <iostream>`

Représente l'inclusion dans notre programme source du fichier "`iostream`" qui contient un certain nombre de définitions: dont celles que nous avons utilisées dans le premier exemple: *cout* et *endl*. Pour l'instant tous nos programmes comporteront cette ligne, elle fait presque partie de son en-tête!

En pratique, les lignes qui commencent par un # (#include, #define ...) sont des lignes traitées par un préprocesseur du compilateur, dont nous reparlerons dans la suite.

- **`using namespace`** `std`

Voici encore une notion que nous n'allons pas développer pour l'instant, prenez-la comme une recette de cuisine à appliquer systématiquement. Elle indique que nous pouvons utiliser directement tous les éléments regroupés sous le nom *std*. Il s'agit pour nous de ce qui nous vient de *iostream* (*cout*, *endl* et bientôt *cin*). Sans cette spécification, le dernier *cout* ci-dessus se serait écrit:

```
std::cout  <<  "salut"  <<  std::endl  <<  "...et  a  bientot"
           << std::endl;
```

ce qui, avouons-le, deviendrait très vite fastidieux.

Note: Tous les éléments présentés ci-dessus pour réaliser des affichages n'existent pas en C; dans ce langage il faudrait par exemple inclure le fichier *stdio.h* et utiliser la fonction *printf*. Ces éléments sont également disponibles en C++, mais comme ils sont finalement plus complexes d'utilisation, nous les présenterons ultérieurement.

❑ *Conclusions sur l'introduction*

Voilà, rassurez-vous, de nombreuses notions vont se préciser petit à petit. Toutefois, donnons encore une nouvelle version de notre premier programme, dans le but d'éviter qu'à la fin du programme la fenêtre ne se referme sans nous laisser le temps de lire les derniers résultats. Dans le contexte rudimentaire de nos premières petites applications, aucune solution propre n'existe pour résoudre ce problème, aussi nous vous proposons une solution spécifique à notre environnement de travail.

```
/*
   Premier programme exemple
     SALUT2
*/
#include <iostream>
using namespace std;
#include <cstdlib>
int  main ( )
{
   // Juste un affichage
   cout << "salut" << endl << "...et a bientot" << endl;
   cout << "Fin du programme... ";
   system ( "pause" );
   return EXIT_SUCCESS;  //[1]
}
```

Ici pour pouvoir utiliser la fonction *system*

Disponible en C++

En C: #include <stdlib.h>

La fonction *system* est disponible dans tous les environnements, par contre ses paramètres changent d'un environnement à l'autre!

Ici, avec la commande *pause*, nous demandons au système (DOS/WINDOWS) qu'il affiche un message informant l'utilisateur qu'il doit presser une touche du clavier avant que le programme ne poursuive son exécution!!!

[1] *EXIT_SUCCESS* est une constante définie dans *cstdlib*, qui correspond à une terminaison normale du programme et qui, dans la très grande majorité des environnements vaut 0!

Les types de base

Mise en garde: dans les 2 chapitres qui vont suivre, nous allons mettre en place un nombre important de règles quelque peu fastidieuses mais qu'il faudra de toute façon connaître. Nous avons donc choisi de les donner brutalement dès le départ!

Avant de commencer cette description, signalons que pour rester compatible avec C dont le contrôle de type est loin d'être une vertu, en C++ nous constaterons que nous pouvons faire et même parfois sommes obligés de faire des choses plutôt étranges! Certaines personnes trouvent que c'est l'un des aspects qui font la force du langage...

❑ *Les types de base sont*

Le tableau ci-dessous vous donne la liste des types de base mis à disposition par le langage:

Nom	Caractéristiques
char	En principe pour des caractères
int	Pour des entiers signés dont la taille peut varier selon l'implémentation
float	Pour des réels en virgule flottante possédant une précision d'au moins 6 chiffres significatifs (généralement 32 bits)
double	Pour des réels en virgule flottante possédant une précision d'au moins 15 chiffres significatifs (généralement 64 bits)
bool	Pour des booléens
wchar_t	Pour des caractères à code étendu
void	Pour marquer une absence ou un type neutre

Notez que les identificateurs de types sont des mots réservés en C/C++, contrairement à d'autres langages (Ada par exemple).

De plus **void** ne représente pas à proprement parler un type:

- Dans le cadre des pointeurs, il identifiera un pointeur neutre.
- Dans le cadre des fonctions il pourra indiquer qu'elle ne livre pas de résultat (elle s'utilise comme une procédure) ou éventuellement (en C mais pas en C++) qu'elle ne possède aucun paramètre.

Les entiers

Le type **int** définit les entiers "standards", mais leur taille (nombre de bits) dépend de l'implémentation.

Bien que nous préciserons les règles par la suite, voici à titre d'exemple la déclaration d'une variable entière appelée *monEntier*:

```
int monEntier;
```

On peut la faire précéder des modificateurs:

- **short**

short int représente un entier de taille plus petit ou égale à **int**.

En pratique **short int** peut s'abréger simplement **short**.

- **long**

long int représente un entier de taille plus grande ou égale à **int**.

En pratique **long int** peut s'abréger simplement **long**.

- **unsigned**

De plus on peut qualifier tous les entiers (**short**, **int** ou **long**) comme pouvant contenir des valeurs non signées en faisant précéder le type du mot-clé **unsigned**. Ainsi, si **int** utilise 16 bits et peut donc prendre des valeurs comprises entre -32768 et +32767, un **unsigned int** pourra prendre des valeurs entre 0 et 65535. Ce qui nous donne les combinaisons possibles suivantes: **unsigned short, unsigned short int, unsigned int, unsigned long** et finalement **unsigned long int**.

- **signed**

Par défaut tous les entiers sont signés; toutefois nous avons le droit de le préciser explicitement en faisant précéder leur déclaration du mot réservé **signed**. Ce qui nous donne les combinaisons: **signed short, signed short int, signed int, signed long** et **signed long int**.

Les caractères

Le type **char** définit les objets de type caractère.

Voici à titre d'exemple la déclaration d'une variable de type caractère appelée *monCaractere*:

```
char monCaractere;
```

Sans aucun problème nous pourrons affecter une valeur entière à une variable déclarée de type **char**, l'opération revenant à affecter à la variable le code du caractère désiré; l'inverse étant également possible.

Nous pouvons également qualifier les objets caractères comme étant signés ou non: **signed char, unsigned char**. Ainsi, si **signed char** contient en principe le code d'un caractère, il peut aussi contenir des valeurs entre -128 et +127 (... eh oui!) et donc, dans le cas d'un **unsigned char** des valeurs entre 0 et 255.

Mais **char** tout simplement, est-il signé ou non, cela dépend malheureusement du compilateur et peut donc provoquer des différences de comportement par exemple lors de comparaisons[1].

Bien que nous ne l'utilisions pas dans le cadre de ce cours, signalons l'existence d'un type de caractères étendus **wchar_t**, utilisant plus de bits pour leur représentation interne et permettant donc de coder plus de caractères.

Les réels

Nous disposons en réalité de 3 possibilités pour déclarer des objets d'un type réel: **float**, **double** et **long double**. Dans l'ordre où ils sont donnés ci-dessus, chacun offre une précision (un nombre de chiffres significatifs) plus grande et/ou un intervalle de validité plus étendu.

> **Attention**:
>
> Pour la suite, n'oubliez jamais que les réels flottants offrent une précision limitée, ce qui en pratique peut poser de très gros problèmes dans les développements!

[1] Cette situation posera d'autres problèmes lors de la surcharge.

Les booléens

Les objets de type booléen se déclarent en utilisant le mot réservé **bool**.

Exemple:

```
bool fini;
```

Les deux constantes prédéfinies de ce type font également l'objet de mots réservés: **false** et **true**.

C++ s'inspire directement de C pour ce type (avec toutes les conséquences que cela a!). En effet dans les anciennes versions de ce langage le type booléen n'existe pas en tant que tel et toute valeur non nulle représente vraie (**true**) et une valeur nulle (0) représente faux (**false**)[1].

Nous reviendrons très prochainement sur le type booléen, en introduisant les opérateurs qui lui sont associés.

Forme des déclarations

Les déclarations peuvent venir n'importe où dans le code, cela ne veut pas dire qu'il faut faire n'importe quoi et, pour des raisons de lisibilité, nous conseillons de ne pas abuser de cette possibilité. En tous les cas il faut faire des regroupements qui reflètent la logique de l'application!

L'ordre des déclarations peut être quelconque, toutefois utiliser un objet avant sa déclaration peut aboutir à une situation aléatoire désastreuse.

Une déclaration d'objet(s) d'un certain type est introduite par le nom du type suivi d'une liste d'identificateurs des objets de ce type.

```
nomDeType listeIdentificateurs = valeur;
```

La partie "= valeur" est optionnelle pour initialiser la variable lors de sa déclaration.

Une liste d'identificateurs peut se réduire à un seul identificateur:

```
int i1; // Rappel: on distingue majuscules/minuscules
```

ou une série d'identificateurs séparés les uns des autres par des virgules:

```
long int i1, i2, i3;
```

[1] La norme C99 a bien introduit le type **_Bool** et d'autres éléments liés non décrits ici, mais lui aussi en restant compatible avec les anciennes formes!

On devrait de préférence écrire:

```
long int i1,    // ...
         i2,    // ...
         i3;    // ...
```

ou

```
long int  i1;  // ...
long int  i2;  // ...
long int  i3;  // ...
```

On peut déclarer des objets d'un certain type, puis d'autres d'un autre type et revenir à des déclarations du premier type, l'ordre comme déjà indiqué n'a pas d'importance:

```
char  c1;      // ....
short int sh1; // ....
char  c2;      // ....
```

Bien entendu les identificateurs doivent être uniques (à un même niveau[1]).

Les objets peuvent être initialisés lors de leur déclaration. Il suffit de faire suivre l'identificateur de l'objet par "= expression " (seul l'objet qui précède est initialisé):

```
int i0 = 20;
int i1, i2 = 10, i3;
```

Dans la deuxième déclaration ci-dessus, seul *i2* reçoit la valeur 10; les variables *i1* et *i3* quant à elles correspondent à une valeur indéfinie, celle qui se trouve par hasard à l'emplacement mémoire attribué.

L'expression d'initialisation doit être du même type que l'objet ou d'un type jugé compatible et ne peut comporter que des objets déjà déclarés**:**

```
int i2 = 'A';  // Types compatibles
```

[1] Nous reviendrons sur cette notion!

Forme des constantes

Constantes caractères

Les constantes caractères s'écrivent simplement en mettant le caractère entre apostrophes.

Exemple: 'A'

Caractères spéciaux

Certains caractères de contrôle ou caractères spéciaux bénéficient d'une représentation particulière commençant toujours par un slash inverse.

En voici la liste:

'\a'	Produit une alerte sonore ou visuelle
'\n'	Passage à la ligne suivante (New Line)
'\t'	Tabulation (<TAB>)
'\v'	Tabulation verticale
'\b'	Espace arrière (Back Space)
'\r'	Retour chariot (Carriage Return)
'\f'	Saut de page (Form Feed)
'\\'	Le caractère slash inverse lui-même
'\''	Le caractère apostrophe
'\13'	Le caractère dont le code est 13 octal (13 dans cet exemple, en pratique, la valeur octale que vous désirez)
'\x13'	Le caractère dont le code est 13 hexadécimal (13 dans cet exemple, en pratique, la valeur hexadécimale que vous désirez)

Mis à part les 2 derniers cas de la liste, ces caractères particuliers peuvent également être utilisés dans les chaînes de caractères. Comme les autres caractères usuels, nous pouvons les affecter à une variable:

```
char tabulation = '\t';
```

> **Attention**:
> N'importe quel code peut être utilisé pour représenter les caractères (dépend du système). Ne faites donc pas d'hypothèse sur le code ASCII, comme vous en avez souvent l'habitude, même si en général c'est bien lui qu'on utilise!

Constantes chaînes de caractères

Notons que dans nos types de base, nous n'avons pas parlé de chaînes de caractères. En C/C++ ce ne sera rien d'autre que des tableaux de caractères avec des conventions d'utilisation dont nous reparlerons par la suite. Toutefois, nous pouvons déjà signaler qu'une constante chaîne s'écrit entre guillemets:

```
"Ceci est une chaine"
""   // C'est une chaine vide
```

Une chaîne se termine normalement avant la fin de la ligne; si elle s'avère trop longue, on peut terminer la première ligne par une barre inverse qui ne fait pas partie de la chaîne:

```
"Ceci n'est \
qu'une seule chaine"
```

Qui correspond à:

```
"Ceci n'est qu'une seule chaine"
```

Nous pouvons aussi utiliser la possibilité de fermer les guillemets à la première ligne et de les ouvrir à nouveau à la ligne suivante:

```
"Ceci n'est "
"qu'une seule chaine"
```

Ce qui s'avère bien pratique pour réaliser des alignements propres sans avoir des espaces entre les 2 parties de la chaîne.

Constantes entières

Elles ont, bien heureusement, leur forme habituelle.

```
Exemple: 127
```

Notons toutefois qu'il est possible de donner des valeurs en hexadécimal; il suffit pour cela de faire précéder cette valeur de "0x".

```
Exemple: 0x127, 0xABC, 0x1A
```

Ou en octal, il suffit de faire précéder la valeur d'un zéro de tête.

```
Exemple: 0127 00412
```

Attention: dès qu'il y a un zéro de tête il s'agit toujours d'une valeur exprimée en octal.

Par défaut les constantes entières décimales[1] sont du premier type qui permet de les contenir, à savoir dans l'ordre: **int**, **long int** ou **unsigned long int**.

Si l'on désire qu'une constante entière soit de type **long** alors que sa valeur a priori ne le nécessite pas, il suffit de faire suivre cette valeur de la lettre "L" ou "l". Dans les autres cas, et dans la mesure où la grandeur de la valeur le permet, le type **int** sera utilisé.

```
Exemples: 123L    9l
```

De plus, si le nombre est terminé par un "U" ou un "u", il sera interprété comme non signé (unsigned).

```
Exemples: 123U    9u
```

Une valeur peut aussi être forcée comme longue et non signée.

```
Exemples: 123LU    9lu
```

Nous pouvons aussi écrire.

```
Exemples: 123UL    9ul
```

Nous pouvons également forcer les constantes octales ou hexadécimales comme étant longues et/ou non signées

```
Exemples: 0x123LU    07u  0xal
```

[1] Si la constante est écrite sous forme octale ou hexadécimale l'ordre des types sera **int**, **unsigned int**, **long int**, **unsigned long int**

Constantes réelles

Elles aussi prennent une forme usuelle. Le point décimal doit être présent sauf éventuellement si la lettre "e" ou "E" est présente pour introduire une puissance de 10. Les constantes suivantes sont toutes correctes:

```
3.14    22.5E-3   .1000  1.     1.2F     9.5L   1e3[1]
```

En pratique elles sont toujours interprétées comme **double** par le compilateur à moins qu'on les fasse suivre des lettres F ou L (ou f, l), auquel cas elles correspondent respectivement à des **float** ou des **long double**.

❑ *Exemple*

Vous trouverez à la page suivante un exemple illustrant les principaux éléments présentés ci-dessus; l'exécution de ce code fournit les résultats suivants:

```
Programme types de base

Entier i1=62
Entier i2 ('A')=65
Entier hexadecimal(0xa, affiche en decimal)=10
Entier octal(0123, affiche en decimal)=83
Caractere(a3)=a
Caractere sp=
*
Fin du programme...Appuyez sur une touche pour continuer...
```

Voici le code:

[1] e3 serait incorrecte, il faut toujours une mantisse!

```
/*
   Programme exemple de declarations et introduction des types de
   base, initialisation des objets et impression
    TYPES_DE_BASE
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
    /* Les declarations */
{
  char   a1;        // declaration d'un caractere
  short  sh1,       // declaration d'un objet de type short
         sh2;       // ... suite de la declaration
  char   a2;        // on revient a une declaration de type char
  long   l1;        // declaration d'un objet d'un type long int
  float  f1;        // declaration d'un objet d'un type float
  double d1;        // declaration d'un objet d'un type double
  int    i1 = 62,   // declaration d'un entier avec initialisation
         i2 = 'A'; // autre declaration avec initialisation …
                   //                et c'est autorise!
  unsigned int u1; // declaration d'un entier non signe
  unsigned     u2; // abreviation pour unsigned int
  unsigned char a3 = 'a';
  long int li;      // suivant les compilateurs, peut etre egal
                    //        a int ou plus grand
  char sp  = '\12'; // definition d'un caractere par son code octal
  int hexa = 0xa;   // initialisation avec valeur hexadecimale
  int octa = 0123;  // initialisation avec valeur octale

  /*    Debut des instructions, exemple de sortie */
  cout << "Programme types de base\n\n";
  cout << "Entier i1=" << i1 << endl
       << "Entier i2 ('A')=" << i2 << endl;
  cout << "Entier hexadecimal(0xa, affiche en decimal)=" << hexa
       << endl << "Entier octal(0123, affiche en decimal)="
       << octa << endl;
  cout << "Caractere(a3)=" << a3 << endl << "Caractere sp="
       << sp << '*' << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Les programmes donnés ici sont des exemples pour le cours, mais vous, vous ne devez pas commenter les vôtres de la même manière!! Dans la réalité, les commentaires doivent apporter une information complémentaire utile pour des personnes sachant programmer!!

Définitions de constantes

"Constantes" const

Le mot **const** peut se mettre devant toute déclaration d'objet initialisé quelque soit son type.

Forme générale:

```
const type identificateur = valeur;
```

Exemple:

```
const int limite = 124;
```

Que nous pouvons aussi écrire:

```
int const limite = 124;
```

Mais attention:

```
const int limite1 = 1, limite2 = 2;
```

Ci-dessus *limite1* et *limite2* sont toutes 2 des constantes; par contre avec:

```
int const limite1 = 1, limite2 = 2;
```

seule *limite1* est une constante et *limite2* une variable.

Ceci s'appliquera aussi aux objets structurés ainsi qu'aux paramètres de sous-programmes (c.f. suite du cours).

Par définition, de tels objets ne peuvent plus changer de valeur; **const** est en fait un attribut qui indique au compilateur qu'il peut envisager certaines optimisations sur son utilisation et qu'il doit en principe faire le nécessaire pour que la valeur de l'objet ne puisse pas changer.

"Constantes" #define

Depuis les premières versions de C et aussi en C++, grâce à la substitution de texte traitée par le préprocesseur, nous disposons d'une possibilité plus générale: la directive *#define*; dont la structure est:

```
#define NOM texte de substitution
```

Rappelons que toute ligne qui commence comme premier caractère significatif par un # est traitée par un préprocesseur, ce qui explique bien la substitution qui va être faite. Dans le programme source, à la suite d'une telle ligne, toutes les occurrences de "NOM" (sauf dans les chaînes de caractères) sont remplacées par le *texte de substitution.*

Ceci permet, entre autres, de définir des *pseudo*s constantes:

```
...
#define MAX 3
...
int i = MAX;   // revient à int i = 3;
...
```

Mais plus généralement n'importe quelle chaîne:

```
...
#define SORTIE "Le résultat : "
...
cout << SORTIE << 12 << endl;
...
```

Et pourquoi pas toute une instruction:

```
...
#define INSTRUCTION cout << SORTIE << 12 << endl
...
INSTRUCTION;
...
```

Ici pour définir "INSTRUCTION" on a utilisé la définition faite au préalable de "SORTIE".

Nous aurons l'occasion de revenir plus en détails sur le *#define* et plus généralement sur le préprocesseur dans un chapitre spécifiquement consacré à ce dernier.

Attention: Evitez de mettre des commentaires à la suite d'un *#define*, ce commentaire ferait alors partie de la chaîne de substitution, ce qui peut poser de graves problèmes.

Exemple

Donnons un autre petit exemple, démontrant l'utilisation de #define:

```
/*
   Programme exemple: Utilisation de #define +lecture avec cin
   entrees/sorties. Definition de constantes, symboles.
    DEFINE
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  /* Definition de constantes, pseudo-symboles */
  #define VAL 62
  #define SORTIE "Le resultat: "
  #define INSTRUCTION cout << SORTIE << VAL << endl

  /* Definition de variables */
  float  f1 = 0.125e2; // declaration d'un objet d'un type float
  int    i1 = VAL;     // declaration d'un entier avec initialisation
  int    i2;           // pour lecture/ecriture

  /* Debut des instructions, exemple de sortie */
  cout << "Programme #define:\n\n";
  cout << "Un reel pour l'exemple (f1)=" << f1 << endl;
  cout << "... et l'entier (i1)=" << i1 << endl << endl;

  /* Utilisation des symboles, constantes */
  cout << SORTIE << VAL << endl;
  INSTRUCTION;

  /* Test de lecture et ecriture */
  cout << "\nDonnez une valeur entiere: ";
  cin >> i2;
  cout << "\nLa valeur que vous venez de donner: " << i2 << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Ce programme donne le résultat:

```
Programme #define:

Un reel pour l'exemple (f1)=12.5
... et l'entier (i1)=62

Le resultat: 62
Le resultat: 62

Donnez une valeur entiere: 33

La valeur que vous venez de donner: 33
Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *Les bases de la lecture*

En conclusion sur cet exemple signalons l'utilisation de nos premières lectures par l'intermédiaire du flux *cin*. Nous ne donnerons que peu d'informations pour l'instant sur l'utilisation de ce flux. Pour lire une valeur, *cin* est suivi de l'opérateur >> et d'une variable. Exemple:

```
cin >> i;
```

Nous pouvons lire, en une seule opération, plusieurs variables:

```
cin >> i >> k >> l;
```

La lecture se fait en format libre, c'est-à-dire:

- Les espaces, fins de lignes et tabulations de tête sont sautés.
- Ensuite on doit pouvoir commencer à construire une valeur du type défini par celui de la variable.
- La lecture s'arrête sur le premier caractère qui ne permet plus de construire la valeur en question.

Voilà pour l'instant nous n'en dirons pas plus sur le sujet, mais ceci sera développé par la suite.

❑ *Les références*[1]

On peut considérer une référence comme étant le synonyme (alias) d'un autre objet qui peut par ailleurs posséder plusieurs autres références.

La forme générale d'une déclaration de référence:

```
type &identificateur = identificateurSource;
```

avec:

type: un type identique à celui de l'*identificateurSource*.

identificateur: le nouveau nom par lequel vous voulez pouvoir référencer l'*identificateurSource*. Le symbole **&** doit le précéder.

identificateurSource: l'objet du type *type* que l'on veut aussi pouvoir désigner par le nom *identificateur*.

Comme pour les constantes, une telle déclaration doit obligatoirement posséder une partie initialisation puisqu'elle ne crée pas à proprement parler un nouvel objet, mais une référence à un objet existant.

Ce qu'il faut comprendre c'est que la référence ainsi créée désigne le même objet que sa source, en fait il n'y a qu'un objet.

Plutôt que de longs discours, voici un tout petit programme illustrant cette possibilité et devant vous permettre de comprendre le mécanisme:

[1] Cette notion n'existe pas en C

```
/*
  Exemple introduisant la notion de reference
    REFERENCE
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int  main ( )
{
  int i = 10;
  int &ref_i = i;  // ref_i : reference (alias) de i
  cout << "i=" << i << "  ref_i=" << ref_i << endl;
  ref_i = i + ref_i;
  cout << "i=" << i << "  ref_i=" << ref_i << endl;
  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Résultat de l'exécution du programme:

```
i=10  ref_i=10
i=20  ref_i=20
Fin du programme...Appuyez sur une touche pour continuer...
```

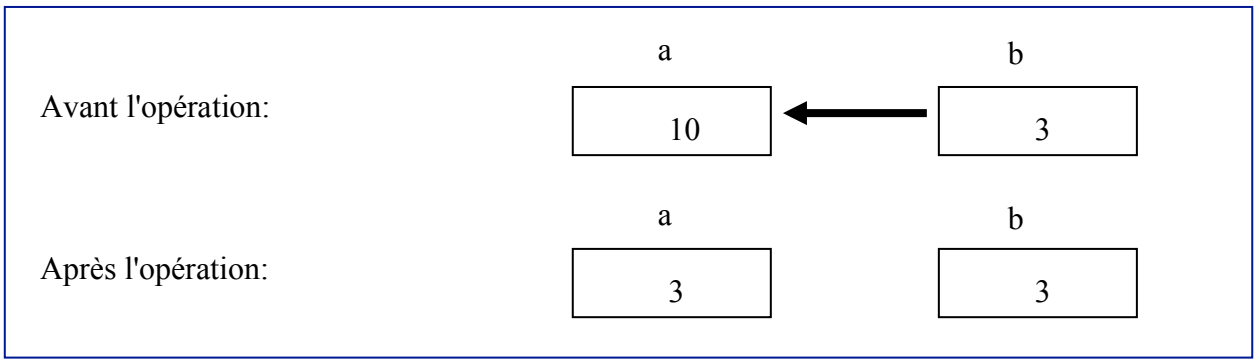
Opérateurs et expressions

Considérations générales/ Affectation simple

Comme son ancêtre C, C++ s'avère riche en opérateurs (les mêmes plus quelques autres), ce qui permet souvent d'écrire un code très compact, mais aussi parfois difficile à relire. Il possède des opérateurs unaires, binaires et un ternaire (travaillant respectivement avec 1, 2 ou 3 opérandes). Tout opérateur livre bien entendu un résultat; il en va de même pour l'affectation considérée comme un opérateur (en fait, comme nous le verrons, il y a plusieurs opérateurs d'affectation). Le résultat d'une affectation correspond à la valeur affectée, cela deviendra important dans la suite pour la notion d'expression généralisée que l'on utilise essentiellement avec les structures de contrôle. Dans ce chapitre nous présenterons une grande partie des opérateurs à disposition, toutefois, certains d'entre eux plus spécialisés ne seront abordés qu'ultérieurement.

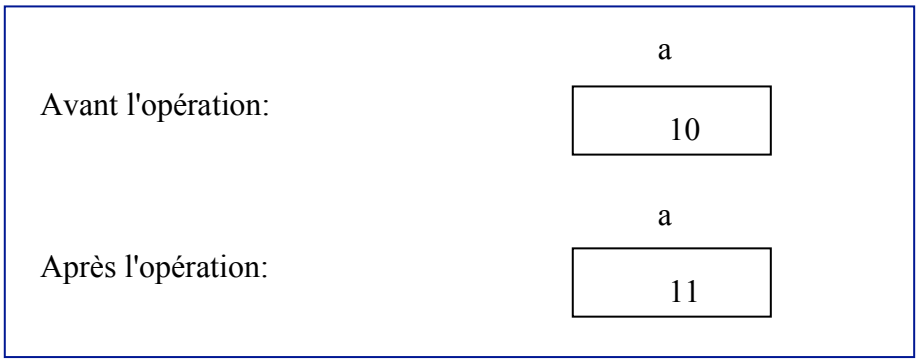
Ne confondez pas le "=" de l'affectation en programmation avec le "=" au sens mathématique. En programmation il faut le prendre comme une opération dynamique, l'expression à droite du signe est d'abord évaluée et on affecte ensuite le résultat de cette évaluation à la variable à gauche du signe (pour cette variable on parlera de *lvalue* c'est-à-dire: un objet pouvant changer de valeur!).

Exemple: `a = b;`



L'aspect dynamique nous permet d'écrire par exemple:

```
a = a + 1;  // Pas de sens en mathématique
```



Nous pouvons déjà introduire la notion d'affectation multiple:

```
v1 = v2 = ... = vn = expression;
```

L'expression est d'abord évaluée et le résultat affecté à *vn* et ensuite, dans l'ordre de droite à gauche aux autres variables.

Donc, même si ici cela n'a aucune importance (ce ne sera pas toujours le cas par la suite):

```
v1 = v2 = v3 = 10;
```

Revient à:

```
v3 = 10;
v2 = v3;
v1 = v2;
```

Nous pouvons aussi écrire des expressions du genre:

```
v1 =  ( v2 = 3 ) + 2;
```

On affecte ici la valeur 3 à *v2* et la valeur 5 à *v1*.

Attention:

Relevons aussi que ce que l'on appelle généralement "instruction d'affectation" dans d'autres langages, devient en C++ une expression (apparemment c'est un détail, mais il a une grande importance pour la compréhension de certains points dans la suite).

L'opération d'enchaînement

Notons aussi, pour son importance par la suite, que l'on dispose d'un opérateur dit d'enchaînement ou d'évaluation séquentielle, la virgule:

```
expression_1, expression_2, ...;
```

Par exemple: `v1 = i + 2,  v2 = v1 + 3;`

En pratique cela revient à dire que nous avons 2 sous-expressions qui globalement n'en forment qu'une seule. Cela, nous permettra plus tard, de faire par exemple des initialisations multiples lors de l'écriture d'une instruction de boucle, ou de garantir qu'une partie d'expression est totalement évaluée, avant de passer à la suite. N'abusez pas de l'utilisation de cet opérateur qui peut rendre difficile la relecture du code.

Cet opérateur étant évalué de gauche à droite, l'exemple ci-dessus revient au même que d'écrire:

```
v1 = i + 2;
v2 = v1 + 3;
```

Les opérations arithmétiques

Il s'agit dans l'ordre des priorités décroissantes de:

-	Moins unaire	+	Plus unaire		
*	Multiplication	/	Division	%	Modulo
+	Addition	-	Soustraction		

Remarques:

- Des parenthèses peuvent toujours modifier cet ordre d'évaluation.
- Le "/" représente indifféremment la division entière et la division réelle, le type des opérandes déterminant la nature de l'opération. Attention toutefois, s'il n'y a pas de problème pour 2 opérandes positifs (le résultat est la valeur approchée par défaut du

quotient), la norme ne précise pas ce qui se passe si l'un ou les 2 opérandes sont négatifs; ainsi 7 / -4 peut donner comme résultat -1 ou -2!

- L'opérateur "%" ne s'applique qu'à des opérandes de type entier (**char** est à considérer comme faisant partie des types entiers). Même remarque que pour la division sur la valeur du résultat si l'un au moins des opérandes est négatif!
- Le "+" unaire ne s'utilise que rarement.
- L'opérateur "=" (puisque cela en est un) a heureusement une priorité inférieure à celle des opérateurs arithmétiques. Globalement, il aura même l'avant-dernier niveau de priorité, juste devant l'opérateur ",". Le résultat fourni par cet opérateur est la valeur affectée!
- Aucun débordement n'est signalé, ainsi l'addition ou la multiplication de 2 valeurs entières positives peut très bien livrer comme résultat une valeur négative.

❑ *Conversions de types*

L'arithmétique mixte entre opérandes de types numériques différents est tout à fait possible. Il suffit simplement de se rappeler que l'opérande de type "le plus faible" est implicitement converti dans le type de l'opérande "le plus fort". Il faut comprendre par type "le plus fort", celui qui englobe les valeurs de l'autre type:

```
char      => int
int       => float
float     => double
etc.
```

Lors de l'affectation il y a conversion automatique dans le type de la variable destination si l'opération peut toujours se réaliser sans problème; attention alors au débordement ou à la perte de chiffres significatifs en passant d'un **double** à un **float**! Votre compilateur devrait vous signaler une erreur[1], ou tout au moins un avertissement si vous tentez de réaliser une affectation potentiellement dangereuse, c'est-à-dire d'un type "plus large" vers un type "plus petit"; une valeur réelle affectée à une variable entière par exemple! Vous devez dans ce cas là réaliser une conversion explicite, expliquée partiellement ci-dessous. Cette forme ne garantira pas qu'il n'y aura aucun problème, mais on espère ainsi sensibiliser le programmeur à l'aspect dangereux et donc qu'il aura bien réfléchi aux conséquences du code qu'il réalise.

[1] Ceci n'est pas le cas en C qui accepte parfaitement la conversion implicite, ce qui bien entendu ne veut pas dire qu'il ne peut pas y avoir de problèmes.

Conversions explicites

Il existe en C++ 2 formulations pour réaliser une conversion de types.

L'opérateur cast

La première, appelée dans la définition du langage "cast", prend la forme:

```
( type ) expression
```

Par exemple:

```
( int ) 13.5
```

Donc il convertit l'expression dans le type précisé entre parenthèses.

Comme déjà signalé, dans le sens:

char -> **int** -> **long** -> **float** -> **double**

les conversions ne posent aucun problème, mais pour d'autres conversions, il faut être prudent:

- Perte de chiffres significatifs entre **double** et **float**.
- Perte des bits de poids forts entre **int** et **short**.
- Résultat imprévisible (peut changer d'un compilateur à un autre) entre valeurs signées et non signées.
- Pour les conversions de réels à entiers, c'est toujours une valeur tronquée qui est livrée.

Note:

En réalité, aucun opérateur arithmétique ne s'applique directement aux opérandes de type **char** ou **short**; ils sont automatiquement convertis en **int** pour réaliser l'opération.

La forme fonctionnelle[1]

La deuxième possibilité prend la forme d'une fonction[2]:

```
type ( expression )
```

Par exemple:

```
int ( 13.5 )  //[3]
```

On utilise le nom du type destination comme une fonction.

Les mêmes remarques que celles faites pour l'opérateur cast s'appliquent aussi ici!

Exemple:

```
/*
   Programme exemple: Utilisation d'operateurs, partie I
     OPERATEURS1
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  int   v1, v2, v3; // Quelques variables de travail
  char  ch = 'A';
  float f1 = 12.5; // Formellement: f1 = 12.5F;

  /* Possibilite d'affectations multiples */
  v1 = v2 = v3 = 12;
  cout << "Resultat de l'affectation multiple:\n"
       << " v1=" << v1 << " v2=" << v2 << " v3=" << v3 << endl;
```

[1] Cette possibilité n'existe pas en C.

[2] Sous cette forme, puisqu'il s'agit d'une "fonction" vous ne pouvez pas écrire **long int** (…); mais dans le cas présent vous pouvez très bien écrire tout simplement **long** (…).

[3] Forme à ne pas utiliser pour des types faisant l'objet d'identificateurs composés comme **unsigned int** par exemple!

```
  /* Operateur d'enchainement: la virgule */
  v1 = v2 + v3, v3 = 2;
  cout << "Resultat apres enchainement:\n"
       << " v1=" << v1 << " v2=" << v2 << " v3=" << v3 << endl;

  /* Utilisation des operateurs arithmetiques usuels */
  v2 = v1 % v3;
  v1 = v1 + v2 * -3;
  cout << "Resultat d'operateurs arithmetiques simples:\n"
       << " v1=" << v1 << " v2=" << v2 << " v3=" << v3 << endl;

  /* Exemples de conversions de types */
  v1 = int (f1);                        // Conversion explicite
  v2 = ch;                              // Conversion implicite
  v3 = v1 / v2;                         // Division entiere
  f1 = ( float ) v1 / ( float ) v2;   /* Conversion explicite
                                          plus division reelle  */
  cout << "Resultat de conversions de types:\n"
       << " v1=" << v1 << " v2=" << v2 << " v3=" << v3
       << " f1=" << f1 << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Ce programme donne le résultat suivant:

```
Resultat de l'affectation multiple:
 v1=12 v2=12 v3=12
Resultat apres enchainement:
 v1=24 v2=12 v3=2
Resultat d'operateurs arithmetiques simples:
 v1=24 v2=0 v3=2
Resultat de conversions de types:
 v1=12 v2=65 v3=0 f1=0.184615
Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *Opérateurs de comparaisons*

Un opérateur de comparaisons donne un résultat vrai si la relation est satisfaite et faux sinon. Toutefois, comme nous l'avons déjà relevé, en C "traditionnel" le type booléen n'existe pas en tant que tel. Par définition une valeur nulle équivaut à faux et toute autre valeur à vrai. Le résultat de la relation donnera 0 si elle n'est pas satisfaite et 1 sinon.

Bien qu'ayant introduit le type **bool**, C++ restant compatible, ce qui précède reste toujours vrai. Ceci fait qu'un booléen, bien que ne prenant que les valeurs **false** (0) ou **true** (1) peut sans autre se combiner avec des variables numériques; à vous de savoir ce que vous faites!!!

Voici tout d'abord un tout petit programme illustrant les possibilités "limites" d'utilisation du type **bool**; regardez-le très attentivement avec les résultats fournis par l'exécution; nous formulerons quelques explications complémentaires après:

```
/*
  Exemple d'utilisation de booleen
    BOOLEENS
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int  main (  )
{
  /* Definition de variables locales au programme principal */
  bool b1 = true, b2 = false;
  bool b3 = 12;
  b2 = 0;
  b1 = b1 + b2 + b3 + 3;
  cout << "b1=" << b1 << endl << "b2=" << b2 << endl
       << "b3=" << b3 << endl << "b3+12=" << b3+12 << endl;
  cout << "12+b1>b2=" << (12+b1>b2) << endl;
  cout << "12+(b1>b2)=" << (12+(b1>b2)) << endl;
  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution de ce programme nous affiche:

```
b1=1
b2=0
b3=1
b3+12=13
12+b1>b2=1
12+(b1>b2)=13
Fin du programme...Appuyez sur une touche pour continuer...
```

Constatations:

- Une variable **bool** contient toujours l'une des valeurs 0 (**false**) ou 1 (**true**), même si nous lui affectons explicitement une autre valeur.
- Notez que rien ne vous interdit (sauf peut-être la logique!) de faire de l'arithmétique avec ces booléens, mais attention alors à la priorité des opérateurs (nous préciserons les choses par la suite) et donc au résultat obtenu; cf. les 2 expressions de notre exemple: `12+b1>b2=1` et `12+(b1>b2)=13`.
- Notez tout particulièrement dans cet exemple les parenthèses que nous avons impérativement dues utiliser afin d'éviter des problèmes de priorité avec les "<<" utilisés pour le *cout*!

Revenons à nos opérateurs de relations, il s'agit de:

<	Inférieur
<=	Inférieur ou égal
==	Egal (Attention 2 fois "=" pour le différencier de l'affectation)
!=	Différent de (Attention "!=" l'écriture change dans beaucoup de langages); peut aussi être remplacé par le mot réservé **not_eq**[1]
>=	Supérieur ou égal
>	Supérieur

Les opérateurs de comparaisons possèdent une priorité inférieure aux opérateurs arithmétiques, ce qui permet d'écrire sans parenthèses des expressions du type:

```
i = a + b == c - d;
```

Si i est de type **int** on peut très bien lui affecter ce résultat (0 ou 1). Notez bien la différence entre le premier "**=**" pour l'affectation et le "**==**" pour la comparaison; ce point est très souvent une source d'erreur dans les programmes.

En réalité dans la définition du langage "==" et "!=" (**not_eq**) ont une priorité inférieure aux 4 autres opérateurs de comparaisons.

[1] En C **not_eq** n'est pas un mot réservé, toutefois on le retrouve généralement sous la forme d'une définition de macro (c.f. chapitre du préprocesseur).

Attention aux expressions du genre: x<y<z ➔ (x<y)<z ➔ (0 ou 1)<z; si le but est de savoir si y est compris entre x et z, il faut écrire: (x<y) && (y<z) ou x<y && y<z.

Pour les réels, on ne devrait en principe jamais comparer `a==b` en raison des valeurs approchées. Il faudrait de préférence utiliser une comparaison du genre abs `(a-b)<epsilon`[1] ou epsilon représente une valeur suffisamment petite pour que l'on admette l'égalité.

❑ *Les opérateurs logiques*

On peut combiner des relations pour former des expressions booléennes, toujours en considérant que ce type donne un résultat 0 (pour faux) et 1 (pour vrai). Les opérateurs logiques sont:

!	non (unaire) inverse	ou[2]	**not**
&&	et (binaire)	ou	**and**
||	ou (binaire)	ou	**or**

Ils s'appliquent à tout opérande de type scalaire (caractère, entier, réel). La priorité des opérateurs logiques binaires est inférieure à celle des opérateurs de comparaison, ce qui permet d'écrire sans parenthèses des expressions du genre:

```
i = a <= b || c > d;
```

Ou encore:

```
i = i + 3 > j - 2 && a + b <= c;
```

Dans la définition du langage, les 3 opérateurs logiques sont à un niveau de priorité différent, soit d'abord "&&" et ensuite "||". Quant à l'opérateur "!" il se situe au même niveau que le "-" unaire et d'autres opérateurs que nous verrons par la suite.

Notes:

- Etant donné les relativement faibles caractéristiques de typage:

```
!i est équivalent à i == 0
```

[1] abs est une fonction qui nous viendra de la bibliothèque.

[2] En C la forme mot réservé de ces opérateurs n'existe pas, toutefois on les retrouve généralement sous la forme d'une définition de macro.

- Bien plus important, l'évaluation d'une expression logique est abandonnée dès que le résultat ne peut plus changer (pour ces opérateurs, on a la certitude que l'opérande gauche est toujours évalué avant l'opérande de droite):
    - Si l'opérande gauche d'un "&&" est faux, la partie droite n'est pas évaluée.
    - Si l'opérande gauche d'un "||" est vrai, la partie droite n'est pas évaluée.

Ceci sera important pour parcourir un tableau jusqu'aux limites, mais sans déborder (ou une liste chaînée). Mais alors, attention aux effets de bord qui ne seront pas réalisés, et le langage en offre de nombreuses possibilités.

❑ *Les opérateurs de manipulation de bits*

Les opérateurs de manipulation de bits se classent en 2 catégories:

- Les décalages (gauche ou droite).
- Les opérations logiques bit à bit.

Ils s'appliquent tous sur des opérandes de type entier (standard, long ou court, char).

❑ Les opérateurs de décalage

Ils sont 2 soit:

```
<<   Décalage à gauche
>>   Décalage à droite
```

Exemples:

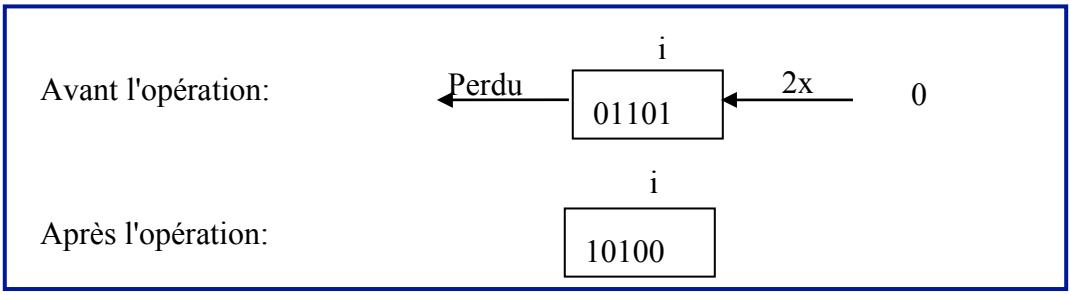
```
i = i << 2;
```

Décale les bits représentant la valeur de i de 2 positions vers la gauche, des zéros entrant à droite. Cette opération revient à multiplier i par 4 (gain de performance!?!). Plus généralement à multiplier la valeur de l'opérande gauche par 2 à la puissance de la valeur de l'opérande droite.

Notre exemple:

```
i = i << 2;
```

Sur un mot de 5 bits, généralisable à n bits:



Dans l'autre sens:

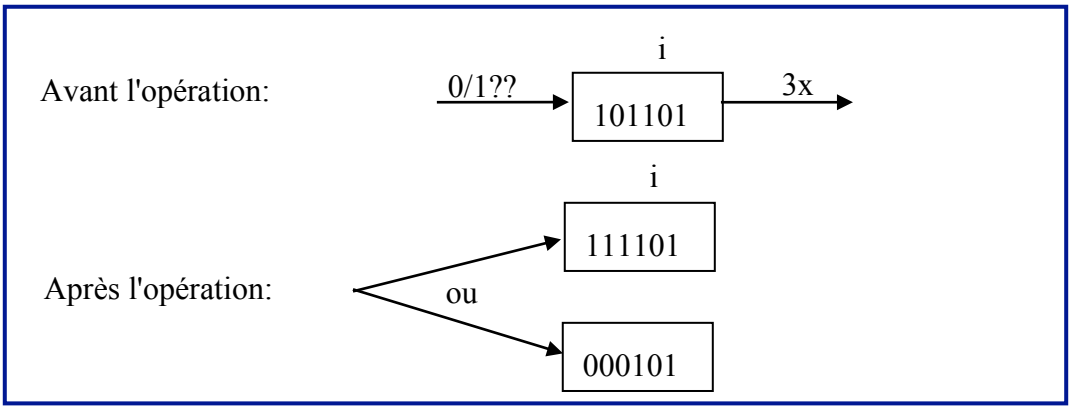
```
i = i >> 3;
```

Décale les bits représentant la valeur de *i* de 3 positions vers la droite, ce qui pour une valeur non signée revient à une division par 8 (2^3). Toutefois, ce qui entre à gauche peut dépendre du compilateur; ce peut être soit des zéros, soit l'extension du bit de signe pour le conserver. En d'autres termes, il faut être très prudent avec le décalage à droite sur une valeur signée, ce qui donne un résultat non portable!

Notre exemple:

```
i = i >> 3;
```

Sur un mot de 6 bits, généralisable à n bits:



La priorité des opérateurs de décalages se situe entre celle des opérateurs arithmétiques et celle des opérateurs de comparaisons.

Notez bien que la valeur de l'opérande gauche d'un opérateur de décalage n'est pas modifiée!

Opérateurs logiques bit à bit

But: appliquer les opérateurs sur l'ensemble des bits des opérandes.

Il s'agit de:

~	Inversion ou complémentation	(unaire) ou[1]	**compl**
&	Et	(binaire) ou	**bitand**
|	Ou	(binaire) ou	**bitor**
^	Ou exclusif	(binaire) ou	**xor**

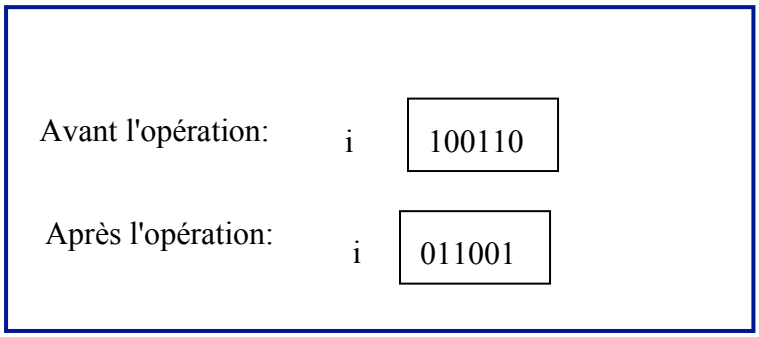
A noter que ces opérateurs s'écrivent à l'aide d'un seul symbole contrairement aux opérateurs logiques simples qui eux utilisent 2 symboles.

~ (ou **compl**) Inverse tous les bits de la valeur entière:

A	~A compl A
0	1
1	0

Exemple sur 6 bits:

```
i =  ~ i;
```
[2]



[1] En C la forme mot réservé de ces opérateurs n'existe pas, toutefois on les retrouve généralement sous la forme d'une définition de macro.

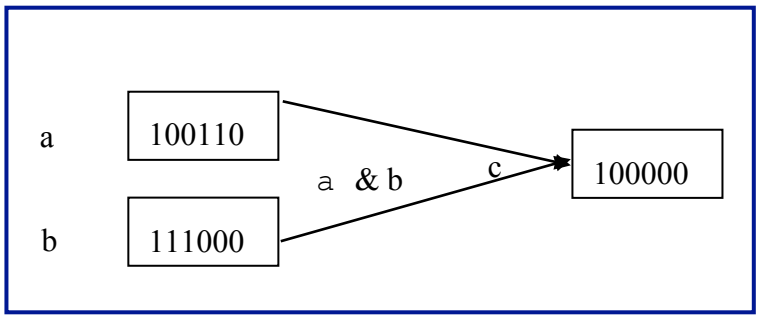
[2] revient à i = -i - 1

& (ou **bitand**) Réalise un "et" logique bit à bit entre les bits de position correspondante de ses 2 opérandes:

A	B	A&B A bitand B
1	1	1
1	0	0
0	1	0
0	0	0

Exemple sur 6 bits:

```
c = a & b;
```

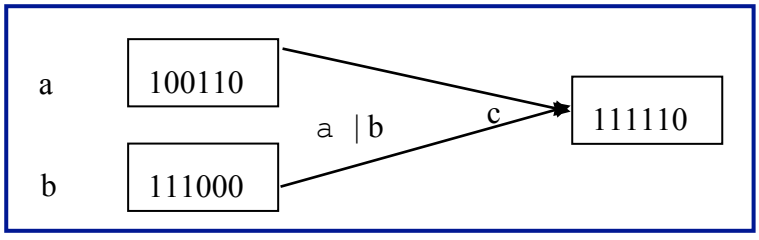


| (ou **bitor**) Réalise un "ou" logique bit à bit entre les bits de position correspondante de ses 2 opérandes:

A	B	A|B A bitor B
1	1	1
1	0	1
0	1	1
0	0	0

Exemple sur 6 bits:

```
c = a | b;
```

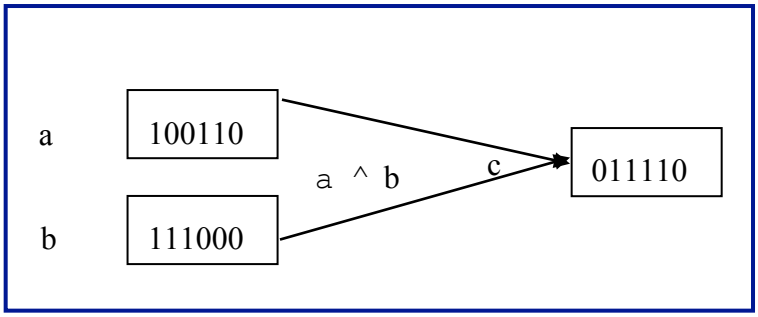


^ (ou **xor**) Réalise un "ou exclusif" logique bit à bit entre les bits de position correspondante de ses 2 opérandes:

A	B	A^B A xor B
1	1	0
1	0	1
0	1	1
0	0	0

Exemple sur 6 bits:

```
c = a ^ b;
```



Le ~ (**compl**) se situe au même niveau de priorité que tous les opérateurs unaires. Pour les autres opérateurs logiques bit à bit, leur priorité se situe entre les opérateurs de comparaisons et les opérateurs logiques simples, mais en fait tous à un niveau différent, soit dans l'ordre décroissant "&", "^", " | ".

Comme nous l'avons déjà dit les opérateurs sont nombreux et leurs niveaux de priorité aussi. Ne vous inquiétez pas trop, nous vous donnerons bientôt un tableau récapitulatif de ces propriétés.

Application

Pour mettre le N^{ième} bit (le bit de poids faible étant en position 0) d'une variable entière *i* à 0 on peut écrire:

```
i = i & ~ ( 1 << N );    // Evident, non?
```

ou

```
i = i bitand compl ( 1 << N );    // Evident, non?
```

Dans une recherche dichotomique, pour trouver le point milieu:

```
milieu = ( gauche + droite ) >> 1; // Pour aller plus vite?
```

En pratique, les opérateurs sur les bits s'utilisent essentiellement pour la programmation système (n'oubliez pas les origines du langage). Il y a que très rarement des combinaisons de ces opérateurs avec d'autres, mis à part l'affectation et les comparaisons.

Exemple

```
/*
   Programme exemple: Utilisation d'operateurs, partie II
    OPERATEURS2
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  int v1 = 1, v2 = 2, v3 = 3; // Quelques variables

  // Exemples d'operateurs de comparaison
  cout << "Resultat de comparaisons:\n"
       << "v1==v2: " << (v1==v2) << "  v3==v3: "
       << (v3==v3) << endl;

  // Exemples d'operateurs logiques
  cout << "Resultat de la negation: !v3: " << !v3 << endl;
  cout << "Resultat du et logique: v1<v2 && v3>v2: "
       << (v1<v2 && v3>v2) << endl;
```

```
  cout << "Resultat du ou logique: v3||v2: " << (v3||v2)
       << endl;

  /*
    Exemple d'operateurs de decalages
    La premiere forme est deconseillee mais permise
  */
  cout << "Resultat du decalage a droite: -16>>3: "
       << (-16>>3) << endl;
  cout << "Resultat du decalage a gauche: -2<<3: " << (-2<<3)
       << endl;

  // Exemples d'operateurs logiques bit a bit
  cout << "\nResultat des operateurs logiques bit a bit:\n";
  cout << "Resultat de l'inversion: ~v3: " << ~v3 << endl;
  cout << "Resultat du et: v1&v3: " << (v1&v3) << endl;
  cout << "Resultat du ou: v1|v3: " << (v1|v3) << endl;
  cout << "Resultat du ou exclusif: v1^v3: " << (v1^v3) << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Ce programme donne le résultat:

```
  Resultat de comparaisons:
  v1==v2: 0  v3==v3: 1
  Resultat de la negation: !v3: 0
  Resultat du et logique: v1<v2 && v3>v2: 1
  Resultat du ou logique: v3||v2: 1
  Resultat du decalage a droite: -16>>3: -2
  Resultat du decalage a gauche: -2<<3: -16

  Resultat des operateurs logiques bit a bit:
  Resultat de l'inversion: ~v3: -4
  Resultat du et: v1&v3: 1
  Resultat du ou: v1|v3: 3
  Resultat du ou exclusif: v1^v3: 2
  Fin du programme...Appuyez sur une touche pour continuer...
```

Puisque plusieurs opérateurs de ce groupe possèdent 2 formes, donnons une autre version de ce programme, utilisant la forme "mots réservés" de certains opérateurs; bien entendu cette version livrera les mêmes résultats. La voici:

```
/*
   Programme exemple: Utilisation d'operateurs, partie II
    OPERATEURS2b
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  int v1 = 1, v2 = 2, v3 = 3; // Quelques variables

  // Exemples d'operateurs de comparaison
  cout << "Resultat de comparaisons:\n"
       << "v1==v2: " << (v1==v2) << "  v3==v3: " << (v3==v3) << endl;

  // Exemples d'operateurs logiques
  cout << "Resultat de la negation: not v3: " <<  not v3 << endl;
  cout << "Resultat du et logique: v1<v2 and v3>v2: "
         << ( v1<v2 and v3>v2 ) << endl;

  cout << "Resultat du ou logique: v3 or v2: " << (v3 or v2) << endl;

  /*
    Exemple d'operateurs de decalages
    La premiere forme est deconseillee mais permise
  */
  cout << "Resultat du decalage a droite: -16>>3: "
         << (-16>>3) << endl;
  cout << "Resultat du decalage a gauche: -2<<3: " << (-2<<3) << endl;

  // Exemples d'operateurs logiques bit a bit
  cout << "\nResultat des operateurs logiques bit a bit:\n";
  cout << "Resultat de l'inversion: compl v3: " << compl v3 << endl;
  cout << "Resultat du et: v1 bitand v3: " << (v1 bitand v3) << endl;
  cout << "Resultat du ou: v1 bitor v3: " << (v1 bitor v3) << endl;
  cout << "Resultat du ou exclusif: v1 xor v3: "
         << (v1 xor v3) << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'opérateur conditionnel

Nous n'avons que peu l'habitude d'un tel opérateur qui est d'ailleurs le seul ternaire du langage.

Sa syntaxe:

```
condition ? expression_1 : expression_2
```

Et sa sémantique:

Livre comme résultat *expression_1* si la condition est vraie (!=0) et livre *expression_2* si la condition est fausse (==0). Ces 2 valeurs peuvent être des expressions quelconques.

Exemple:

```
i = a > b ? a : b + 1;
```

Qui a pour effet d'affecter à *i* la valeur de *a* si *a* > *b* et *b+1* sinon.

Nous aurions pu aussi écrire:

```
cout << "Le plus grand:" << (a > b ? a : b) << endl;
```

Pour afficher la plus grande de ces 2 valeurs.

L'opérateur est généralement globalement symbolisé par: " ?: ".

Sa priorité se situe entre les opérateurs logiques et les opérateurs d'affectation.

Exemple pratique d'utilisation:

```
cout << i << " valeur" << (i > 1 ? "s" : "");
```

N'abusez pas de cet opérateur, surtout pas dans des formes imbriquées; il rend les codes peu lisibles et généralement on le remplace facilement par une autre structure!

❑ *Les opérateurs d'incrémentation et de décrémentation*

Là aussi, une possibilité à laquelle nous ne sommes pas habitués dans des langages d'autres familles que C/C++, sauf éventuellement au niveau des langages d'assemblage pour refléter certains modes d'adressage.

Dans une expression, chaque variable (lvalue entière, pointeur par la suite et éventuellement réels, mais peu recommandé pour eux!) peut être affectée d'un effet secondaire d'incrémentation (++) ou de décrémentation (--); de plus cet effet secondaire peut être provoqué avant l'utilisation de l'objet ou après son utilisation; il suffit de mettre l'opérateur respectivement devant ou derrière l'identificateur de l'objet.

Ceci nous donne donc 4 possibilités:

```
++identificateur
identificateur++
--identificateur
identificateur--
```

Il peut s'utiliser directement comme instruction:

```
i++;
```

Qui dans le cas présent revient au même que:

```
++i;
```

Ou encore:

```
i = i + 1;
```

Et il y a encore d'autres possibilités ...

Ici, l'opérateur s'utilise comme effet principal de l'instruction, mais il peut aussi n'avoir qu'un effet secondaire dans une instruction principale:

```
i = j++ * 3;
```

Qui a 2 conséquences:

- *i* prend pour valeur l'ancienne valeur de *j* multipliée par 3
- *j* prend pour valeur son ancienne valeur plus 1

Mais **attention**, dans le cas d'une expression du genre:

`i || j++` *j* ne sera pas incrémenté si *i* est différent de 0 (vrai!).

Ces opérateurs s'utilisent typiquement dans le cadre de contrôle de boucles ou pour parcourir des indices de tableaux.

Il s'agit d'opérateurs unaires, ils ont un même niveau de priorité que les autres opérateurs unaires.

La norme ne précise pas rigoureusement quand l'effet secondaire se produit; exemple avec un élément de tableau (que nous étudierons plus tard!):

```
a [i] = i++;
```

Si avant cette instruction *i* vaut 3, suivant les compilateurs elle correspond à:

```
a [3] = 3;
```

Ou à: `a [4] = 3;`

Alors, prudence!

❑ *Les opérateurs d'affectations*

Et oui, il n'y a pas qu'une seule opération d'affectation car elle peut se lier à des effets secondaires en rapport avec d'autres opérateurs déjà étudiés. En fait, la forme simple d'une affectation:

```
variable = expression;
```

Peut se combiner avec les opérateurs:

`+ - * / % << >> & ^ |` (c'est-à-dire les opérateurs binaires arithmétiques et les opérateurs de manipulation de bits).

Pour donner:

`variable += expression;`	`=> variable = variable + expression;`
`variable -= expression;`	`=> variable = variable - expression;`
`variable *= expression;`	`=> variable = variable * expression;`
`variable /= expression;`	`=> variable = variable / expression;`
`variable %= expression;`	`=> variable = variable % expression;`
`variable <<= expression;`	`=> variable = variable << expression;`
`variable >>= expression;`	`=> variable = variable >> expression;`
`variable &= expression;`	`=> variable = variable & expression;`
`variable ^= expression;`	`=> variable = variable ^ expression;`
`variable |= expression;`	`=> variable = variable | expression;`

Exemple, si avant l'opération *i* vaut 2, après:

```
i *= 3;
```

i vaut 6 car l'opération est équivalente à:

```
i = i * 3;
```

La priorité est identique pour tous ces opérateurs, très basse, entre l'opérateur conditionnel et l'opérateur virgule.

Sous cette forme l'éventuelle expression pour calculer l'adresse de la variable (tableaux, pointeurs) n'est évaluée qu'une seule fois. Ceci se révèle important en relation avec les effets secondaires.

Voilà, il n'y a pas grand-chose à ajouter, si ce n'est qu'ainsi on espère que le compilateur générera un code plus efficace; mais un bon compilateur peut aussi générer un code efficace depuis l'autre forme!

Opérateur sizeof

L'opérateur **sizeof** s'applique, soit à une `expression`:

```
sizeof expression
```

Soit à un type mis entre parenthèses:

```
sizeof ( nom_de_type )
```

Il livre comme résultat une valeur entière représentant le nombre d'octets occupés en mémoire par l'objet ou par un objet du type spécifié. Cet opérateur se révèle utile pour résoudre certains problèmes de portabilité.

Si l'*expression* se réduit souvent à une variable, il peut toutefois s'avérer intéressant de connaître la taille du résultat d'une expression plus complexe, ce qui n'est pas toujours évident à déterminer avec le jeu des conversions implicites de types! Notez aussi, puisque l'on a toujours le droit d'ajouter des parenthèses inutiles, qu'*expression* peut aussi être mise entre parenthèses!

Signalons encore qu'*expression* n'est pas évaluée pour en déterminer la taille, donc:

```
sizeof variable++
```

est une expression correcte, mais variable ne sera pas modifiée!

Exemple dans notre environnement:

```
char c;
cout << sizeof c;
cout << sizeof (c+1);
```

affichera respectivement 1 et 4 (dans notre environnement)!!!

Il s'agit d'un opérateur unaire, et il se place au même niveau de priorité que tous les autres opérateurs unaires.

Exemple

```
/*
   Programme exemple: Utilisation d'operateurs, partie III
   OPERATEURS3
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  int v1 = 1, v2 = 2, v3 = 3; // Quelques variables

  // Operateur conditionnel, afficher la plus grande de 2 valeurs
  cout << "La valeur la plus grande de " << v1 << " et "
       << v2 << " = " << (v1 > v2 ? v1 : v2) << endl;

  // Exemples d'operateurs d'incrementation
  cout << "v1++: " << v1++ << endl;
  cout << "v1 tout simplement: " << v1 << endl;
  cout << "--v1: " << --v1 << endl;

  // Exemple d'operateurs d'affectation
  cout << "v2*=v2: " << (v2*=v2) << endl;
  cout << "Resultat de v2<<=3: " << (v2<<=3) << endl;

  // Exemples de l'operateur sizeof
  // On insiste pour les valeurs dans notre environnement!!!
  cout << "sizeof v3: "          << sizeof v3              << endl;
  cout << "sizeof bool: "        << sizeof ( bool )        << endl;
  cout << "sizeof char: "        << sizeof ( char )        << endl;
  cout << "sizeof wchar_t: "     << sizeof ( wchar_t )     << endl;
  cout << "sizeof short: "       << sizeof ( short )       << endl;
  cout << "sizeof int: "         << sizeof ( int )         << endl;
  cout << "sizeof long: "        << sizeof ( long )        << endl;
  cout << "sizeof float: "       << sizeof ( float )       << endl;
  cout << "sizeof double: "      << sizeof ( double )      << endl;
  cout << "sizeof long double: " << sizeof ( long double ) << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Ce programme donne le résultat suivant[1]:

```
La valeur la plus grande de 1 et 2 = 2
v1++: 1
v1 tout simplement: 2
--v1: 1
v2*=v2: 4
Resultat de v2<<=3: 32
sizeof v3: 4
sizeof bool: 1
sizeof char: 1
sizeof wchar_t: 2
sizeof short: 2
sizeof int: 4
sizeof long: 4
sizeof float: 4
sizeof double: 8
sizeof long double: 12
Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *Conclusion*

Voilà, comme annoncé au début de ce chapitre, il existe encore quelques opérateurs que nous étudierons le moment venu dans leur chapitre respectif. Nous allons tout de même maintenant donner un tableau complet de tous les opérateurs, avec leur niveau de priorité allant du plus élevé (en haut) au plus faible (en bas). La dernière colonne du tableau indique, pour des opérateurs de même niveau, s'ils sont évalués de gauche à droite (GD) ou de droite à gauche (DG).

[1] Les valeurs affichées peuvent changer en fonction de l'environnement.

Priorité	*Opérateurs*	*Sens d'évaluation*
Haute	::	GD
	( ) [ ] -> ++ -- (postfix) *cast_spécifiques* .	GD
	! ++ -- - + *(pointeur) & (adresse) ~ (conversion) sizeof new delete	DG
	.* ->*	GD
	* / %	GD
	+ -	GD
	<< >>	GD
	< <= > >=	GD
	== !=	GD
	&	GD
	^	GD
	|	GD
	&&	GD
	||	GD
	?:	DG
	= += -= *= /= %= >>= <<= &= |= ^=	DG
Basse	,	GD

C'est bien compliqué, mais une vue simplifiée et classique des choses fonctionne tout aussi bien dans le 99% des cas. Certaines présentations ajoutent un niveau en isolant le *cast* simple!

Attention: Il faut toutefois rester prudent, car le compilateur peut "jouer" sur des propriétés spécifiques de commutativité, ainsi, si l'on écrit:

```
f ( x ) + g ( y ) // c.f. chapitre des fonctions
```

on ne sait pas laquelle des fonctions sera évaluée en premier. Il est donc important qu'elles ne comportent pas d'effets de bord. Cela fait partie de toute façon des recommandations de base.

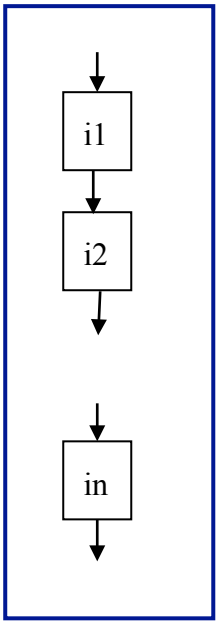
Nous reviendrons par la suite sur cette notion d'effet de bord.

Structures de contrôle

Séquences et blocs

Les structures de contrôle du langage paraissent assez classiques par rapport à d'autres langages, bien que syntaxe et sémantique puissent varier partiellement.

Nous avons déjà vu que les instructions s'enchaînent séquentiellement de manière naturelle:




```
INSTRUCTION_1;
INSTRUCTION_2;
...
INSTRUCTION_N;
```

Nous savons aussi que les instructions peuvent être groupées dans un bloc pour former une instruction composée:

```
{                   // debut du bloc
 INSTRUCTION_1;
 INSTRUCTION_2;
 ...
 INSTRUCTION_N;
}                   // fin du bloc
```

Un tel bloc peut d'ailleurs comporter toutes les déclarations que vous voulez; les objets ainsi déclarés n'existent que dans le bloc en question:

```
{                   // debut du bloc
   DECLARATIONS;
   INSTRUCTIONS;
   DECLARATIONS;
   ...
}                   // fin du bloc
```

Notez aussi, qu'un bloc peut être vide:

```
{ }                 // bloc vide
```

L'instruction la plus simple est l'instruction nulle ou vide, en d'autres termes l'absence d'instruction qui se note tout simplement:

```
;                    //  instruction vide
```

Si le ";" termine une instruction, il ne termine pas un bloc; si l'on en met un après l'accolade fermante, on ajoute une instruction vide!

Par contre, dans nos cas toujours limites, un bloc ne comportant que des instructions vides reste un bloc vide:

```
{ ; }               // bloc vide
```

Une expression devient une instruction, si elle se termine par ";":

```
i + 2;              // C'est une instruction
```

Ceci, même si l'on ne fait rien du résultat, ce qui a peu de sens ici!

Par contre:

```
i++;
```

a un effet!

Les instructions de sélection

Elles permettent de choisir selon des conditions les parties du programme qui seront exécutées ou non!

Comme dans la grande majorité des autres langages, elles sont au nombre de 2:

- Le **if** (Classique)
- Le **switch** (A peu près l'équivalent du case dans d'autres langages)

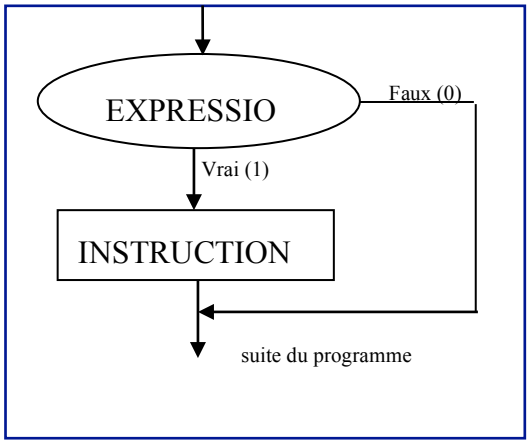
L'instruction if

La forme simple du **if** permet d'effectuer une instruction si une condition est vraie et de ne rien faire sinon:

```
if ( EXPRESSION )
      INSTRUCTION;
```

INSTRUCTION peut être quelconque, celles que nous avons déjà vues et celles que nous verrons par la suite. Donc cela peut aussi être une autre instruction **if**.

Cette possibilité d'imbrication est vraie pour toutes les autres instructions structurées que nous étudierons.



Si la partie *INSTRUCTION* doit comporter plusieurs instructions, il suffit de les grouper dans un bloc entre accolades:

```
if ( EXPRESSION )
{
    INSTRUCTION1;
    INSTRUCTION2;
    ...
}
```

A noter:

- L'absence de **then** après l'expression, contrairement à de nombreux autres langages.
- La présence de parenthèses obligatoires autour de l'expression.

L'expression représente bien une condition booléenne au sens où nous l'avons vu au chapitre qui précède.

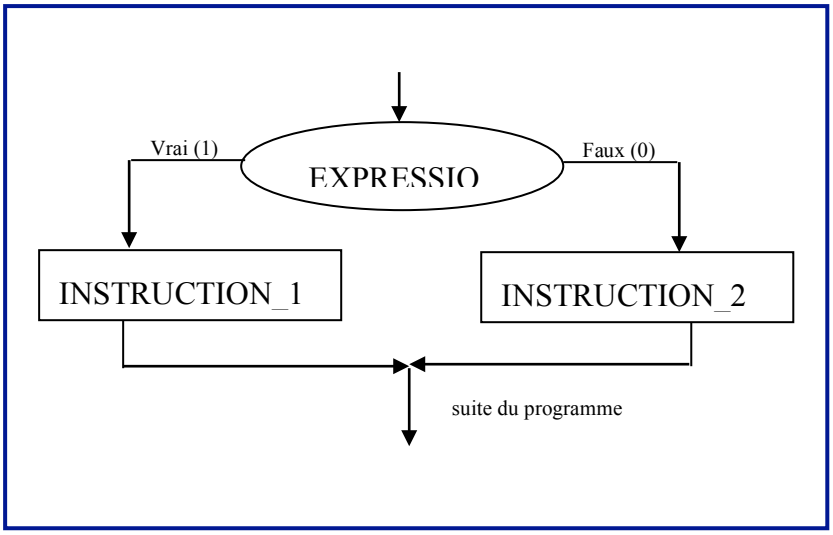
Exemples:

```
if ( a > b )
   i = a + 1;  // 1 seule instruction


if ( a > b && b != 0 )
{
   i += a;          //  Plusieurs instructions, avec
   b--;             //  un formalisme tres C
}
```

Bien entendu, une instruction **if** peut comporter une alternative **else** exécutée si la condition n'est pas satisfaite:

```
if ( EXPRESSION )
   INSTRUCTION_1;
else
   INSTRUCTION_2;
```



Dans le cas d'imbrication:

```
if ( EXPRESSION_1 )
   if ( EXPRESSION_2 )
      INSTRUCTION_1;
   else
      INSTRUCTION_2;
```

Le **else** se rapporte au **if** interne; l'indentation n'y est pour rien, mais elle aide à la compréhension lors de la relecture.

Comme un programme est écrit une fois mais relu de très nombreuses fois...!

Toutefois, si l'on désire que le **else** se rapporte au **if** externe, il faudra obligatoirement utiliser un bloc:

```
if ( EXPRESSION_1 )
{   if ( EXPRESSION_2 )
       INSTRUCTION_1;
}
else
   INSTRUCTION_2;
```

L'instruction se trouvant devant un **else** doit se terminer par un ";". Cependant nous vous rappelons qu'un bloc ne se termine **pas** par un ";". Si nous en ajoutions un avant les **else** nous aurions 2 instructions (le bloc et l'instruction vide), ce qui provoquerait une erreur de syntaxe!

Pour des alternatives elles-mêmes conditionnelles, plutôt que d'accumuler les indentations, il sera préférable d'adopter une présentation du type :

```
if ( EXPRESSION_1 )
   INSTRUCTION_1;
else  if ( EXPRESSION_2 )     /*  Seulement si la condition
                             1 est fausse                */
   INSTRUCTION_2;
else  if .....           /*  Seulement si condition 1
                              et condition 2 sont fausses */
else  /*  Si toutes les autres conditions sont fausses    */
   INSTRUCTION;
```

Exemple

```
/*
   Programme exemple: Utilisation de l'instruction if
   Resolution du probleme de l'equation du deuxieme degre
    INSTRUCTION_IF
*/
#include <cmath>
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  double    a, b, c,   // coefficients a*x**2 + b*x +c = 0
  discriminant,        // pour le calcul du discriminant
  temporaire;          // pour calcul intermediaire
  cout << " Resolution de l'equation du 2eme degre\n\n";
  cout << " Introduisez le coefficient A: ";  cin >> a;
  cout << " Introduisez le coefficient B: ";  cin >> b;
  cout << " Introduisez le coefficient C: ";  cin >> c;

  /* l'equation est-elle degeneree */
  if ( a == 0.0 )
    /* oui, le signaler */
    cout << " Ce n'est pas une equation du 2me degre\n";
  else
  { /* non, calculer le discriminant */
    discriminant = b * b - 4.0 * a * c;

    /* les solutions sont-elles imaginaires? */
    if ( discriminant < 0.0 )
      /* oui: le signaler */
      cout << " Pas de solution reelle\n";

    /* non: y a-t-il 1 seule solution reelle? */
    else if ( discriminant == 0.0 )
      /* oui, calculer et afficher cette solution */
      cout << " Il n'y a qu'une solution, qui est X = "
           << -b / ( 2.0 * a ) << endl;


    else
    {
      /* 2 solutions reelles, les calculer et les afficher */
      temporaire = ( -b - sqrt(discriminant ) ) / ( 2.0 * a );
      cout << " Il y a 2 solutions:  X1 = " << temporaire
           << " X2 = " << -b / a - temporaire << endl;
    }
  }

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Un exemple d'exécution:

```
Resolution de l'equation du 2eme degre

 Introduisez le coefficient A: 2
 Introduisez le coefficient B: 4
 Introduisez le coefficient C: -6
 Il y a 2 solutions:  X1 = -3 X2 = 1
Fin du programme...Appuyez sur une touche pour continuer...
```

L'instruction switch

L'instruction **switch** consiste à permettre de poursuivre le programme à une partie (une branche) spécifique de l'instruction, en fonction de la valeur que prend une expression. Il s'agit donc d'un aiguillage!

Sa syntaxe générale:

```
switch ( EXPRESSION )
{
   case CAS_1 :
      INSTRUCTION(S)_1;
   ...
   case CAS_n :
      INSTRUCTION(S)_n;
   default :
      INSTRUCTION(S);
}
```

L'instruction s'introduit par le mot réservé **switch** suivi d'une expression entre parenthèses. Elle doit obligatoirement être d'un type entier; mais n'oublions pas que le type **char** est assimilable aux entiers et par la suite les énumérés aussi. Ensuite, les différents cas sont

énumérés entre des { } qui représentent le corps de l'aiguillage.

Chaque cas s'introduit lui-même par le mot réservé **case** suivi d'une expression constante, en d'autres termes évaluable à la compilation (le plus souvent il s'agit d'une simple constante), et se terminant par " : ". Ensuite, viennent les instructions à exécuter pour ce cas; il peut y en avoir une ou plusieurs, sans ici devoir mettre des { }.

Plusieurs valeurs de l'expression d'aiguillage peuvent conduire à l'exécution des mêmes instructions; pour cela, il suffit de mettre devant les instructions en question chaque valeur de cas, introduite par le mot réservé **case** et suivi de " : ".

Soit:

```
...
case CAS_1 :
case CAS_2 :
/* les instructions a executer lorsque la valeur de
     l'expression vaut CAS_1 ou CAS_2 */
```

Contrairement à d'autres langages, lorsque l'on arrive à la fin des instructions d'un cas, le programme se poursuit en séquence, c'est-à-dire exécute les instructions du cas suivant. Ceci peut être utile dans certaines situations particulières.

Ainsi:

```
...
case CAS_1 :
  INSTRUCTION_1;
case CAS_2 :
  INSTRUCTION_2;
...
```

Si l'aiguillage nous envoie à CAS_1, nous exécutons INSTRUCTION_1 puis INSTRUCTION_2... et éventuellement encore d'autres.

Pour rompre le passage automatique à la branche suivante, il existe l'instruction **break**. Celle-ci interrompt l'exécution de la structure et le contrôle passe à l'instruction suivant l'accolade fermante de l'instruction **switch** (**break** sera également utilisé dans les boucles).

Ainsi:

```
   ...
   case CAS_1 :
      INSTRUCTION_1;
      break;
   case CAS_2 :
      ...
}
/* suite du programme apres un break */
```

Dans ce cas-là, si l'aiguillage nous envoie dans la branche CAS_1, seule

INSTRUCTION_1 est exécutée, ensuite nous passons aux instructions qui suivent le **switch**.

Finalement, une et une seule branche d'un aiguillage peut être introduite par:

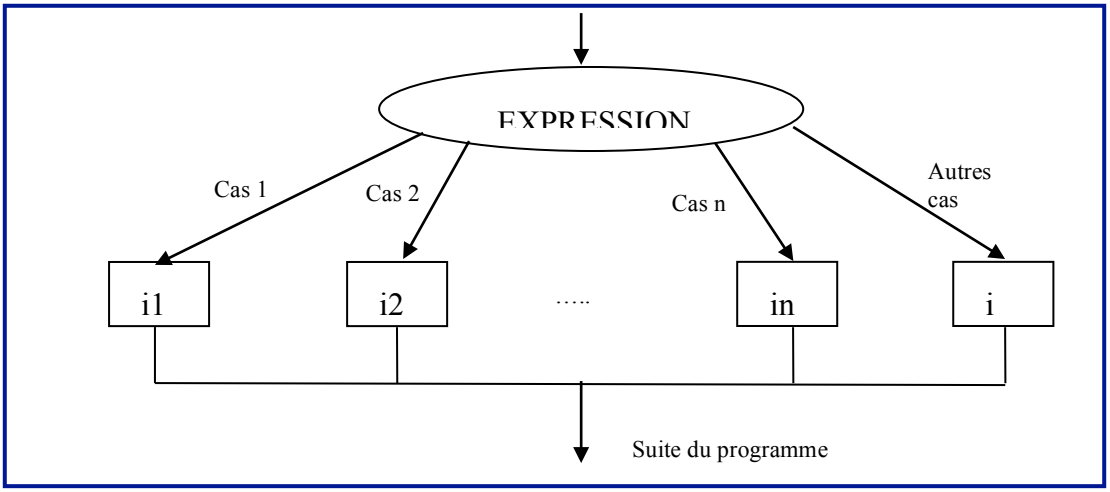
```
    ...
    default :
        INSTRUCTION(S);
}
```

Cette branche s'exécute lorsque la valeur de l'expression diffère de tous les cas prévus explicitement. Bien que cela ne soit pas indispensable, il semble raisonnable pour la lisibilité de mettre cette branche comme dernière de l'aiguillage.

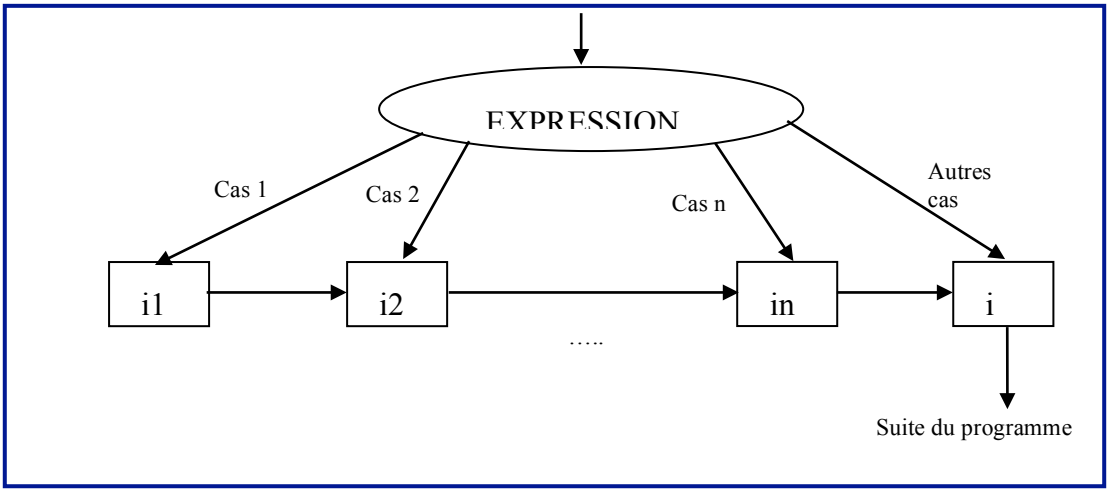
Si la branche **default** n'existe pas et que l'expression du **switch** prend une valeur non prévue dans les différents cas, le programme se poursuit simplement après l'accolade fermante correspondant au début de l'instruction **switch**, tout se passe comme si l'instruction n'existait pas!

Notons encore qu'une valeur de cas ne peut apparaître qu'une seule fois dans la liste.

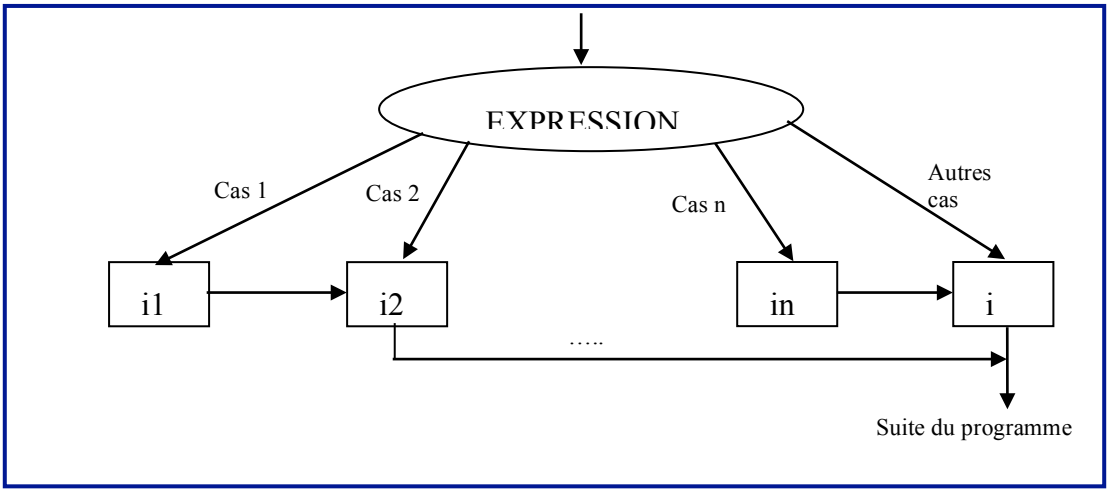
Nous pouvons représenter l'effet de l'instruction **switch** de la manière suivante s'il y a un **break** à la fin de toutes les branches:



Par contre si aucune branche ne se termine par une instruction **break**:



Et si seules certaines branches se terminent par une instruction **break**, par exemple:



Un exemple de programme complet sera donné dans la suite, en liaison avec l'instruction de boucle **do** … **while**.

Les instructions de boucles

Elles permettent sous différentes formes et conditions de répéter certaines parties d'un programme.

Elles existent sous 3 formes:

- **do** ... **while** (Caractéristique: toujours exécutée au moins une fois)
- **while** (Caractéristique: peut ne jamais être exécutée)
- **for** (Caractéristique: à utiliser en principe lorsqu'on peut déterminer en début de boucle le nombre de répétitions)

do while

Sa syntaxe générale a la forme suivante:

```
do
   INSTRUCTION;
while ( EXPRESSION );
```

Et comme toujours s'il y a plusieurs instructions:

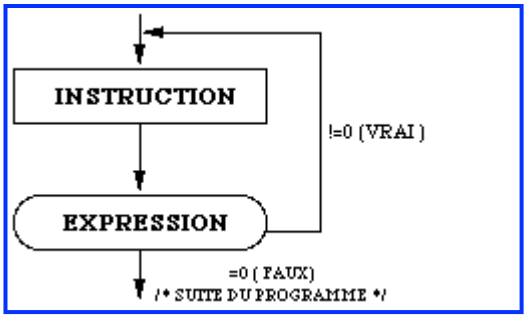
```
do
{
  INSTRUCTION_1;
  ...
  INSTRUCTION_n;
}
while ( EXPRESSION );
```

EXPRESSION est une expression booléenne au sens habituel pour le langage.

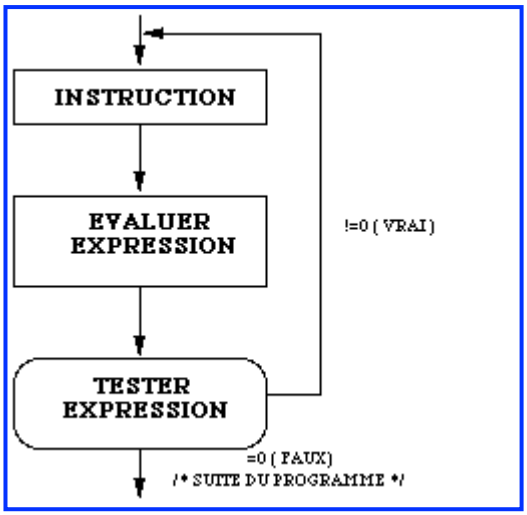
Les instructions de ce type de boucle s'exécutent toujours au moins une fois.

Attention, on reprend l'exécution de la boucle *si la condition est vraie* (EXPRESSION différente de 0).

On peut représenter, de manière classique, l'effet de cette boucle sous la forme:



Mais, étant donné la notion généralisée d'expression, il serait préférable de décomposer cette représentation de la manière suivante:



Nous préférons cette deuxième forme car l'évaluation de *l'expression* peut modifier certaines variables; il y a alors un double effet!

Exemple: nous pourrions très bien écrire quelque chose du genre:

```
do
{ ...
}
while ( ( i += 2 ) <= 10 );
```

Relevons que dans des cas limites, et bien que cela soit peu recommandable pour la lisibilité du programme, le corps de la boucle peut être vide, les opérations se faisant toutes dans la condition:

```
do
{}
while ( cout << i-- << endl, i );
```

Ou encore, ce qui revient au même:

```
do ;
while ( cout << i-- << endl, i );
```

Nous créons très facilement une boucle infinie:

```
do
{ ... }
while ( 1 );
```

Nous verrons plus tard, à la fin de ce chapitre, comment sortir d'une telle boucle!

A noter pour la syntaxe, les parenthèses requises autour de *EXPRESSION*.

> **Attention**: comme dans tous les langages, il est raisonnable que dans les instructions de la boucle ou dans *EXPRESSION* elle-même, on modifie au moins l'une des valeurs qui servent à l'évaluation de *EXPRESSION*.

Le programme exemple qui va suivre illustre aussi l'utilisation de l'instruction **switch.**

Exemple

```
/*
   Programme exemple: Utilisation des instructions switch
    et do while. Preparation d'un menu de selection
     INSTRUCTION_SWITCH
*/
#include <cstdlib >
#include <iostream>
using namespace std;
int main ( )
{
  char reponse; // Pour la reponse de l'utilisateur
  /* Boucle pour traiter toutes les demandes de l'utilisateur */
  do
  {
    /* Afficher le menu, mise en page pour la comprehension */
    cout << "\n\t\t Menu des options:\n"
         << " 1\tOption 1\n"
         << " 2\tOption 2\n"
         << " 3\tOption 3\n"
         << " 4\tOption 4\n"
         << " 5\tOption 5\n"
         << " 6\tOption 6, pour terminer\n"
         << " \n\n\tDonnez votre option: ";
    /* Lire la reponse de l'utilisateur */
    cin >> reponse;

    /* Aiguillage en fonction de la reponse de l'utilisateur */
    switch ( reponse )
    {
      case '1': // cas simple
        cout << "\n Vous avez choisi l'option 1\n";
        break; // sortie de l'aiguillage
      case '2': // cas se poursuivant sur le cas suivant
        cout << "\n Vous avez choisi l'option 2\n"
             << "... qui se poursuit par...\n";
        /* le cas se poursuit avec le cas 3!, car pas de break */
      case '3': // cas simple pouvant etre la suite du cas qui precede
        cout << "\n Vous avez choisi l'option 3\n";
        break;
      case '4': // traitement identique pour 2 cas
      case '5':
        cout << "\n Vous avez choisi l'option 4 ou 5\n";
        break;

     case '6': // cas simple, et condition d'arret
        cout << "\n Vous avez choisi l'option 6 pour terminer\n";
        break;
```

```
    default:
      /* cas final traitant tous les cas non prevus explicitement */
      cout << "\nCette option """ << reponse
           << """ n'est pas valable\n"
           << "\nRECOMMENCEZ\n";
   }

  /*
     condition de sortie de la boucle, cas 6
  */
  } while ( reponse != '6' );

  return EXIT_SUCCESS;
}
```

Ce programme peut donner le résultat suivant:

```
                  Menu des options:
1       Option 1
2       Option 2
3       Option 3
4       Option 4
5       Option 5
6       Option 6, pour terminer


        Donnez votre option: 1

 Vous avez choisi l'option 1

                  Menu des options:
1       Option 1
2       Option 2
3       Option 3
4       Option 4
5       Option 5
6       Option 6, pour terminer

        Donnez votre option: 2
 Vous avez choisi l'option 2
 ... qui se poursuit par...

 Vous avez choisi l'option 3


                  Menu des options:
1       Option 1
2       Option 2
3       Option 3
```

```
4       Option 4
5       Option 5
6       Option 6, pour terminer


        Donnez votre option: 4

Vous avez choisi l'option 4 ou 5

                  Menu des options:
1       Option 1
2       Option 2
3       Option 3
4       Option 4
5       Option 5
6       Option 6, pour terminer


        Donnez votre option: 6

Vous avez choisi l'option 6 pour terminer
```

Note:

Nous n'avons pas prévu en fin de ce programme, comme nous le faisions habituellement, une attente pour éviter la fermeture automatique de la fenêtre. Cette option semble raisonnable ici, l'utilisateur venant de choisir de quitter l'application.

La boucle while

Une boucle très classique et conforme aux autres langages. Sa syntaxe:

```
while ( EXPRESSION )
   INSTRUCTION;
```

Et si l'on a plusieurs instructions à exécuter dans le corps de la boucle:

```
while ( EXPRESSION )
{  …
   INSTRUCTIONS;
   …
}
```

Mis à part le fait que l'expression s'évalue en début de boucle, et donc que les instructions de la boucle peuvent ne pas être exécutées du tout, ce que nous avons dit de la boucle **do** ... **while** s'applique aussi à la boucle **while**. Par conséquent nous pouvons faire les mêmes remarques: suivant la situation toutes les "instructions" peuvent se trouver dans la condition, avec un corps de boucle vide:

```
while ( cout << i-- << endl, i );
```

Ce qui revient au même que:

```
do
{}
while ( cout << i-- << endl, i );
```

De manière générale, la boucle:

```
while ( "condition" ) { }
```

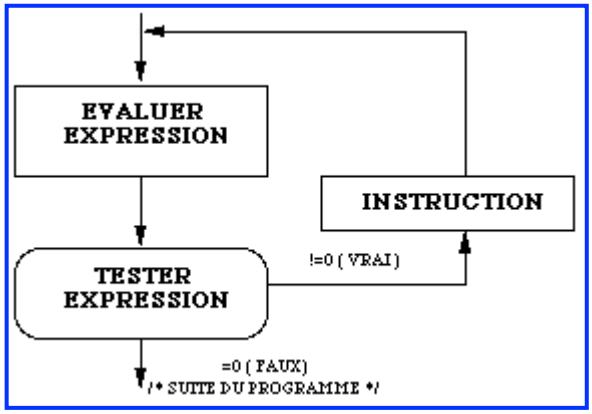
est totalement équivalente à:

```
do { } while ( "condition" );
```

Une boucle infinie se crée facilement:

```
while ( 1 ) { … }
```

Finalement, on peut représenter son exécution par:



Exemple

```
/*
  Programme exemple: Utilisation de l'instruction while
  Calcul du PGCD de 2 nombres
   PGCD
*/
#include <cstdlib >
#include <iostream>
using namespace std;
int main ( )
{
  int val_1, val_2; // Les 2 valeurs donnees par l'utilisateur
  int temporaire;   // Variable temporaire pour l'echange

  /* Demander a l'utilisateur ses 2 valeurs */
  cout << "Calcul du PGCD de 2 valeurs entieres\n\n";
  cout << "Donnez la premiere valeur: ";    cin >> val_1;
  cout << "Donnez la deuxieme valeur: ";    cin >> val_2;

  /* Boucler tant que l'on a pas trouve la valeur */
  while ( val_2 != 0 )      // Pourrait s'ecrire: while (val_2)
  {
    temporaire = val_1 % val_2;
    val_1 = val_2;
    val_2 = temporaire;
  }
```

```
  /* affichage du resultat */
    cout << "Le PGCD de ces 2 valeurs est: " << val_1 << endl;

    cout << "Fin du programme...";
    system ( "pause" );
    return EXIT_SUCCESS;
  }
```

Exemple d'exécution:

```
Calcul du PGCD de 2 valeurs entieres

Donnez la premiere valeur: 132
Donnez la deuxieme valeur: 426
Le PGCD de ces 2 valeurs est: 6
Fin du programme...Appuyez sur une touche pour continuer...
```

La boucle for

Boucle à utiliser en principe dans les situations où l'on veut exprimer quelque chose du genre:

" pour toutes les valeurs comprises entre 2 bornes faire …"

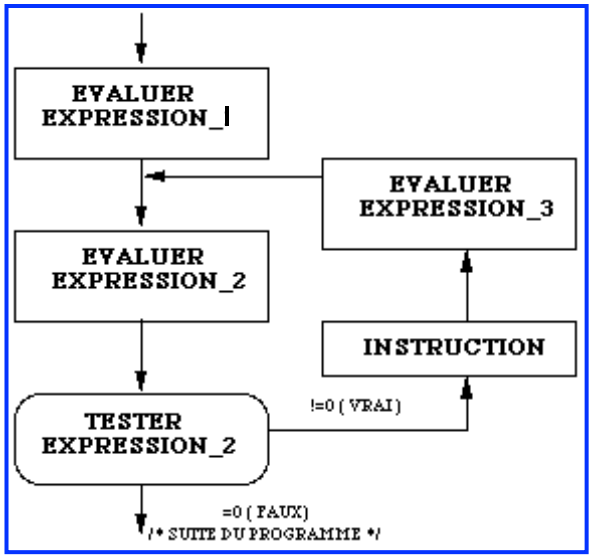
En d'autres termes lorsqu'on peut fixer à l'entrée de la boucle le nombre de fois qu'elle sera effectuée!

Sa syntaxe prend la forme générale suivante:

```
for ( EXPRESSION_1; EXPRESSION_2; EXPRESSION_3)
   INSTRUCTION;
```

Comme toujours INSTRUCTION représente une instruction simple, ou un groupe d'instructions mis entre { }.

Expliquons cette boucle **for**, mais pour bien comprendre ce qui se passe, donnons tout de suite l'organigramme de l'exécution d'une telle instruction:



Cette représentation nous montre qu'une boucle **while** peut remplacer une boucle **for** (pas nécessairement du point de vue de la lisibilité!), si on écrit:

```
EXPRESSION_1;
while ( EXPRESSION_2 )
{
   INSTRUCTION(S);
   EXPRESSION_3;
}
```

Exprimons maintenant en français son effet:

EXPRESSION_1 :

S'évalue une fois pour toute, avant d'entrer dans la boucle; elle sert donc généralement à initialiser la variable de boucle, mais pas nécessairement.

EXPRESSION_2 :

S'évalue en début de boucle lors de chaque passage (premier compris); c'est le résultat de cette évaluation qui détermine si l'on réexécute la boucle (!= 0/**true**) ou pas. Comme pour la boucle **while**, les instructions de la boucle **for** peuvent ne pas du tout être exécutées.

EXPRESSION_3 :

S'évalue en fin de boucle, après l'exécution des instructions de son corps et avant de revenir évaluer et tester *EXPRESSION_2*.

Dans le cadre d'une programmation propre:

- *EXPRESSION_1* devrait servir à initialiser une variable de boucle.
- *EXPRESSION_2* devrait servir de condition de sortie de la boucle
- *EXPRESSION_3* devrait servir pour incrémenter/décrémenter la variable de boucle. Ainsi nous nous approchons d'une boucle **for** classique.

Toutefois, comme nous l'avons constaté, nous pouvons utiliser ces expressions dans d'autres buts; cela ne veut toutefois pas dire qu'il faut le faire!

Exemple classique propre:

```
for ( i = 1; i <= n; i++ )
    ...;
```

Il correspond à "pour tous les *i* de 1 à *n,* par pas de 1, faire …"

La présence des 3 expressions n'est pas impérativement obligatoire. Ainsi, si la variable de contrôle (disons *i*) est initialisée par le programme avant l'entrée de la boucle, nous pouvons alors écrire par exemple:

```
for ( ; i <= n; i++ )
    ...;
```

Si on sort de la boucle d'une autre manière (c.f. suite, instruction **break**), ceci n'étant toutefois que peu recommandable pour ce genre de boucle:

```
for ( i = 1;  ; i++ )
   ...;
```

Cette possibilité existe parce que une instruction vide livre comme "résultat" la valeur **true**, donc une condition toujours vraie et par conséquence une boucle a priori infinie!

De plus, si l'incrémentation se fait par les instructions de la boucle, alors le **for** peut se réduire à:

```
for ( ; i <= n; )
   ...;
```

Ou alors si l'initialisation est nécessaire:

```
for ( i = 3; i <= n; )
   ...;
```

Finalement une boucle infinie peut s'écrire (mais on a déjà vu d'autres possibilités!):

```
for ( ; ; )
   ...;
```

Rappelez-vous qu'une expression peut se composer d'un enchaînement d'expressions (sous expressions) séparées par des virgules. Ainsi, si pour une boucle nous avons besoin d'initialiser plusieurs variables, pas seulement la variable dite de contrôle de boucle, nous pouvons écrire par exemple:

```
for ( i = 1, j = 3; i < n; i++ )
   ...;
```

Ceci peut s'appliquer à chacune des expressions, mais en pratique que pour EXPRESSION_1 et EXPRESSION_3.

N'oubliez pas non plus qu'en C/C++, presque tout est une expression, ce qui nous permet d'écrire des choses très condensées, comme le montrera la deuxième version de notre programme exemple dans la suite:

```
for ( i = 2; i <= limite;
   cout << (serie += 1.0 / i++) << endl );
```

Dans ce cas EXPRESSION_3 est:

```
cout << (serie += 1.0 / i++) << endl
```

L'incrémentation de la variable de contrôle étant réalisée indirectement par l'auto incrémentation de la variable *i* lors de son utilisation dans le calcul de la valeur de l'expression à afficher.

Encore une fois, les expressions sont quelconques. Ne déduisez pas de nos exemples que le pas est toujours de 1; en fait il est *ce que vous voulez*, entier ou fractionnaire, positif, négatif ou même nul puisqu'il peut être absent.

En réalité, il ne s'agit pas réellement d'un pas, mais d'une expression arbitraire quelconque.

Exemples

```
/*
   Programme exemple: Utilisation de l'instruction FOR
   Calcul de la serie 1 + 1/2 + 1/3 + ...
   jusqu'a une limite fixee par l'utilisateur
     SERIE
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
   int   limite,        // Limite de la serie
         i;             // Variable de boucle
   float serie = 1.0;  // Valeur calculee de la serie

  /* Demander a l'utilisateur la limite */
  cout << "Calcul de la serie 1+1/2+...+1/N\n"
       << "Donnez la limite N: ";
  cin >> limite;

  /* Calcul de la serie avec affichage des valeurs intermediaires */
  for ( i = 2; i <= limite; i++ )
  {
    serie += 1.0 / i;
    cout << serie << endl;
  }
  /* Affichage du resultat */
  cout << "Pour N = " << limite << " la serie vaut: " << serie << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Exemple d'utilisation:

```
Calcul de la serie 1+1/2+...+1/N
Donnez la limite N: 7
1.5
1.83333
2.08333
2.28333
2.45
2.59286
Pour N = 7 la serie vaut: 2.59286
Fin du programme...Appuyez sur une touche pour continuer...
```

Nous avons déjà signalé la possibilité de déclarer des variables n'importe où dans le code, tout en déconseillant pour des raisons de lisibilité de l'utiliser, ou plutôt d'en abuser!

Toutefois, la boucle **for** représente un cas particulier puisque nous pouvons, dans la parenthèse définissant les caractéristiques de la boucle, déclarer par exemple ce que nous considérerons comme la variable de boucle. L'avantage d'une telle déclaration réside dans le fait que la variable n'existera que pour les instructions qui constituent le corps de la boucle, elle reste donc purement locale à celle-ci et ne pourra pas s'utiliser en dehors.

Avec cette possibilité, le programme ci-dessus pourrait s'écrire:

```
/*
  Programme exemple: Utilisation de l'instruction FOR
  Calcul de la serie 1 + 1/2 + 1/3 + ...
  jusqu'a une limite fixee par l'utilisateur
  SERIEb
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  int   limite;         // Limite de la serie
  float serie = 1.0;    // Valeur calculee de la serie
  /* Demander a l'utilisateur la limite */
  cout << "Calcul de la serie 1+1/2+...+1/N\n";
  cout << "Donnez la limite N: ";
  cin >> limite;
  /* Calcul de la serie avec affichage des valeurs intermediaires */
  for ( int i = 2; i <= limite; i++ )
  {
    serie += 1.0 / i;
    cout << serie << endl;
  }
  /* Affichage du resultat */
  cout << "Pour N = " << limite << " la serie vaut: " << serie << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Il donne évidemment les mêmes résultats que la première version.

Et encore une autre version de ce programme, donnant toujours les mêmes résultats, mais plus proche des habitudes prises depuis C, par contre très certainement moins lisible et donc a priori peu recommandable:

```
/*
  Programme exemple: Utilisation de l'instruction FOR
  Calcul de la serie 1 + 1/2 + 1/3 + ...
  jusqu'a une limite fixee par l'utilisateur
    SERIE2
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  int   limite;       // Limite de la serie
  float serie = 1.0;  // Valeur calculee de la serie

  /* Demander a l'utilisateur la limite */
  cout << "Calcul de la serie 1+1/2+...+1/N\n";
  cout << "Donnez la limite N: ";
  cin >> limite;

  /* Calcul de la serie avec affichage des valeurs intermediaires */
  for ( int i=2; i <= limite; cout << (serie += 1.0 / i++) << endl );

  /* Affichage du resultat */
  cout << "Pour N = " << limite << " la serie vaut: " << serie << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

❑ *Instruction break*

Nous pouvons, comme nous l'avons vu, facilement construire des boucles infinies. Il faut donc disposer d'un moyen complémentaire pour sortir d'une telle boucle.

Ce moyen, l'instruction:

```
break;
```

Nous l'avons déjà utilisée en conjonction avec l'instruction **switch**. **break** provoque un arrêt brutal de la boucle et une poursuite du programme en séquence à l'instruction qui suit la fin de celle-ci. Bien entendu en pratique **break** sera généralement associé à une condition:

```
... /* a l'interieur de la boucle */
if ( i == 0 )
   break;
```

Attention toutefois à ne pas abuser de cette possibilité, qui peut apparaître dans chacune des 3 formes de boucles.

Pour les boucles imbriquées, **break** ne fait sortir que de la boucle la plus interne. Cette situation peut éventuellement justifier l'utilisation d'un **goto**!

❑ *Instruction continue*

L'instruction **continue** interrompt l'itération courante de la boucle pour recommencer l'itération suivante ou abandonner la boucle, en fonction de l'état actuel de la condition de boucle, soit:

- Dans le cas de la boucle **while**, le contrôle passe en début de boucle pour évaluer la condition et décider de la suite.
- Dans le cas de la boucle **do** ... **while**, le contrôle passe à la fin de la boucle pour évaluer la condition et décider de la suite.
- Dans le cas de la boucle **for**, le contrôle passe d'abord en fin de boucle pour évaluer *EXPRESSION_3*, puis revient en début de boucle pour évaluer et tester *EXPRESSION_2* afin de décider de la suite.

Comme **break**, **continue** s'utilise aussi en pratique en conjonction avec une condition:

```
while ( EXPRESSION )
{
   ... /* debut de la boucle */
   if ( CONDITION )
      continue;
   ... /* suite de la boucle */
}
```

A nouveau n'abusez pas de cette possibilité, encore moins que pour le **break**, car la situation ci-dessus s'écrit tout aussi simplement, et ceci de manière bien plus lisible:

```
while ( EXPRESSION )
{ ...  /* debut de la boucle */
   if ( !CONDITION ) // i.e. condition inverse
   {
      ... /* suite de la boucle */
   }
}
```

En tous les cas, un commentaire explicatif s'impose.

Notez aussi que dans une fonction (c.f. chapitre suivant) un **return** à l'intérieur d'une boucle provoquera l'abandon de la fonction, donc celui de la boucle.

❑ *Instruction goto*

Et oui, cela existe aussi en C/C++! Mais si vous avez pu vous en passer sans problème jusqu'à maintenant, il y a peu de raisons pour que cela change[1], les structures de contrôle du langage sont suffisamment riches. Et rassurez-vous, nous ne vous donnerons pas de programme exemple!

L'instruction **goto** s'écrit:

```
goto ETIQUETTE;
```

L'exécution séquentielle du programme est interrompue, pour se poursuivre à l'instruction portant l'*ETIQUETTE*. Une étiquette est un identificateur suivi de " : " pouvant se placer devant n'importe quelle instruction:

```
ETIQUETTE:
   INSTRUCTION;
```

Fort heureusement, nous ne pouvons pas réaliser un **goto** dans/hors d'un sous-programme, ni par-dessus des déclarations, à moins que ces déclarations soient comprises dans un bloc que l'on "saute" complètement.

Pour le reste, il n'y a pas de restriction; on peut faire un **goto** avant ou après l'instruction courante, et même de l'extérieur vers l'intérieur d'une structure de contrôle, mais alors attention les dégâts!!

Note: bien que nous déconseillions cette possibilité, une étiquette peut porter le même nom qu'un autre objet déclaré dans le même bloc.

[1] Le **goto** peut devenir utile pour sortir de boucles imbriquées.

Sous-programmes: fonctions

Introduction

Il existe de nombreuses différences sur cette notion de sous-programme entre la définition initiale du langage C et les possibilités offertes par C++ dans ce domaine.[1]

Comme nous l'avons déjà laissé entendre, un sous-programme consiste en une fonction, mais pouvant ne livrer aucun résultat. On peut l'appeler sous forme d'une procédure, sans utilisation du résultat même si elle a été prévue pour en livrer un, ou sous la forme fonctionnelle usuelle, avec utilisation du résultat.

Le programme principal lui-même consiste en une fonction, devant obligatoirement s'appeler "main" et pouvant éventuellement posséder des paramètres, cela dépend de l'interface avec le système d'exploitation.

Signalons tout de suite qu'une fonction peut se compiler de manière indépendante (séparée); nous préciserons ceci plus tard.

Selon la norme, et contrairement aux habitudes en général dans d'autres langages, une fonction ne peut pas être interne au programme principal ou à un autre sous-programme (les sous-programmes ne peuvent pas être imbriqués, même si certains compilateurs le permettent):

```
void sub ( ) // sous-programme
{ …
}


int main ( ) // programme principal
{ …
}
```

[1] Dans ses différentes normes, C a aussi considérablement évolué sur ce point.

La définition d'une fonction commence par l'indication du type du résultat de la fonction (pour nous et pour l'instant pas de résultat d'où le mot réservé **void**!), puis vient un identificateur, au sens usuel du terme, (le nom de la fonction) et se poursuit par des parenthèses; même s'il n'y a aucun paramètre ces parenthèses sont obligatoires[1].

Le corps de la fonction se met entre { } comme celui du programme principal. Il peut comme celui-ci comporter des déclarations:

```
void nom ( )      // sous-programme
{
     // Instructions/declarations : corps
}
```

Les déclarations locales ne sont a priori valables que dans le corps du sous-programme. Pour plus d'informations, se référer au chapitre des classes de déclarations.

Les instructions du corps du sous-programme peuvent être quelconques, toutes celles que nous connaissons plus le **return** dont nous reparlerons d'ici peu!

Le fait d'atteindre la fin du sous-programme ("}") provoque un retour automatique à l'appelant.

Pour l'appel, nous utilisons cette fonction comme une procédure (sans résultat, elle n'en livre d'ailleurs pas!). Dans ce but il faut donner le nom de la fonction, avec à nouveau des parenthèses vides requises:

```
nom ( );
```

Plus généralement pour la suite:

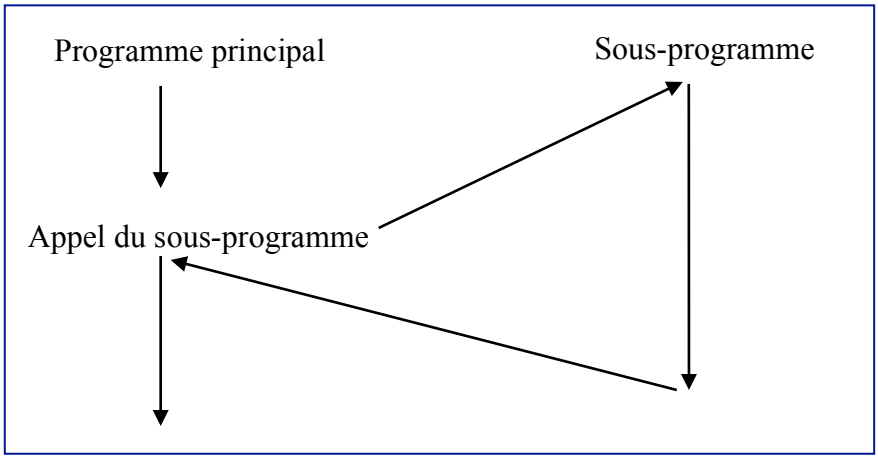
```
nom ( paramètres );
```

[1] Selon la dernière norme C nous devons préciser qu'une fonction ne possède pas de paramètres en mettant le mot réservé **void** entre les parenthèses.

Que se passe-t-il lors de l'appel d'un sous-programme?

- Arrêt de l'exécution des instructions de l'appelant.
- Sauvetage (sur la pile) de l'adresse de retour (l'adresse de l'instruction qui suit l'appel) et d'autres informations suivant le contexte.
- Exécution des instructions du sous-programme (de la fonction).
- Retour à l'appelant, à l'adresse qui avait été sauvée sur la pile et ajustement de la pile.

Nous pouvons symboliser ceci de la manière suivante:



❑ *Le "type" void*

Ce pseudo type indique en fait une absence de typage. On l'utilise ici pour spécifier qu'une fonction ne rend pas de résultat, en d'autres termes: qu'elle s'utilise comme une procédure et/ou éventuellement qu'elle n'a pas de paramètres[1].

[1] Pour les paramètres, ceci est possible, mais absolument pas obligatoire en C++ par contre cela l'était devenu pour C selon la norme C99!

Exemple

L'exemple ci-dessous nous montre une fonction simple, sans paramètre et sans retour d'un résultat; dans d'autres langages nous la qualifierions de procédure.

```
/*
   Programme exemple: Definition et utilisation d'un sous-
           programme (fonction sans parametre)
   FONCTION_SANS_PARAMETRE
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Definition d'un sous-programme */
void sp1 ( )
{ /* Definition d'une variable locale au sous-programme */
  int i = 2;
  /* Debut du sous-programme sp1 */
  cout << "\nDans le sous-programme: " << i << endl << endl;
} // Fin du sous-programme sp1

int main ( )
{
  /* Definition d'une variable locale au programme principal */
  int i = 1;
  /* Debut du programme principal */
  cout << "Avant l'appel du sous-programme: " << i << endl;
  /* Appel du sous-programme */
  sp1 ();
  cout << "Apres le sous-programme: " << i << endl << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Ce programme donne le résultat:

```
Avant l'appel du sous-programme: 1

Dans le sous-programme: 2

Apres le sous-programme: 1

Fin du programme...Appuyez sur une touche pour continuer...
```

Les paramètres

Deux grands principes s'offrent à nous pour passer des paramètres à une fonction:

- Par valeur, ce qui correspond à un paramètre d'entrée uniquement.
- Par variable, ce qui correspond à un paramètre d'entrée/sortie et là nous distinguerons 2 formes possibles.

Nous présenterons d'abord la situation des paramètres par valeur (d'entrée), beaucoup plus simple à comprendre.

Paramètres par valeur

La déclaration d'une fonction avec paramètres revient à compléter sa ligne d'en-tête, en indiquant entre les parenthèses une liste de paramètres. Cette liste consiste en une série d'identificateurs (les paramètres formels) précédés de leur type et séparés les uns des autres par des virgules:

```
void nom ( int p1, char p2 )
{
    /* Corps de la fonction */
}
```

Notons cependant que, contrairement aux déclarations de variables lorsque les paramètres possèdent tous le même type, nous ne pouvons pas donner une liste d'identificateurs et dire qu'ils sont tous d'un type donné; on associe un type spécifique à chaque paramètre!

Lors de l'appel, le passage des paramètres effectifs se fait par **position**, le premier paramètre effectif de l'appel remplace le premier paramètre formel de la définition, et ainsi de suite. Ils doivent correspondre au type de leur paramètre formel associé ou tout au moins à un type jugé compatible.

Avec la déclaration de fonction:

```
void nom ( int p1, char p2 )
```

Nous pouvons écrire l'appel:

```
nom ( 27, 'A' );      // par exemple
```

Le mode de passage par valeur implique une recopie locale du paramètre effectif.

Ceci a pour conséquence que nous pouvons modifier dans le sous-programme la valeur d'un paramètre, sans que cette modification ne provoque de conséquences sur le paramètre effectif.

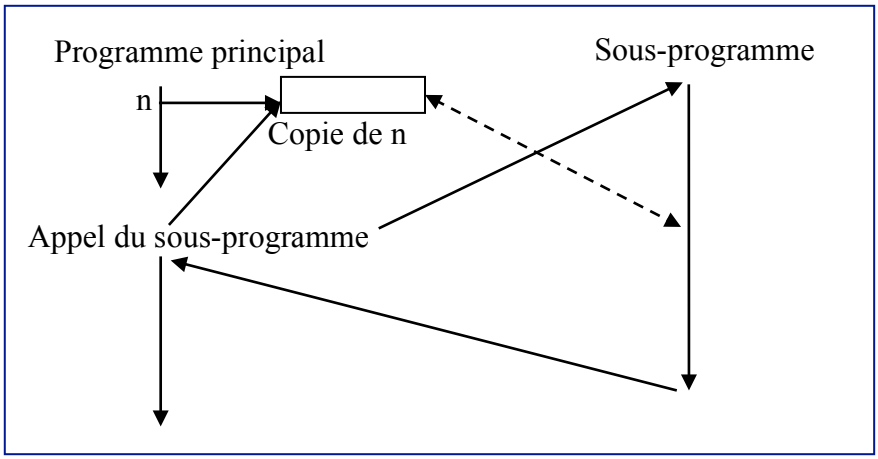
Donc, pour des paramètres d'entrée, la valeur modifiée est utilisable tant que l'on reste dans le sous-programme:

```
void nom ( int p1 )
{
   …
   p1 = p1 + 2;
   … // utilisation de la valeur modifiee de p1
}
```

Autre exemple:

```
void exemple ( int n )
{
     for ( ; n >= 1; cout << n-- << endl );
}
```

Schématisons ce qui se passe dans une telle situation:



Note: Lors de l'appel d'un sous-programme, l'ordre d'évaluation des paramètres n'est pas déterminé.

Ainsi pour l'appel:

```
autreExemple ( f1 ( x ), f2 ( y ) );
```

nous ne pouvons pas dire si f1 ou f2 sera appelée (évaluée) en premier; attention donc aux fonctions avec effets de bord, mais ceux-ci étant de toute façon à proscrire, vous ne devriez pas vous mettre dans une situation délicate.

Un effet de bord consiste par exemple à modifier une variable globale, chose que nous ne savons pas faire pour l'instant, mais qui deviendra malheureusement possible prochainement.

Exemple

```
/*
   Programme exemple: Definition et utilisation d'un sous-programme
      (fonction avec parametres)
      FONCTION_AVEC_PARAMETRES
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Definition d'un sous-programme */
void sp1 ( int i, char c )
{
  /* Debut du sous-programme sp1 */
  cout << "Dans le sous-programme: " << c << endl;
  cout << "Dans le sous-prog., avant incrementation: "
       << i++ << endl;
  cout << "Dans le sous-prog., apres incrementation: "
       << i << endl;
} /* sp1 */

int main ( )
{
  /* Definition de variables locales au programme principal */
  int  i  = 1;
  char ch = 'a';
  /* Debut du programme principal */
  cout << "Avant l'appel du sous-programme: " << i << endl;
  /* Appel du sous-programme */
  sp1 ( i, ch );
  cout << "Apres le sous-programme: " << i << endl << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Ce programme donne comme résultat:

```
Avant l'appel du sous-programme: 1
Dans le sous-programme: a
Dans le sous-prog., avant incrementation: 1
Dans le sous-prog., apres incrementation: 2
Apres le sous-programme: 1

Fin du programme...Appuyez sur une touche pour continuer...
```

Paramètres variables (entrée/sortie)

Comme indiqué en introduction de ce chapitre il existe 2 possibilités pour transmettre des paramètres variables:

- Les références[1]
- Les pointeurs

Par référence

La solution la plus simple consiste à utiliser le mécanisme des références, notion que nous avons introduite au chapitre des types de base.

Il suffit simplement de mettre l'opérateur "&" devant le nom du paramètre formel que l'on désire transmettre par référence. Ensuite l'utilisation du paramètre dans la fonction se fait normalement, comme pour un paramètre par valeur. Toutefois maintenant on accède directement au paramètre effectif, c'est-à-dire que toute modification du paramètre dans la fonction implique en fait celle du paramètre effectif transmis.

C'est la technique que nous vous conseillons d'utiliser a priori.

Voici un exemple classique d'utilisation de cette manière de procéder, une fonction pour l'échange de 2 valeurs:

[1] Cette possibilité n'existe pas en C

```
/*
   Programme exemple: Utilisation de parametres d'entree/sortie
     ECHANGE_REF (05 - sous-pgm param ref)
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Sous-programme realisant l'echange des valeurs
     de ses parametres */
void echange ( int &v1, int &v2 )
{
  int tempo = v1;

  v1 = v2,
  v2 = tempo;
}

int main ( )
{
  int i = 1, j = 2;
  /* Afficher les valeurs avant echange */
  cout << "Avant echange i=" << i <<" et j=" << j << endl;
  echange ( i, j );
  /* Afficher les valeurs apres echange */
  cout << "Apres echange i=" << i <<" et j=" << j << endl << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;

}
```

Ce programme donne comme résultat:

```
Avant echange i=1 et j=2
Apres echange i=2 et j=1

Fin du programme...Appuyez sur une touche pour continuer...
```

Remarques complémentaires:

Contrairement au passage par valeur, il n'y a pas pour les références de conversion implicite du type des paramètres.

Exemple:

```
int f1 ( float p ) ….
int f2 ( float & p ) ….
int f3 ( const float & p ) ….
int i;
```

Pour l'appel:

```
cout << f1 ( i );
```

La valeur du paramètre effectif *i* est convertie en **float** avant l'appel de f1!

Alors que la tentative:

```
cout << f2 ( i );  // ERREUR
```

lève une erreur à la compilation; dans le cas d'une référence, le type doit être strictement le même.

De plus la transmission par référence implique en principe la modification du paramètre effectif et le compilateur travail selon cette démarche, nous ne pouvons donc pas écrire:

```
cout << f2 ( 3.5 );  // ERREUR
```

Par contre l'appel:

```
cout << f3 ( 3.5 );
```

redevient tout à fait correct puisque nous avons signalé au compilateur que le paramètre doit rester constant, donc ne sera pas modifié dans la fonction.

A noter que:

```
cout << f3 ( i );
```

serait correct. La raison: *f3* reçoit une référence temporaire contenant le résultat de la conversion de *i* en **float**.

❑ Par pointeur

Rappelons tout d'abord que tous les paramètres sont passés par recopie locale, mais comme nous allons transmettre une copie de l'adresse de l'objet, par un mécanisme d'indirection nous pourrons modifier l'objet désigné par cette adresse!

Les paramètres d'entrée/sortie font appel à la notion de pointeur qui n'est rien d'autre qu'une adresse ou, plus précisément, qu'un objet dont le contenu est l'adresse d'un autre objet.[1]

Nous ne détaillerons pas pour l'instant le principe des pointeurs; nous nous contenterons de donner le minimum nécessaire pour pouvoir utiliser des paramètres de "sortie".

Exemple:

```
void etrange ( int *valeur )
{
   *valeur = *valeur * 2 + 1;
}
```

L'étoile pour le ou les paramètres formels:

```
int *valeur
```

indique au compilateur que nous ne transmettons pas une valeur mais une adresse donc un pointeur. Toutefois pas n'importe quel pointeur puisque le type indique la nature de l'objet désigné par ce pointeur.

Vous pouvez mélanger dans une même fonction des paramètres transmis selon divers modes!

Lors de l'utilisation du paramètre dans le corps du sous-programme il faut aussi indiquer l'indirection au moyen de l'étoile:

```
*valeur = *valeur * 2 + 1;
```

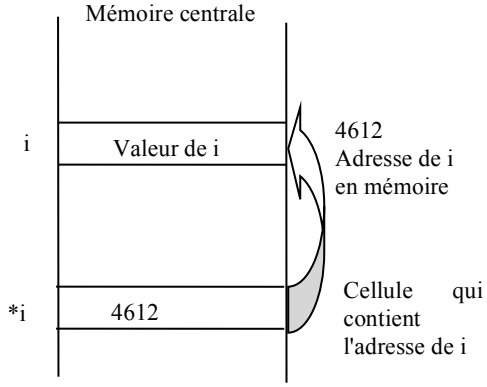
Pour qu'une fonction livre un résultat par ses paramètres, il faut au moment de l'appel, transmettre l'adresse des paramètres effectifs. Il s'agit donc bien de pointeurs qu'il faut traiter comme tels dans la fonction. Pour réaliser cette opération nous utilisons l'opérateur d'adresse noté: "&" mis devant le nom du paramètre effectif.

[1] La notion de référence n'existant pas en C, dans ce langage c'est la seule manière d'obtenir des paramètres d'entrée/sortie!

On peut appeler la fonction de l'exemple ci-dessus, si `i` est une variable déclarée de type **int**, par:

```
etrange ( &i );
```
[1]

Essayons d'illustrer quelque peu cette notion de pointeur; si *i* est une variable déclarée de type **int**, une place lui est réservée dans la mémoire à une certaine adresse. **i* est elle-même une cellule de mémoire qui contient l'adresse de *i*. Pour accéder à *i*, que nous ne connaissons pas de manière directe, nous devons passer par le "guichet" **i*!!!



Voilà, nous développerons d'avantage cette notion de pointeur par la suite!

[1] Contrairement à de nombreux compilateur C, C++ signalera une erreur si nous ne transmettons pas une adresse comme paramètre effectif!

Exemple

```
/*
    Programme exemple: Utilisation de parametres d'entree/sortie
    ECHANGE  (05 - sous-pgm param ptr)
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Sous-programme realisant l'echange des valeurs de ses parametres */
void echange ( int *v1, int *v2 )
{
  int tempo = *v1;

  *v1 = *v2,
  *v2 = tempo;
}

int main ( )
{
  int i = 1, j = 2;
  /* Afficher les valeurs avant echange */
  cout << "Avant echange i=" << i <<" et j=" << j << endl;
  echange ( &i, &j );
  /* Afficher les valeurs apres echange */
  cout << "Apres echange i=" << i <<" et j=" << j << endl << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;

}
```

Ce programme donne le même résultat que la version qui précède.

Complément:

Un paramètre, quelque soit son mode de passage, peut toujours être qualifié par **const** afin de garantir qu'il ne subira pas de modification dans le corps de la fonction; accessoirement cela permet au compilateur, dans certaines circonstances, d'optimiser le code généré.

Exemple:

```
void exemple ( const int *valeur )…
```

Valeur de retour

N'importe où dans une fonction, qu'elle livre un résultat ou non, on peut décider d'abandonner son exécution pour retourner à l'appelant. Il suffit pour cela d'introduire l'instruction:

```
return;
```

Bien entendu en pratique une instruction **return** au milieu du corps du sous-programme sera subordonnée à une condition, sinon nous n'exécuterions jamais les instructions qui suivent ce **return**. Nous pouvons aussi déduire de nos exemples de sous-programmes que la rencontre de l'accolade fermante terminant le sous-programme provoque un **return** implicite. Une telle fonction ne livre pas de résultat en retour, ce qui est normal pour une fonction utilisée comme procédure.

Mais une fonction livre en principe un résultat par son nom, devant lequel lors de la définition de la fonction nous en précisons le type.

Pour livrer effectivement cette valeur de retour, l'instruction **return** se complète par une expression dont le résultat de l'évaluation donnera la valeur à fournir:

```
return EXPRESSION;
```

Exemple:

```
return i + 3;
```

Comme dans le cas d'une instruction d'affectation, et avec les mêmes règles, le type du résultat de l'expression est éventuellement converti dans le type annoncé comme résultat de la fonction.

Rappel: même si votre fonction est prévue pour fournir une valeur de retour, l'appelant n'est pas obligé de tenir compte de cette valeur et il peut continuer à appeler la fonction sous forme de procédure. Toutefois, l'appel normal d'un tel sous-programme consiste en une utilisation fonctionnelle, au sens usuel du terme; l'appelant fait quelque chose du résultat, par exemple l'affiche:

```
cout << ma_fonction ( i );
```

Ou encore l'affecte à une variable:

```
j = ma_fonction ( 3 ) + 5;
```

Attention: Si lors de l'exécution, la dernière instruction traitée de la fonction n'est pas un **return** avec une expression, le résultat livré est quelconque; il y a alors peu de chances que la suite de votre programme corresponde à un comportement raisonnable. Normalement un compilateur C++ l'indiquera par un avertissement!

Exemple:

```
/*
   Programme exemple: Définition et utilisation d'un sous-
     programme. Fonction avec paramètre et retournant un
     résultat par la fonction elle-même
   FONCTION_AVEC_RESULTAT (05 - sous-pgm return)
*/

/* Definition d'un sous-programme */
int sp1 ( int i )
{
  /* Debut du sous-programme sp1 */
  i *= 2;
  /* Resultat de la fonction et fin du sous-programme */
  return i + 1;
} /* sp1 */


#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  /* Variable locale au programme principal */
  int i = 1;
  /* Affichage du resultat de l'appel de la fontion  */
  cout << sp1 ( i ) << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Ce programme livre comme résultat:

```
3
Fin du programme...Appuyez sur une touche pour continuer...
```

Complément:

Une fonction peut livrer une référence ou un pointeur comme résultat; nous ne faisons que signaler cette possibilité pour l'instant, car dans l'état actuel de nos connaissances elle ne nous est guère utile, mais cela le deviendra par la suite. Toutefois relevons tout de suite que ni le pointeur ni la référence ne doivent retourner l'adresse d'une variable locale à la fonction, car cette variable disparaît avec la fin de l'exécution du sous-programme; dans l'unité appelante, nous nous référerions donc à un objet qui n'existe plus!

❑ *Récursivité*

Nous ne l'avons pas encore vu dans nos exemples, une fonction peut appeler une autre fonction, pour autant que l'appelante vienne après dans le texte source ou, et c'est le cas le plus courant, la fonction fait l'objet d'une déclaration de prototype (cf. dans la suite de ce chapitre).

Une fonction peut également s'appeler elle-même; il suffit pour cela de mettre dans son corps, de manière usuelle, un appel à la fonction que nous sommes en train de définir; c'est la forme la plus simple (la plus directe) de la récursivité.

Exemple de fonction récursive, la suite de Fibonacci: par définition de la suite, on pose que les 2 premières valeurs sont égales à 1 et on calcule les valeurs suivantes de proche en proche en additionnant les 2 précédentes, ce qui donne la série suivante:

1, 1, 2, 3, 5, 8, 13, …

La fonction correspondante peut s'écrire:

```
unsigned int fibo ( unsigned int i )
{
   /* Cas limite simple? */
   if ( i <= 1 )
      /* Oui, retourner la valeur 1 */
      return 1;
   else
      /* Non, appels recursifs */
      return fibo ( i - 1 ) + fibo ( i - 2 );
} /* fibo */
```

Nous n'allons que peu développer dans ce cours les notions liées à la récursivité; notons simplement qu'un appel récursif "raisonnable" doit être lié d'une manière ou d'une autre à une condition, sinon le sous-programme n'arrêterait pas de s'appeler (jusqu'à provoquer un débordement de la pile ou une autre erreur).

En d'autres termes, une fonction récursive doit comporter au moins une situation où elle peut livrer un résultat sans appel récursif, dans notre exemple:

```
if ( i <= 1 )
   return 1;
```

Elle peut comporter un ou plusieurs appels récursifs (dans notre exemple 2):

```
else
   return fibo ( i - 1 ) + fibo ( i - 2 );
```

L'exemple simple des nombres de Fibonacci doit nous aider à comprendre les principes de base de la récursivité, cependant il ne s'avère pas idéal comme modèle, car nous obtenons presque aussi facilement une version itérative plus performante et tout aussi compréhensible.

Pour faciliter cette compréhension de la récursivité, illustrons ce qui se passe avec notre fonction *fibo* que nous appellerons *f* ci-dessous pour simplifier les écritures.

Le programme appelle la fonction *f* avec 3 comme paramètre effectif.

- Dans la fonction 3 n'est pas <= 1, donc on exécute l'instruction:

a) **return** f (2) + f (1)

Ceci implique 2 nouveaux appels de la fonction, le deuxième ne se réalisera que lorsque nous aurons obtenu un résultat pour le premier (notez que nous n'avons pas terminé l'appel venant du programme principal).

Evaluons donc le résultat du premier appel, soit *f (2):*

- Dans la fonction 2 n'est pas <= 1, donc on exécute l'instruction:

b) **return** f (1) + f (0)

Ceci implique 2 nouveaux appels de la fonction, le deuxième ne se réalisera que lorsque nous aurons obtenu un résultat pour le premier (notez que nous n'avons toujours pas terminé l'appel venant du programme principal).

Evaluons donc le résultat du premier appel, soit *f (1):*

- Dans la fonction 1 est <= 1, donc on exécute l'instruction: **return** 1.

Nous pouvons donc dans b) remplacer f (1) par 1 ce qui nous donne:

b') **return** 1 + f (0)

Maintenant le programme peut passer à l'évaluation de *f (0)* qui livrera lui aussi 1 comme résultat que nous pouvons remplacer dans b'), ce qui nous donne:

b'') **return** 1 + 1

Nous disposons alors du résultat final de l'évaluation de b) : 2 qui en fait correspondait à l'évaluation de *f (2)* dans a), nous pouvons donc remplacer dans a) pour obtenir:

a') **return** 2 + f (1)

Maintenant nous pouvons enfin passer à l'évaluation de *f (1)* du point a) qui était resté en attente. Cette évaluation donnera bien évidemment à nouveau 1 que nous pouvons remplacer:

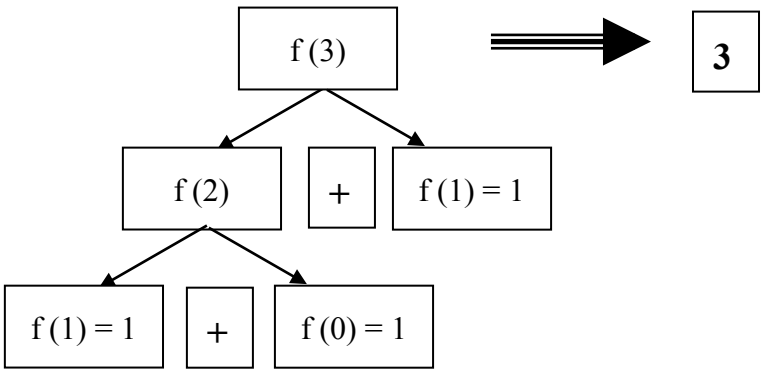
a'') **return** 2 + 1

Ce 3 représente le résultat final de l'évaluation de a) et donc la valeur livrée au programme pour son appel: f (3)

Si nous essayons de formuler ceci de manière un peu plus synthétique nous pouvons écrire:

f (3) = f (2) + f (1)
= f (1) + f (0) + f (1)
= 1 + 1 + 1
= 3

Ou, sous forme d'un arbre représentant les appels de la fonction:



Remarque finale: la récursivité n'apparaît pas toujours de manière aussi directe. Nous pouvons nous trouver dans la situation où une première fonction en appelle une deuxième, qui elle-même rappelle la première, tout ceci bien entendu lié à des conditions.

Nous qualifions cette situation de récursivité croisée. Elle nécessite l'utilisation de prototypes, notion que nous allons introduire, puisque chaque fonction devrait être définie avant l'autre pour pouvoir l'appeler!

Exemple complet

```
/*
   Programme exemple: Definition et utilisation d'une fonction
     recursive: les nombres de Fibonacci
      FONCTION_RECURSIVE (05 - sous-pgm recursivité fibo)
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Fonction calculant le nombre de Fibonacci numero i  */
unsigned int fibo ( unsigned int i )
{
  /* Cas limite simple? */
  if ( i <= 1 )
    /* Oui, retourner la valeur 1 */
    return 1;
  else
    /* Non, appels recursifs */
    return fibo ( i - 1 ) + fibo ( i - 2 );
} /* fibo */

int main ( )
{
  unsigned int valeur; // La valeur donnee par l'utilisateur
  /*   Demande a l'utilisateur de la valeur et
         appel de la fonction recursive
  */
  cout << "Pour quelle valeur voulez-vous calculer: ";
  cin >> valeur;
  cout << "Fibo (" << valeur << ")= " <<  fibo ( valeur ) << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
 return EXIT_SUCCESS;
}
```

Exemple d'exécution:

```
Pour quelle valeur voulez-vous calculer: 4
fibo (4): 5
```

❑ *Prototypes et fonctions externes*

Toute fonction utilisée avant sa définition complète doit faire l'objet d'une déclaration de prototype; ceci est particulièrement utile pour réaliser par exemple de la récursivité croisée! De toute façon il s'agit là d'un principe que nous devrions appliquer systématiquement, d'autant plus qu'il deviendra indispensable pour réaliser de la compilation séparée.

Un prototype consiste en l'en-tête de la fonction; on parle alors de déclaration de la fonction et de sa définition lorsque l'on donne le corps de cette fonction comme nous l'avions fait jusqu'à maintenant. Ceci doit permettre au compilateur de contrôler la validité des appels, c'est-à-dire la correspondance du nombre et du type des paramètres ainsi que du type du résultat livré.

Il prend la forme générale suivante:

```
type_resultat nom ( type nom, …, type nom );
```

Exemples:

```
int sp1  ( char ch );
void sp2 ( char ch, int i );
```

Formellement les noms des paramètres dans le prototype, ici *ch* et *i*, ne sont en pratique pas utilisés; leur présence n'est là que pour des raisons de lisibilité; on peut les enlever, seuls les noms des types sont réellement nécessaires.

Nos exemples de prototypes peuvent donc s'écrire:

```
int sp1 ( char );
void sp2 ( char, int );
```

Programme exemple, d'abord le programme principal:

```
/*
   Programme exemple: Definition et utilisation de fonction
     venant apres dans le fichier source
      PROTOTYPE1  (05 - sous-pgm prototype)
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Declaration d'une fonction prototype */
char sp1 ( char ch );

int main ( )
{
  char ch = 'A';
  /* Appel de la fonction */
  ch = sp1 ( ch );
  cout << "Resultat de la fonction: " << sp1 ( ch ) << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Ensuite le sous-programme:

```
/* Sous-programme ne faisant que d'incrementer le parametre
    Prototype1SP
*/
char sp1 ( char ch )
{
  /* Retourner la valeur */
  return ch + 1;
}
```

Et le résultat de l'exécution:

```
Resultat de la fonction: C
```

Compilation séparée

On appelle généralement *module* un fichier contenant un certain nombre d'éléments compilés séparément. L'unité utilisant des fonctions compilées séparément doit comporter des déclarations de *prototypes* pour ses fonctions externes, afin que le compilateur puisse contrôler la validité du résultat de la fonction et celle de ses paramètres. En pratique, nous le verrons par la suite, ces prototypes seront définis dans des fichiers d'inclusion tels ceux prédéfinis dans l'environnement et que nous avons déjà utilisé (*iostream, cstdlib*).

La forme générale stricte d'un tel prototype:

```
extern type_resultat nom ( type nom, …, type nom );
```

Exemple:

```
extern int sp1 ( char ch );
```

Par défaut, une fonction dont le corps n'est pas défini dans le module courant est considérée comme externe. Il n'est donc pas obligatoire de le dire, ce qui permet entre autre de faire un fichier d'inclusion, utilisable aussi bien par le module de définition que par les autres modules. Notre prototype peut donc aussi s'écrire:

```
int sp1 ( char ch );
```

Notre programme exemple ci-dessus, dans le cas de compilation séparée s'écrit strictement de la même manière, la fonction *main* et le prototype de la fonction *sp1* se trouvent dans un fichier alors que la définition proprement dite de la fonction se situe elle dans un autre fichier. Les modalités pratiques pour réaliser la compilation dépendent elles de l'environnement de travail.

Nombre de paramètres variable

Certains sous-programmes peuvent comporter un nombre variable de paramètres, ceci s'indique par "..." à la fin de la liste de paramètres. La bibliothèque "c*stdarg*"[1] offre des outils sous forme de macros pour traiter ce genre de situation, nous détaillerons ce point par la suite.

Exemple:

```
extern int sp1 ( char ch, ... );
```

A noter que l'on ne donne aucune indication sur le nombre et le type de ces paramètres supplémentaires, donc aucun contrôle ne peut se faire. Ceci dit la fonction doit comporter au moins un paramètre spécifié de manière usuelle.

Sous-programme inline

Signalons aussi que depuis la norme C99 et en C++ la définition d'une fonction peut être précédée du mot réservé **inline**. Ceci indique au compilateur que nous désirons qu'il remplace chaque appel de cette fonction par ses instructions effectives. A partir de 2 appels cette opération augmentera la taille du programme exécutable généré, mais dans tous les cas elle réduira son temps d'exécution.

Exemple:

```
inline int pred ( int ch )
{
  return ch - 1;
} /* pred */
```

Notez toutefois que le compilateur doit toujours accepter cette indication **inline**, mais il n'est pas obligé d'en tenir compte dans le code qu'il génère!!!

Par définition des objectifs d'une fonction **inline**, celle-ci ne peut être que locale à une unité de compilation (un module). Si vous désirez utiliser cette même fonction dans plusieurs unités, il faudra en principe la définir dans un fichier d'inclusion qui sera importé dans toutes les unités en question. Pour le détail des fichiers d'inclusion, se référer au chapitre consacré au préprocesseur.

[1] En C on utilisera pour cela: *stdarg.h*

❑ *Valeurs par défaut*[1]

Les paramètres peuvent comporter une valeur par défaut, c'est-à-dire que lors de l'appel il ne sera pas nécessairement obligatoire de transmettre un paramètre effectif correspondant à ce paramètre formel. Tout se passera comme si vous aviez effectivement transmis la valeur par défaut. Généralement cette possibilité s'utilise avec des paramètres transmis par valeur, toutefois dans des cas limites on peut envisager de l'utiliser pour des pointeurs ou des références.

Pour obtenir ce résultat il suffit de compléter la ligne d'en-tête de la fonction, après le nom du paramètre en question: "=valeur" où valeur correspond au type annoncé pour le paramètre.

Exemple:

```
int bidon ( int p = 5 );
```

On peut appeler cette fonction sans lui transmettre de paramètre:

```
cout << bidon ( );
```

Ceci correspond alors à:

```
cout << bidon ( 5 );
```

La valeur par défaut a été prise automatiquement; toutefois si cette valeur par défaut ne nous convient pas, nous pouvons évidemment en transmettre une autre explicitement:

```
cout << bidon ( 222 );
```

Quelques règles complémentaires:

- Si votre fonction possède plusieurs paramètres, évidemment seuls les derniers peuvent posséder des valeurs par défaut. La situation inverse n'a pas de sens et fait l'objet d'un message d'erreur du compilateur; en effet nous ne pouvons pas appeler la fonction sans transmettre les premiers paramètres, l'appel se faisant par position uniquement.
- Si vous définissez un prototype pour votre fonction, les valeurs par défaut des paramètres se donnent sur le prototype et non lors de la déclaration effective de la fonction. Par contre si la fonction ne fait pas l'objet d'un prototype les valeurs par défaut se donnent lors de la définition de celle-ci.

[1] Cette possibilité n'existe pas en C.

Un petit programme complet pour illustrer ceci:

```
/*
   Programme exemple: Fonction avec parametres par defaut
      PARAMETRES_DEFAUT (05 - sous-pgm param par defaut)
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Livre la somme de ses parametres */
int somme ( int p1, int p2=0, int p3=0 );

int main ( )
{
  int  i1, i2, i3;
  cout << "Donnez i1 premiere valeur entiere: ";
  cin >> i1;
  cout << "somme(i1)= " << somme ( i1 ) << endl;
  cout << "Donnez i2 deuxieme valeur entiere: ";
  cin >> i2;
  cout << "somme(i1,i2)= " << somme ( i1, i2 ) << endl;
  cout << "Donnez i3 troisieme valeur entiere: ";
  cin >> i3;
  cout << "somme(i1,i2,i3)= " << somme ( i1, i2, i3 ) << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
/* Livre la somme de ses parametres */
int somme ( int p1, int p2, int p3 )
{
  return p1 + p2 + p3;
} /* somme */
```

Et un exemple d'exécution:

```
Donnez i1 premiere valeur entiere: 2
somme(i1)= 2
Donnez i2 deuxieme valeur entiere: 4
somme(i1,i2)= 6
Donnez i3 troisieme valeur entiere: 6
somme(i1,i2,i3)= 12

Fin du programme...Appuyez sur une touche pour continuer...
```

La surcharge [1] [2]

Plusieurs fonctions peuvent porter le même nom pour autant que leur signature diffère. La signature d'une fonction correspond aux caractéristiques de ses paramètres, à savoir: leur nombre et le type respectif de chacun d'eux. Le compilateur choisira la fonction à utiliser selon les paramètres effectifs de l'appel par rapport aux paramètres formels des différentes fonctions candidates.

Le résultat d'une fonction ne fait pas partie des éléments permettant au compilateur de différencier 2 surcharges, ni d'ailleurs les éventuelles valeurs par défaut que nous pourrions donner aux paramètres.

Il faut rester très vigilant dans l'utilisation de la surcharge qui peut très facilement nous conduire à des situations ambiguës. Regardez bien les instructions mises en commentaires dans le programme qui suit.

Voici un exemple de programme utilisant la surcharge:

```
/*
   Programme exemple: Surcharge de fonction
      SURCHARGE
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Premiere fonction surchargee
    livre p1+p2
*/
int f ( int p1, int p2 );
/* Deuxieme fonction surchargee
    livre (p1+p2)*10
*/
double f ( double p1, double p2 );
```

[1] Cette possibilité n'existe pas en C.

[2] Très souvent en C++ on utilise le terme de *surdéfinition*, nous préférons garder celui de *surcharge* également utilisé dans d'autres langages.

```
int main ( )
{
  cout << "f(1,3)= " << f ( 1, 3 ) << endl;
  cout << "f(1.0,3.0)= " << f ( 1.0, 3.0 ) << endl;
//
// Attention, les instructions mise en commentaires
//   ci-dessous correspondent à des appels ambigus
//   qui provoquent des erreurs de compilation.
//  cout << "f(1,3.0)= " << f ( 1, 3.0 ) << endl;
//  cout << "f(1.0,3)= " << f ( 1.0, 3 ) << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}


/* Premiere fonction surchargee
    livre p1+p2
*/
int f ( int p1, int p2 )
{
  return p1 + p2;
} /* f */


/* Deuxieme fonction surchargee
    livre (p1+p2)*10
*/
double f ( double p1, double p2 )
{
  return ( p1 + p2 ) * 10.0;
} /* f */
```

L'exécution donne les résultats suivants:

```
f(1,3)= 4
f(1.0,3.0)= 40

Fin du programme...Appuyez sur une touche pour continuer...
```

Les mécanismes qui régissent la surcharge s'avèrent complexes et parfois difficiles à appréhender.

Attention par exemple au fait qu'une déclaration locale masque une déclaration globale de même nom. Ainsi le programme:

```
/*
   Programme exemple: Surcharge de fonction
      SURCHARGE2
*/
#include <cstdlib>
#include <iostream>
using namespace std;

int f ( int p1 );

int main ( )
{
  double f ( double p1 );
  cout << "f(1)= " << f ( 1 ) << endl;
  cout << "f(1.0)= " << f ( 1.0 ) << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}

int f ( int p1 )
{
  return p1 + 1;
} /* f */

double f ( double p1 )
{
  return p1 * 10.0;
} /* f */
```

La fonction *f* ayant un paramètre **double** et livrant un résultat **double** sera toujours appelée et le paramètre **int** promu automatiquement en **double**.

Le compilateur choisit toujours la meilleure fonction possible selon des règles bien établies, que nous ne pouvons pas détailler avec nos connaissances actuelles et que nous vous donnons donc brutalement ci-dessous dans l'ordre décroissant des possibilités de choix:

- Une fonction correspond exactement à la signature de l'appel.
- Une conversion simple est possible: tableau ou fonction transformé en pointeur ou inversement; une variable transformée en constante (mais pas l'inverse!).
- Une promotion automatique est possible: **short** ou **char** vers **int**; **float** vers **double**.
- Une conversion du genre **int** vers **float** est possible.

Nous ne donnerons pas plus de détails pour l'instant sur cet aspect du langage, mais vous aurez largement l'occasion d'y revenir dans le cadre de la programmation objet.

Classes de déclarations et attributs

Nous pouvons maintenant généraliser la forme d'une déclaration:

```
classe type identificateur = valeur_initiale, …;
```

La valeur initiale reste facultative, comme on le sait depuis longtemps déjà; il en va de même pour la classe puisque nous ne l'avons encore jamais utilisée!

Les objets peuvent appartenir à 4 classes différentes: **auto** / **static** / **register** / **extern**. Nous commencerons par présenter ci-dessous le cas des variables automatiques (**auto**) et statiques (**static**).

Toute variable locale à une fonction (toutes celles que nous avons utilisées jusqu'à présent) appartient par défaut à la classe **auto**.

Ainsi la déclaration:

```
int toto = 3;
```

correspond à:

```
auto int toto = 3;
```

Il suffit de commencer une déclaration par un nom de classe pour en fixer l'appartenance des objets ainsi déclarés, ce qui en détermine les caractéristiques.

Si l'on veut un objet de classe **static**, on doit le préciser explicitement puisque la classe par défaut est **auto**:

```
static char tutu;
```

Une variable de la classe **auto** se crée lors de sa déclaration, et disparaît lorsqu'on quitte son bloc de définition. Si vous désirez qu'une variable locale garde sa valeur entre deux appels d'une fonction (ou lors du prochain retour dans le bloc) il faut obligatoirement la déclarer de classe **static**. Lors de l'appel suivant, elle contient toujours la valeur qui était la sienne à la fin de l'appel précédent. Une telle variable a donc une durée de vie plus grande que le bloc où elle est déclarée; en d'autres termes, elle se comporte comme une variable globale sans en posséder les propriétés de visibilité. De plus, une variable **static** est toujours initialisée par défaut à 0 (Attention à la signification suivant le type!).

Si l'on initialise explicitement une variable **static**:

```
static int vendredi = 13;
```

la valeur initiale est donnée une fois pour toute au moment où on lance le programme et non pas à chaque entrée dans le bloc qui contient sa déclaration, alors que pour:

```
int chose = 10;
```

ou ce qui revient au même:

```
auto int chose = 10;
```

la valeur "10" est réaffectée à *chose* à chaque entrée dans le bloc.

❑ *Variables locales et globales*

Toutes les variables que nous avons déclarées jusqu'à présent, qu'elles soient **auto** ou **static**, appartiennent localement au bloc où apparaît leur déclaration. En d'autres termes, elles ne sont pas visibles des autres fonctions ou blocs, sauf les blocs plus internes.

Nous pouvons aussi travailler avec des variables globales; il suffit de les déclarer hors des fonctions. De telles variables demeurent visibles depuis l'endroit de leur définition et dans toutes les fonctions qui suivent à l'intérieur du fichier source. Par définition elles sont de classe ***static***, donc initialisées avec des valeurs nulles.

Soit le squelette de programme suivant:

```
/* exemple de declarations */
long global_1 = 12;       // Variable globale

void sous_prog_1( )  // Premier sous-programme
{
   char local_2;               // Locale a sous_prog_1
   ... /* Corps de sous_prog_1 */
} /* sous_prog_1 */
```

```
void sous_prog_2 ( )       // Deuxieme sous-programme
{ /* Pas de declaration locale */
  ...
} /* sous_prog_2 */

short global_2;       // Autre variable globale

int main ( )
{
  int local_1;        // Locale au Prog. Princ.
  ... /* Corps programme principal */
}
```

Dans cet exemple le programme principal voit et peut utiliser:

global_1, global_2 et local_1, sous_prog_1, sous_prog_2

Le sous-programme sous_prog_1 voit et peut utiliser:

global_1 et local_2, sous_prog_1 (récursivité!)

Et enfin le sous-programme sous-prog_2 voit et peut utiliser:

global_1, sous_prog_1, sous_prog_2 (récursivité!)

On peut initialiser les variables globales comme les variables locales.

Par défaut les variables ainsi que les fonctions globales sont visibles, donc utilisables par d'autres unités compilées séparément, pour autant qu'on les déclare **extern** dans le module utilisateur.

Toutefois, pensez aux dangers de l'utilisation de variables globales, à la difficulté pour rechercher des erreurs dans de tels programmes! Nous vous déconseillons vivement d'en utiliser, sauf bonnes raisons particulières et justifiées.

Exemple

Etudiez les sorties du programme suivant pour bien comprendre entre autre la différence de comportement entre variable **static** et **auto**.

```
/*
   Programme exemple: Definition et utilisation de variables
     auto et static; locales et globales
      CLASSES_DE_DECLARATIONS
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Declaration d'une variable globale valable dans toutes les
     sources qui suivent
*/
char global1 = 'G';

/* Sous-programme illustrant la difference entre variables
    static et auto
*/
void sp1 ( )
{
  /* Variables locales static et auto */
  static int l1 = 1;
  auto   int l2 = 1;   // la meme chose que int l2 = 1;

  /* Debut du sous-programme SP1 */
  cout << "\nDans sp1 variable globale a l'ensemble: "
       << global1 << endl;
  cout << "Dans sp1 variable static: " << l1++ << endl;
  cout << "Dans sp1 variable auto: "   << l2++ << endl;
  return; // Pas indispensable, mais preferable
} // sp1


/* Deuxieme sous-programme pour l'utilisation des
     variables globales
*/
void sp2 ( )
{
  /* Debut du sous-programme SP2 */
  cout << "\nDans sp2 variable globale a l'ensemble: "
       << global1 << endl;
  return; // Pas indispensable, mais preferable
} // sp2
```

```
/* Definition d'une autre variable globale a la suite du source */
char global2 = 'S';

int main ( )
{
  /* Utilisation de la variable globale */
  cout << "Variables globales: " << global1 << "  "
       << global2 <<endl;
  /* Appels des sous-programmes */
  cout << "\nPremier appel des sous-programmes:" << endl;
  sp1 (); sp2 ();
  cout << "\nDeuxieme appel des sous-programmes:" << endl;
  sp1 (); sp2 ();

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Ce programme donne le résultat suivant:

```
Variables globales: G  S

Premier appel des sous-programmes:

Dans sp1 variable globale a l'ensemble: G
Dans sp1 variable static: 1
Dans sp1 variable auto: 1

Dans sp2 variable globale a l'ensemble: G

Deuxieme appel des sous-programmes:

Dans sp1 variable globale a l'ensemble: G
Dans sp1 variable static: 2
Dans sp1 variable auto: 1

Dans sp2 variable globale a l'ensemble: G

Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *Variables externes*

Comme annoncé, les variables globales peuvent se partager entre unités compilées séparément. Dans ce cas, un seul module doit réellement déclarer la variable comme nous l'avons fait jusqu'à maintenant:

```
int truc; // vraie declaration
sous_prog_1 ()
{
   ...
} // sous_prog_1
```

Les autres modules ne feront qu'indiquer qu'ils utilisent une variable qui porte ce nom, qui possède ce type, mais que cette variable vient d'ailleurs, de l'extérieur. Ceci se fait en attribuant aux objets la classe **extern**:

```
extern int truc; // Ici pas de reservation de place
sous_prog_2 ()
{
   ...
} // sous_prog_2
```

Plusieurs autres modules peuvent déclarer cet objet *truc* comme externe, et ainsi se le partager entre eux.

Attention: en C++, contrairement à C, une constante (**const**) demeure toujours locale à son module de définition.

Chose a priori surprenante, une variable globale déclarée explicitement de classe **static** ne restera visible *que* dans le fichier où apparaît sa déclaration et ceci dans toutes les fonctions venant après sa déclaration. Rappelons aussi au passage qu'une variable statique, même globale, sera toujours initialisée avec des octets nuls.

```
static int chose; // visible que dans ce fichier
int main ( )
{
   ...
}
```

Le point:

Sont visibles et donc utilisables dans une fonction:

- Ses paramètres formels
- Ses variables locales
- Les variables globales, locales au module
- Les variables globales exportées par d'autres modules

Il ne faut pas confondre durée de vie et visibilité d'un objet. Ainsi une variable globale **static** a une durée de vie égale à celle du programme, mais une visibilité limitée au module (fichier) ou apparaît sa définition.

❑ *Classe register*

Une variable locale peut se définir de classe **register**:

```
register int machin, truc;
```

On demande ainsi au compilateur, dans la mesure du possible, d'utiliser pour cet objet un registre de la machine, dans le but d'en augmenter l'efficacité d'utilisation, solution intéressante par exemple pour gérer des indices de tableaux fréquemment utilisés.

Toutefois, il faut bien demeurer attentif aux faits que:

- Seule une variable locale peut être **register**.
- Le nombre de registres d'une machine est limité et donc seul un nombre limité de variables peuvent effectivement être conservées dans des registres.
- Il faut que le type de la variable soit physiquement compatible avec un registre (pas possible pour un tableau ou une structure par exemple!).
- Si le compilateur ne peut satisfaire la demande de classe **register** pour un objet, il le transforme sans autre en un objet de classe **auto**.

❑ *Variables locales à un bloc*

En fait tout bloc, pas seulement celui du programme principal ou ceux des fonctions, mais aussi les blocs d'instructions composées, peuvent comporter des déclarations

```
int main ( )
{
   /* Variable locale au programme principal */
   int i;
   ... /* Instructions du programme principal */
  {
      /* Bloc interne avec ses propres declarations */
      int  i;
      char ch = 'A';
      /* Instructions du bloc interne */
      ...
   } // Fin du bloc interne
} // Fin du programme principal
```

La visibilité de tels objets est celle du bloc de déclaration, leur durée de vie dépend de la classe de l'objet. Un identificateur identique à celui d'un bloc de niveau supérieur masque la visibilité de l'objet global portant le même nom.

Une variable locale à un bloc interne peut appartenir à la classe **extern**:

```
{
   extern int i,
   ...
}
```

Dans ce cas, il s'agit bien d'un objet global défini ailleurs; sa durée de vie reste celle des variables globales, mais sa visibilité est restreinte au bloc et non à l'ensemble de l'unité de compilation.

Les 2 attributs suivants ne font pas réellement partie des classes de déclaration, mais étant donné leur nature nous les présentons brièvement dans ce chapitre.

Attribut const

Nous le savons, il s'agit de définir un objet dont la valeur ne peut théoriquement pas être modifiée par la suite. Il faudra l'initialiser lors de sa déclaration, exemple:

```
const int limite = 100;
```

Une autre utilisation fréquente de cet attribut consiste à indiquer qu'un paramètre de sous-programme ne doit pas être modifié, exemple:

```
void affiche ( const char * ch );
```

Attribut volatile

Un objet déclaré *volatile* représente un élément dont la valeur peut être modifiée par le monde extérieur; soit par exemple un port d'entrées/sorties, une horloge, etc.

Cette directive indique au compilateur qu'il ne doit pas faire d'optimisation dans le contexte où de tels objets sont utilisés puisque leur contenu peut se modifier indépendamment du code que l'on réalise. Si le programme lui-même ne doit pas modifier l'objet, les attributs **const** et **volatile** peuvent se combiner.

Ainsi une déclaration du genre:

```
extern const volatile int realTimeClock;
```

est tout à fait possible.

Remarque: l'ordre des mots clés (**extern const volatile**) n'a pas d'importance.

En conclusion

- Dans un module une fonction est par défaut mise à disposition des autres modules. Si l'on désire qu'elle ne soit visible que dans le module de définition, il faut explicitement la déclarer de la classe **static**:

```
static int sp ( );
```

avec **extern**, c'est la seule classe que l'on peut préciser pour un sous-programme!

- Un problème se pose lorsqu'une unité C++ veut appeler une fonction C. Le compilateur C++ génère pour les sous-programmes des noms dits "long": en plus du nom effectif de la fonction, on y trouve des indications sur les paramètres, ceci pour que l'éditeur de liens puisse résoudre proprement les problèmes de surcharge, chose que le compilateur C ne fait pas. Si nous désirons appeler une fonction C nous devons l'indiquer explicitement au compilateur par une forme particulière de la spécification **extern**, exemple:

```
extern "C" int bidon ( int * );
```

Après le mot réservé nous devons ajouter: "C". Si nous avons de nombreuses fonctions possédant cette particularité, nous pouvons les regrouper dans un bloc particulier en indiquant que tous les éléments du bloc ont cette particularité:

```
extern "C"
{
   int bidon ( int * );
   void bidon2 ( float );
   ...
}
```

- Pour leur initialisation, les variables de classes **static** n'acceptent que des expressions constantes[1] (indirectement il en va de même pour la classe **extern**), alors que les variables des classes **auto** ou **register** admettent n'importe quelle expression légale, mais toujours avec l'affectation simple (pas d'opération composite d'affectation). Il faut bien comprendre les conséquences de ceci; pour les variables **static**, la valeur est générée à la compilation; les variables **auto** et **register** nécessitent elles un code d'initialisation impliquant un programme plus gros en taille et plus lent à l'exécution. A noter que cette dernière remarque ne doit pas devenir le critère principal de choix entre les 2 classes.

[1] Attention toutefois, en C++ vous pouvez utiliser une constante "**const**" alors que ce n'est pas le cas en C!

- Les valeurs initiales des variables de classe **static** ne sont affectées que une et une seule fois au lancement du programme, alors que les valeurs initiales des variables **auto** sont recalculées et réaffectées à chaque fois que l'on entre à nouveau dans le bloc où elles sont déclarées. Rappelons que ces variables de classe **auto** disparaissent à la fin de leur bloc de définition et recréées si par la suite on entre à nouveau dans le bloc en question!

Le préprocesseur

Introduction

Nous en avons déjà un peu parlé au début de ce cours et nous allons brièvement compléter nos connaissances sur le sujet. Le préprocesseur est un programme qui transforme notre fichier source avant que le compilateur proprement dit n'intervienne.

Son rôle se borne à des manipulations de chaînes de caractères.

Les lignes du programme source qui représentent des directives pour le préprocesseur commencent toutes par un "#" comme premier caractère significatif; ensuite vient la directive elle-même!

Si en C on fait une utilisation importante des possibilités qu'il offre, en C++ on a tendance à minimiser son utilisation.

#define

Nous avons déjà introduit:

```
#define SYMBOLE texte de substitution
```

qui permet d'associer la chaîne de caractères *texte de substitution* au *SYMBOLE* (qui est un identificateur). En d'autres termes, partout, dans la suite du programme source, où apparaît *SYMBOLE* en tant qu'unité lexicale, il est remplacé (substitué) par le *texte de substitution*.

Exemples:

```
#define begin {
#define end }
```

La chaîne de substitution peut être vide:

```
#define then
```

Notez qu'ici, pour des raisons évidentes, on ne respecte pas la convention qui veut que les constantes soient écrites en majuscules!

Attention, il s'agit réellement d'une simple substitution de chaîne de caractères; ainsi si nous avons défini:

```
#define SOMME1 a + b
```

et

```
#define SOMME2 (a + b)
```

Si dans le programme nous écrivons:

```
i = SOMME1 * 2;      // i = a + b * 2
```

ce n'est pas la même chose que:

```
i = SOMME2 * 2;    // i = (a + b) * 2
```

C'est l'une des raisons (mais pas la seule) qui fait qu'on a tendance à mettre de nombreuses parenthèses apparemment inutiles. Ainsi les définitions ci-dessus s'écriraient plutôt:

```
#define SOMME ((a) + (b))
```

Dans l'exemple qui précède, toutes les parenthèses peuvent devenir utiles si a et b représentent eux-mêmes des symboles correspondants à des expressions.

La chaîne de substitution peut se prolonger sur plus d'une ligne; dans ce cas il faut terminer chaque ligne non finale par un "\" :

```
#define CHOIX cout << "\n Votre reponse: "; \
        cin >> reponse;
```

Remarques complémentaires

Le préprocesseur ne faisant que traiter des chaînes de caractères avant compilation, il est possible, mais très vivement déconseillé, de redéfinir des mots clés:

```
#define while loop
```

Le préprocesseur travaille par unité lexicale (token), il est donc possible mais toujours aussi peu recommandable d'écrire:

```
#define a=b
```

ce qui revient au même que d'écrire:

```
#define a  =b
```

Attention donc aux risques d'erreurs. Toutefois si votre compilateur se conforme à la norme ISO il devrait afficher un message d'avertissement!

Le préprocesseur n'interprète pas les [,] , { , } comme des opérateurs, ce qui peut amener des confusions.

Si nous définissons la macro:

```
#define ADD( a, b) ((a) + (b))
```

L'appel de cette macro avec un élément de tableau à 2 dimensions comme paramètre:

```
x = ADD ( tab [ 1, 1 ], x );
```

générera une erreur car il présente pour le préprocesseur 3 paramètres (qui n'ont d'ailleurs pas de sens!): `tab [ 1` `1 ]` `x`

Avec quelques définitions (on pourrait en ajouter d'autres!), on donne à des programmes C/C++ une apparence (un début tout au moins) un peu plus proche de celle des programmes Pascal ou Ada.

Reprenons l'exemple de l'équation du deuxième degré et modifions-le dans ce sens (cela ne veut pas dire qu'il faut travailler ainsi en pratique!).

Mettons un certain nombre de définitions dans un fichier externe, inclus dans le programme, voici ce fichier:

```
/*
   Fichier de definitions locales
*/
#define then
#define begin {
#define end }
```

Et maintenant le programme qui l'utilise:

```
/*
Programme exemple: Utilisation de l'instruction if
  Resolution du probleme de l'equation du deuxieme degre
    STYLE_PASCAL
*/
#include <cmath>
#include <cstdlib>
#include <iostream>
using namespace std;
#include "params.h"
int main ( )
begin
  double a, b, c, // coefficients a*x**2 + b*x +c = 0
  discriminant,   // pour le calcul du discriminant
  temporaire;     // pour calcul intermediaire
  cout << " Resolution de l'equation du 2me degre" << endl;
  cout << "Introduisez le coefficient A: "; cin >> a;
  cout << "Introduisez le coefficient B: "; cin >> b;
  cout << "Introduisez le coefficient C: "; cin >> c;
  /* l'equation est-elle degeneree? */
  if ( a == 0.0 ) then
    /* oui, le signaler */
    cout << " Ce n'est pas une equation du 2me degre" << endl;
  else
  begin /* non, calculer le discriminant */
    discriminant = b * b - 4.0 * a * c;
    /* les solutions sont-elles imaginaires? */
    if ( discriminant < 0.0 ) then
      /* oui, le signaler */
      cout << " Pas de solution reelle" << endl;
    /* non, y a-t-il 1 seule solution reelle? */
    else if ( discriminant == 0.0 ) then
```

```
    begin
      /* oui, calculer et afficher cette solution */
      cout <<" Il n'y a q'une solution, qui est X = "
           << -b / ( 2.0 * a ) ;
    end
    else
    begin /* Il y a 2 solutions reelles, calculer et afficher */
      temporaire = ( -b - sqrt ( discriminant ) ) / ( 2.0 * a );
      cout << " Il y a 2 solutions:" << endl;
      cout << " X1 = " << temporaire << " et X2 = "
           << -b / a - temporaire ;
    end
  end
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
end
```

❑ #undef

On peut annuler la définition d'un symbole, soit parce que l'on ne désire plus de substitution dans la suite du programme, soit pour lui donner une autre définition.

Ceci se fait par la directive:

```
#undef SYMBOLE
```

Exemple:

```
/* Zone du programme ou TOTO est indefini */
/* Donc pas de substitution               */
...

#define TOTO chaine 1
/* Zone du programme ou TOTO est remplace par chaine 1 */
...
#undef TOTO
/* Zone du programme,ou TOTO est indefini */
/* Donc pas de substitution               */
...
#define TOTO chaine 2
/* Zone du programme ou TOTO est remplace par chaine 2 */
...
/* etc... */
```

Certains compilateurs admettent que l'on redéfinisse un symbole sans passer par un *undef*, mais ce n'est pas la règle et pas la norme! Toutefois un compilateur qui l'accepte devrait le signaler par un avertissement (warning).

> ***Attention:*** La substitution ne se fait pas dans une constante chaîne de caractères ni dans une constante caractère**.**

Exemple: `cout << "J'affiche TOTO\n";`

Affichera toujours: `J'affiche TOTO`

Même si le symbole TOTO a été défini par une directive *#define*.

❑ *Les macro-instructions*

Il s'agit en fait d'un *#define* avec une possibilité de paramètres.

Sa forme générale:

```
#define SYMBOLE( liste_de_parametres ) texte
```

A noter:

- Il ne doit pas y avoir d'espace entre *SYMBOLE* et la parenthèse ouvrante de la *liste_de_paramètres*, car sinon l'on retombe dans une simple définition de symbole.
- La *liste_de_paramètres* formels est constituée de un ou plusieurs identificateurs. S'il y en a plusieurs, on les sépare les uns des autres par des virgules.
- Les paramètres peuvent être utilisés une ou plusieurs fois dans le texte de substitution. Ces paramètres formels seront remplacés lors de "l'appel" par les paramètres effectifs. Nous avons volontairement mis appel entre guillemets car en fait il ne s'agit pas d'un appel, mais d'une simple substitution de texte par le préprocesseur.

Exemple:

```
#define MAX( X, Y ) ( (X) > (Y) ? (X) : (Y) )
```

Notez bien le parenthèsage complet, pas strictement indispensable, mais **très vivement conseillé**, ce qui évite dans certains cas des problèmes!

Nous pouvons utiliser cette définition par exemple de la manière suivante:

```
cout << MAX ( 12, 9 ) << endl;
```

Qui bien entendu affichera 12 puisqu'il correspond à:

```
cout << ( (12) > (9) ? (12) : (9) ) << endl;
```

Encore une fois, rappelez-vous qu'il s'agit de substitution de texte et non pas de réels appels de fonctions. Ce qui veut dire entre autre, que si l'on gagne en vitesse d'exécution, on le paye, sauf dans quelques cas particuliers, en taille du programme! Et attention aussi aux effets secondaires; car si nous écrivons:

```
MAX ( ++i, --k )
```

C'est *I+2* ou *K-2* qui sera livré comme résultat, suivant les valeurs respectives de *i* et de *k*.

Note: Dans la majorité des situations une fonction **inline** remplace avantageusement une macro.

Complément sur les paramètres des macros

- Deux paramètres peuvent être concaténés par ##

Exemple :

```
#define IDENTIFICATEUR( I, J ) I##J
```

L'appel:

```
IDENTIFICATEUR ( toto, 1 )
```

sera remplacé par *toto1*. Dans ce contexte, cela n'a pas beaucoup de sens, mais de manière générale, cela permet de construire par exemple des identificateurs sur la base de 2 (ou plus) paramètres.

- #paramètre devant une chaîne convertit le paramètre en une chaîne, ainsi:

```
#define ECRIT( V ) cout << #V" = " << V << endl;
```

L'appel:

```
ECRIT ( resultat )
```

est équivalent à:

```
cout << "resultat = " << resultat << endl;
```

❑ *Macros prédéfinies*

Il existe des macros prédéfinies, à savoir:

_ _LINE_ _

Livre le numéro de la ligne du fichier source où elle se trouve.

_ _FILE_ _

Livre sous forme d'une chaîne de caractères le nom du fichier source contenant l'appel de la macro.

_ _DATE_ _

Livre sous forme d'une chaîne de caractères la date à laquelle le fichier a été compilé. Sa forme: Mmm jj aaaa (*Mmm*: le nom anglais du mois, *jj*: le numéro du jour et *aaaa* l'année).

_ _TIME_ _

Livre sous forme d'une chaîne de caractères l'heure à laquelle le fichier a été compilé. Sa forme: hh:mm:ss.

_ _ STDC_ _

Livre le degré de conformité de votre compilateur par rapport à la norme, ou tout au moins, le degré que lui estime!

Une valeur de 1 est censée indiquer une conformité totale.

❑ *Directives prédéfinies*

Il existe aussi les directives:

`#line`

Qui permet de forcer le numéro de ligne et le nom du fichier; elle influencera donc l'effet des macros _ _LINE_ _ et _ _FILE_ _

Sa forme générale:

```
#line NUMERO "NOM"
```

avec:

NUMERO: une valeur entière statique positive représentant le numéro que l'on désire attribuer à la prochaine ligne de notre programme source.

"NOM": une chaîne de caractères optionnelle, sensée représenter le nom du fichier qui contient notre programme source.

`#error  texte a afficher`

qui a pour effet d'afficher lors de la compilation le reste du contenu de la ligne (ce qui vient après le #error!). Ceci peut être utile lors de la détection d'un problème. Généralement liée à une condition de compilation du genre: [1]

```
#ifndef NUMERO
   #error Le symbole NUMERO doit etre defini
#endif
```

`#pragma`

Permet d'activer un pragma, c'est-à-dire une directive de compilation. La forme du pragma dépend du compilateur, mais on peut imaginer quelque chose du genre:

```
#pragma nooptimize
```

pour demander au compilateur qu'il ne cherche pas à optimiser le code généré!

Voici un tout petit programme illustrant très brièvement une partie de ces possibilités:

[1] Possibilité traitée dans la suite de ce chapitre sous la rubrique "Compilation conditionnelle"!

```
/*
   Programme exemple: Utilisation des macros predefinies
   MACROSPREDEFINIES
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  cout << "Cette instruction se trouve a la ligne: "
       << __LINE__<< endl;
  cout << "Le fichier s'appelle: " << __FILE__ << endl;
  cout << "Ce fichier a ete compile le: " << __DATE__
       << " a  " << __TIME__ << endl;
  cout << "\nLe compilateur annonce une conformite a la norme de: "
       << __STDC__ << endl;
  #line 24 "Faux_Nom"
  cout << "Cette instruction se trouve a la ligne: "
       << __LINE__ << endl;
  cout << "Le fichier s'appelle: " << __FILE__ << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Résultat d'une exécution:

```
Cette instruction se trouve a la ligne: 10
Le fichier s'appelle: E:\JMT\Polycop\MacrosPredefinies.cpp
Ce fichier a ete compile le: Aug 10 2007 a  10:55:21

Le compilateur annonce une conformite a la norme de: 1
Cette instruction se trouve a la ligne: 24
Le fichier s'appelle: Faux_Nom

Fin du programme...Appuyez sur une touche pour continuer...
```

Inclusion de fichiers

Tous nos programmes commencent en général par:

```
#include <iostream>
```

Qui a pour effet d'inclure le contenu du fichier *iostream* à l'endroit où cette directive est utilisée dans notre programme source.

En fait, nous avons le droit d'inclure à n'importe quel endroit du programme, le contenu de n'importe quel fichier externe, pour autant que l'ensemble résultant respecte la syntaxe du langage.

En pratique, la directive d'inclusion peut prendre 2 formes:

```
#include <nom_du_fichier>
```

ou

```
#include "nom_du_fichier"
```

La différence entre les deux dépend essentiellement des outils de développement. La forme (1) implique une recherche dans des répertoires spécifiques fixés par l'environnement. La forme (2) implique d'abord une recherche dans le répertoire courant et ensuite seulement dans ceux de l'environnement.

Par habitude, mais ceci sans obligation:

- L'extension normalement admise pour de tels fichiers est: "**.h**". Les fichiers de la bibliothèque standard C portent cette extension, mais ceux de la bibliothèque C++ n'en ont pas.
- Ces fichiers sont généralement utilisés pour définir des symboles et des prototypes de fonctions.

Les inclusions peuvent s'imbriquer les unes dans les autres, donc un fichier inclus peut lui-même comporter une ou plusieurs lignes *#include*.

Comme nous l'avons déjà vu en pratique, l'usage veut que tous les éléments qu'un module met à disposition des autres modules soient regroupés dans un fichier portant le même nom que le module, mais avec l'extension "**.h**".

Remarques complémentaires

- Une implémentation peut limiter le nombre d'imbrications des fichiers inclus, toutefois cette limitation ne doit pas être inférieure à 8.
- En règle générale (il y a des exceptions!) on ne devrait pas inclure 2 fois le même fichier dans la même unité de compilation. Une telle situation peut poser de graves problèmes. Elle se produit si dans votre unité vous incluez 2 fichiers qui eux-mêmes font chacun un *#include* d'un même fichier. Une simple précaution du côté des fichiers inclus prévient facilement ce genre de situation. Il suffit d'entourer le contenu du fichier inclus par:

```
#ifndef FICHIER  [1]
#define FICHIER
…/* Le contenu de votre fichier à inclure */
#endif
```

Il faut choisir le symbole *FICHIER* de telle sorte qu'il ne se retrouve en principe pas défini ailleurs dans le contexte d'utilisation!

[1] Pour les règles de compilation conditionnelle, reportez-vous au paragraphe suivant!

❑ *Compilation conditionnelle*

Il s'agit là certainement d'un héritage des langages d'assemblage!

Cette possibilité nous permet, par des conditions évaluables à la compilation, de prendre ou de ne pas prendre en considération pour cette compilation des parties de codes présentes dans le fichier source.

La forme générale d'une zone de compilation conditionnelle est la suivante:

```
#if condition_1
  // Zone compilee si condition_1 vraie
#elif condition_2
  // Zone compilee si condition_1 fausse et condition_2 vraie
  ...
#else
  // Zone compilee si toutes les conditions prealables fausses
  ...
#endif
```

Les parties *#elif* (il peut y en avoir autant que vous voulez!) et *#else* (une seule en dernière position!) sont optionnelles, ce qui veut dire que la forme la plus simple d'une compilation conditionnelle est:

```
#if condition
  ...;
#endif
```

Les conditions peuvent prendre la forme usuelle d'une relation, mais évidemment avec des expressions évaluables à la compilation, par exemple:

```
#if SYMBOLE == 2
   ...;
#endif
```

(Avec *SYMBOLE* défini par: #define SYMBOLE …)

On peut également utiliser la directive *defined* pour savoir si un symbole est défini:

```
#if defined TOTO
#define    TOTO2 BIDON
#endif
```

qui peut aussi s'écrire:

```
#if defined ( TOTO )
#define    TOTO2 BIDON
#endif
```

On réalise des conditions composées, en utilisant les opérateurs booléens:

```
! , && , ||
```

Exemple:

```
#if !defined TOTO && TRUC > 3
#define     TOTO TRUC
#endif
```

De manière générale la compilation conditionnelle s'utilise pour:

- Inclure dans une phase de développement des instructions de trace, mais dans ce cas, il vaudrait mieux disposer d'un bon debugger.
- Générer du code différent suivant le type de machine sur laquelle on travaille.
- Générer des messages distincts en fonction de la langue des utilisateurs:

```
#if FRANCAIS == 1
#define    MESSAGE_1 OUI
#else
#define    MESSAGE_1 YES
#endif
```

En pratique on aura certainement plusieurs messages dans chaque langue, on utilisera de préférence des fichiers distincts comportant la définition des messages pour chacune des langues, et ces fichiers seront inclus conditionnellement dans le code source.

```
#if FRANCAIS == 1
#include    "francais.h"
#else
#include    "anglais.h"
#endif
```

On peut aussi recourir à des dictionnaires chargés dynamiquement, ce qui représente une option totalement différente.

❑ Quelques cas particuliers pour les expressions du #if:

- Tout symbole non défini dans une expression traitée par le préprocesseur se remplace par la valeur 0:

  `#if int` ←→ `#if 0`

- le préprocesseur a priori ne connaît que les types **long** et **unsigned long** (mais ceci n'est pas toujours respecté par tous les compilateurs!). Il ne faudrait donc pas utiliser de valeurs réelles dans les expressions qui lui sont destinées.
- Les opérateurs *cast* et **sizeof** ne peuvent pas non plus s'utiliser dans de telles expressions.

Les tableaux

Introduction

Notons tout de suite que, comme bien souvent en C/C++, les notions les plus diverses ont des liens étroits entre elles, et pour les tableaux, si nous en donnons d'abord les bases nous devrons en reparler dans la suite car les tableaux s'utilisent peu sous la forme présentée ci-dessous; on les manipule le plus souvent par l'intermédiaire de pointeurs, ce que nous ne ferons pas encore dans ce chapitre et en C++ on utilise généralement des classes spécifiques pour de telles opérations.

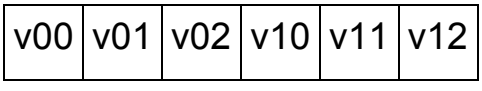
On peut représenter schématiquement un tableau à une dimension (unidimensionnel) sous la forme:

Ici nous avons un tableau de 5 éléments dont les valeurs respectives sont: v0, v1, v2, v3 et v4 (nous ne donnons encore aucune indication sur la notion d'indice pour l'instant!)

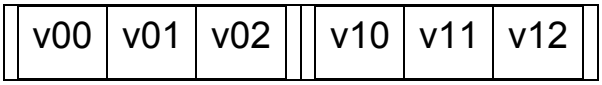
Un tableau à 2 dimensions quant à lui pourrait se représenter sous la forme:

Même si ceci offre une bonne idée de la notion de tableau à 2 dimensions, cette représentation ne reflète pas la réalité dans le langage étudié, pour lequel un nom de tableau consiste en réalité en un pointeur. Les éléments se suivent en mémoire.

Une représentation plus juste serait quelque chose du genre:



Ou, encore préférable:



Ceci généralisable à plus de 2 dimensions.

❑ *Les tableaux unidimensionnels*

Avant toute utilisation, un tableau doit faire l'objet d'une déclaration prenant la forme générale suivante:

```
classe type identificateur [expression] , ...;
```

Exemples:

```
int tableau [100];
#define limite 10
static char texte [ limite + 1 ];
```

classe:

Représente une spécification optionnelle, comme pour les variables simples, avec la même signification que celle vue dans les chapitres qui précèdent.

Bien entendu, la classe **register** ne s'applique pas aux tableaux!

type:

Il s'agit du type des éléments du tableau, de son contenu; comme dans les autres langages en général, les éléments d'un tableau sont tous du même type. Il peut s'agir d'un type de base connu du compilateur, un type pointeur, un type structuré que nous étudierons par la suite, ou un type tableau que nous sommes en train de découvrir.

identificateur:

Le nom que vous donnez à votre variable; il doit respecter les règles usuelles de syntaxe

des identificateurs.

expression:

Elle fixe la taille de votre tableau, son nombre d'éléments. Cette expression ne peut se composer que de termes constants, car elle doit être évaluable à la compilation.

La borne inférieure d'un indice de tableau vaut *toujours* 0 (on n'a pas le choix!).

Ainsi, la déclaration:

```
int t [5];
```

définit un tableau de 5 entiers dont les éléments sont:

```
t [0], t [1], t [2], t [3], t [4]
```

Lors de la référence à un élément de tableau, une expression à résultat entier (toujours au sens large) doit s'utiliser:

```
t [i] = t [k - 1];
```

Considérez un tableau non pas comme une variable, mais comme une collection de variables, toutes du même type.

Partout où l'on peut utiliser une variable simple d'un certain type, on peut également utiliser un élément de tableau de ce type.

On ne peut pas faire une affectation entre tableaux, ni réaliser d'autres opérations globales, mis à part le passage en paramètre.

Note: Pour un tableau déclaré externe, donc dont la définition réelle se trouve dans un autre module, en principe on ne donne pas de dimension; toutefois si nous le faisons elle peut être quelconque le compilateur ne l'utilisant de toute façon pas, alors autant ne pas la donner:

```
extern int t [ ];
```

❑ *Exemple*[1]

```
/*
  Programme de calcul des nombres premiers par la methode du crible
    d'Eratosthene. Montre la definition et l'utilisation de tableaux
  ERATOSTHENE (06 - tableau eratosthène)
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Definition des prototypes */
void afficher ( bool [], int );
void cribler  ( bool [], int );

/* Valeur maximale a atteindre */
#define MAXIMUM 500
/* Nombre maximum d'elements */
#define NB_ELEMENTS (MAXIMUM-3)/2+1


int main ( )
{
  /* Tableau pour le crible lui-meme */
  bool crible [ NB_ELEMENTS ];
  /* Initialiser le crible, a priori tout est premier  */
  for ( register int i = 0; i < NB_ELEMENTS; i++ )
           crible [ i ] = true;
  cribler  ( crible, NB_ELEMENTS );  // Determiner les nombres premiers
  afficher ( crible, NB_ELEMENTS );  // Afficher les resultats

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

[1] En C, utilisez un tableau d'éléments de type **char** (ou **_Bool** en C99) et les valeurs 0/1 en lieu et place de **false** et **true**!

```
/* Nombre de valeurs par ligne */
#define L_LIGNE 8
/* Sous-programme pour afficher les resultats        */
/* Le tableau et la limite sont passes en parametres */
void afficher ( const bool crible [], int limite )
{
  int nbVal = 3; // Compteur de valeurs par ligne
  /* Traiter les 2 premieres valeurs comme cas particuliers */
  cout << 1 << '\t' << 2 << '\t';
  /* Traiter tous les elements du tableau */
  for ( register int i = 0; i < limite; i++ )
    if ( crible [i] ) // Si c'est un nombre premier...
    {
      cout << 2*i+3 << '\t';     // ... l'afficher
      /* Passer a la ligne toutes les 8 valeurs    */
      if ( (nbVal++ % L_LIGNE) == 0 ) cout << endl;
    }
} /* afficher */



/* Sous-programme realisant le crible a proprement parler */
/* Le tableau et la limite sont passes en parametres      */
void cribler ( bool crible [], int limite )
{
  /* Traiter tout le tableau */
  for ( register int i = 0; i < limite; i++ )
    if ( crible [i] ) // Si c'est un nombre premier...
      /* ... enlever tous ses multiples */
      for ( register int k = 3*( i+1 ); k < limite; k += 2*i+3 )
        crible [k] = false;
} /* cribler */
```

L'exécution de ce programme donne le résultat:

```
     1       2       3       5       7       11      13      17
     19      23      29      31      37      41      43      47
     53      59      61      67      71      73      79      83
     89      97      101     103     107     109     113     127
     131     137     139     149     151     157     163     167
     173     179     181     191     193     197     199     211
     223     227     229     233     239     241     251     257
     263     269     271     277     281     283     293     307
     311     313     317     331     337     347     349     353
     359     367     373     379     383     389     397     401
     409     419     421     431     433     439     443     449
     457     461     463     467     479     487     491     499

     Fin du programme...Appuyez sur une touche pour continuer...
```

Comme le montre le programme exemple, le passage en paramètre d'un tableau ne se comporte pas comme pour les variables simples où, nous l'avions vu, on transmet par défaut une copie de la valeur du paramètre effectif. Lorsque nous transmettons un tableau en paramètre, il s'agit en fait de son adresse (un pointeur). Nous ne devons surtout pas lui ajouter l'opérateur d'adresse (**&**) comme nous le faisons pour les variables simples lorsque nous voulons obtenir un paramètre de sortie.

Dans la définition du sous-programme, on indique que le paramètre consiste en un tableau par de simples parenthèses carrées vides "[ ]", ne précisant en principe pas la taille de celui-ci:

```
int sp ( int t [] )
{ ...
```

Toutefois il n'est pas interdit d'ajouter dans les [ ] une constante (spécifiant théoriquement la taille), mais dans le cas de nos tableaux à 1 dimension elle ne sert à rien puisque pas utilisée par le compilateur:

```
int sp ( int t [5] )
{ ...
```

Pour une utilisation dans les limites fixées par la définition du tableau, on doit passer sa dimension comme paramètre supplémentaire, aucun autre outil ne permettant de retrouver cette information lorsqu'on se trouve dans la fonction:

```
int sp ( int t [ ], int n )
{
   ...
   for ( int i = 0; i < n; i++ )
      t [i] = i; // par exemple
```

Attention, en C/C++ aucun contrôle n'est effectué sur le débordement des indices pour ce genre de tableaux!

Cela signifie que nous pouvons accéder à des pseudo-éléments du tableau qui en réalité n'existent pas et par conséquent détruire éventuellement d'autres informations du programme.

Notons encore que dans une même déclaration, on peut définir des objets de nature différente.

Ainsi:

```
int tab [10], j, k;
```

déclare un tableau de 10 éléments entiers (*tab*) et 2 variables simples aussi entières (*j* et *k*). Toutefois, nous vous déconseillons cette formulation pour des raisons de lisibilité et donc de facilité de compréhension. Utilisez de préférence des déclarations distinctes:

```
int tab [10]; // .....
int j, k;     // .....
```

Avec nos connaissances actuelles du langage et les contraintes imposées, nous ne sommes pas capables de déclarer un tableau dont la dimension est fixée en cours d'exécution! [1]

❑ *Initialisation de tableaux*

Nous pouvons initialiser un tableau lors de sa déclaration, ceci par un mécanisme de type "agrégat[2]":

```
int tab [5] = { 1, 3, 7, 9, 11 };
```
[3]

Il suffit de mettre entre accolades la liste des valeurs. Cette liste n'a pas besoin d'être complète; en d'autres termes, il est possible d'initialiser qu'une partie du tableau, ses premiers éléments:

```
static int tab [5] = { 2, 4, 6 };
```

Ici tab [0] = 2, tab [1] = 4, tab [2] = 6, tab [3] = 0 et tab [4] = 0

Les 2 derniers éléments du tableau ne sont pas initialisés explicitement, mais par défaut ils sont mis à zéro, ceci aussi pour la classe **auto**.

[1] Nous reviendrons sur cette question dans la suite du cours.

[2] Terminologie non C++, mais utilisée dans d'autres langages!

[3] Un agrégat peut être vide, ce qui crée un tableau vide qui peut poser des problèmes par la suite.

Pour un tableau que l'on initialise lors de sa déclaration, il n'est pas indispensable de préciser sa taille, celle-ci pouvant se déduire de l'expression d'initialisation, par contre dans ce cas nous devons évidemment fournir les valeurs de tous ses éléments:

```
int tab [ ] = { 1, 2, 3, 4 };
```

qui correspond à:

```
int tab [4] = { 1, 2, 3, 4 };
```

Par contre, il n'est pas possible de spécifier explicitement une dimension et de donner une liste d'initialisation comportant plus d'éléments.

❑ *Fonctions générales pour tableaux à 1 dimension*

Pour les tableaux à une dimension, nous pouvons assez facilement réaliser des fonctions générales de traitement s'adaptant à des tableaux de tailles différentes.

Exemple pour illustrer ceci: recherche de la plus petite valeur dans un tableau non trié d'entiers:

```
/*
  Recherche de la plus petite valeur dans un vecteur
    PLUSPETIT1
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Fonction livrant la plus petite valeur d'un vecteur
    d'entiers de taille quelconque
*/

int minimum ( int vecteur [], unsigned int taille )
{ /* Le cas du tableau vide n'est pas gere ici! */
  int plusPetit = vecteur [0];
  for ( register int i = 1; i < taille; i++ )
    if ( vecteur [i] < plusPetit )
      plusPetit = vecteur [i];
  return plusPetit;
}
```

```
int  main ( )
{
  int v1 [] = { 1, 9, 3};
  int v2 [] = { 5, 7, 1, -2, 10 };
  cout << "Test de la fonction minimum" << endl << endl;
  cout << minimum ( v1, 3 ) << endl;
  cout << minimum ( v2, 5 ) << endl;
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Relevons toutefois qu'aucun contrôle n'étant réalisé par le compilateur, il incombe au programmeur de transmettre des choses cohérentes à l'appel; rien ne l'empêche par exemple d'écrire: `minimum ( v1, 333 )`; mais le résultat sera "n'importe quoi"! Par contre l'appel: `minimum ( v2, 3 )` peut être raisonnable, dans l'optique de ne traiter qu'une partie (une tranche) du tableau.

❑ *Les tableaux multidimensionnels*

Il s'agit simplement d'une généralisation des tableaux à une dimension. Le langage n'impose pas de limitation sur le nombre de dimensions. Nous décrirons les problèmes pour les tableaux à 2 dimensions, mais ceci se généralise à un nombre quelconque de dimensions.

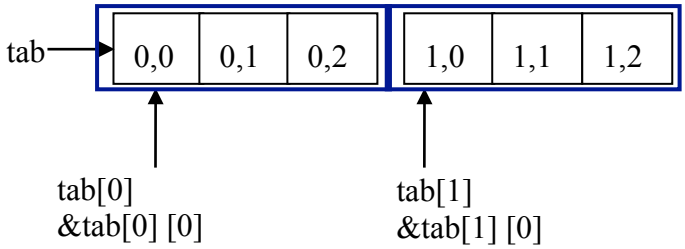
❑ Tableaux à 2 dimensions

Signalons tout de suite qu'il ne s'agit pas réellement de tableaux à 2 dimensions, mais bien de tableaux de tableaux.

Une déclaration prend à titre d'exemple la forme:

```
int tab [2] [3];
```

On peut le représenter de la manière suivante:



On précise chacune des tailles dans des parenthèses carrées séparées. Il en va de même lors de l'utilisation:

```
tab [i] [j] = 0;
```

Par rapport à la déclaration qui précède, *i* et *j* représentent des expressions de type entier qui devraient (mais rappelons qu'aucun contrôle n'est réalisé!) livrer des valeurs comprises entre 0 et 1 pour *i* et entre 0 et 2 pour *j* dans notre exemple.

Dans l'exemple de déclaration ci-dessus, nous avons un tableau de deux lignes, chaque ligne étant elle-même composée d'un tableau de 3 éléments, donc:

- tab désigne un tableau de tableaux
- tab [1] désigne un tableau (à 1 dimension et comportant 3 éléments de type **int**)
- tab [1] [1] désigne 1 élément simple de type entier.

Cette distinction est importante pour le passage en paramètre, l'affectation et par la suite l'utilisation de pointeurs; ce que nous avons déjà dit sur les tableaux à une dimension reste valable pour ceux à plusieurs dimensions.

Ainsi, nous pouvons et devrions initialiser un tableau à 2 dimensions par un agrégat d'agrégats[1]:

```
int tab [2] [3] =   {{ 0, 1, 2 }, // tab [0] []
                     { 3, 4, 5 }, // tab [1] []
                    };
```

Bien que purement formel, ceci aide à la lisibilité, car nous pouvons aussi écrire:

```
int tab [2] [3] = { 0, 1, 2, 3, 4, 5 };
```

[1] La norme C99 a introduit une forme d'initialisation par nom, décrite plus loin dans ce cours!

Remarquons que du point de vue de l'utilisation de la mémoire, le dernier indice de la déclaration varie le plus rapidement.

Rappelez-vous qu'une initialisation n'a pas besoin d'être complète; nous pouvons donc écrire:

```
int tab [2] [3] = { 0, 1, 2, 1 };
```

Dans ce cas *tab [1] [1]* et *tab [1] [2]* ne sont pas initialisés explicitement, mais ils contiendront la valeur 0.

Nous pouvons aussi écrire cette déclaration sous la forme plus lisible:

```
int tab [2] [3] = { { 0, 1, 2 },
                    { 1 }
                  };
```

La notation *agrégat d'agrégats* permet de laisser non initialisées explicitement des positions plus spécifiques; par exemple:

```
int tab [2] [3] = { { 0, 1 },
                    { 2 }
                  }
```

Ici ce sont *tab [0] [2]*, *tab [1] [1]* et *tab [1] [2]* qui ne sont pas initialisés explicitement mais par défaut ils valent 0.

A titre d'exemple donnons encore la déclaration d'un tableau à 3 dimensions:

```
int tab [3] [5] [7];
```

Si dans le code nous devons initialiser les éléments du tableau à des valeurs entières successives:

```
int m = 0
for ( int i = 0; i < 3; i++ )
  for ( int j = 0; j < 5; j++ )
    for ( int k = 0; k < 7; k++ )
      tab [i] [j] [k] = m++;
```

Les boucles **for** associées aux opérateurs d'auto incrémentation, respectivement décrémentation, sont des constructions typiques associées au traitement des tableaux!

❑ *Problème du passage en paramètre*

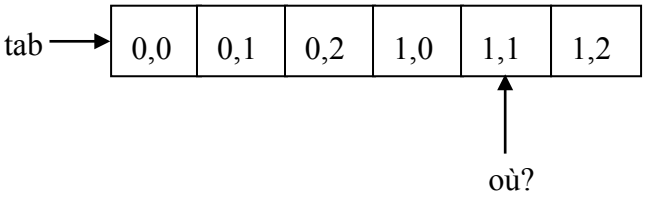
Pour le passage d'un tableau en paramètre, il faut que le compilateur puisse calculer exactement l'emplacement mémoire d'un élément. Pour les tableaux à une dimension, comme nous l'avons vu, aucun problème particulier ne se pose à part celui de savoir si nous nous référons bien à un élément qui existe par rapport à la définition du tableau, mais ce point ne relève pas du compilateur mais du programmeur.

Pour les tableaux à plusieurs dimensions la situation se complique. N'oublions pas que les éléments du tableau sont enregistrés dans une zone contiguë de la mémoire, avec du point de vue de l'ordre de stockage, le dernier indice qui varie le plus rapidement.

Ainsi, pour la déclaration:

```
int tab [2] [3]
```

nous trouvons en mémoire:



Dans ces conditions, comment savoir où se trouve en mémoire l'élément *tab [1][1]* par exemple?

Quand nous transmettons en paramètre le tableau *tab*, en fait c'est l'adresse de son premier élément qui est transmise (*&tab [0][0]*). Il faut que le compilateur puisse générer le code pour calculer la bonne position mémoire contenant l'élément *tab [1][1]* . Pour réaliser cette opération, il doit disposer de la grandeur de chaque dimension, sauf éventuellement la première.

Dans ce but nous devons préciser dans l'en-tête de la fonction, le nombre d'éléments du (ou des) indice(s) autre(s) que le premier. Soit par exemple, pour un tableau à 2 dimensions:

```
void f ( int tab [ ] [15], int n, int m );
```

Ici *n* et *m* seront effectivement utilisés dans le code pour fixer par exemple les limites des boucles de parcours du tableau.

Bien que cela ne soit pas indispensable, on peut aussi imaginer pour cette même fonction l'en-tête:

```
void f ( int tab [7] [15], int n, int m )
```

Mais inutile puisqu'il n'y a pas de contrôle de débordement!!!

Nous reviendrons dans le contexte des pointeurs sur le calcul que doit faire le compilateur afin de déterminer la position en mémoire d'un élément du tableau.

Pour un tableau à 3 dimensions, on aura quelque chose du genre:

```
void f ( int tab [ ] [15] [5], int n, int m, int k )
```

ou éventuellement:

```
void f ( int tab [7] [15] [5], int n, int m, int k )
```

Note complémentaire:

Puisque, lorsque l'on passe un tableau en paramètre c'est son adresse qui est transmise, rien n'empêche a priori de modifier le contenu de ce tableau; afin de prévenir ce genre de problèmes contre une erreur de programmation, il est possible de préciser dans la définition d'un paramètre qu'il s'agit d'une constante, donc non modifiable:

```
int longueurChaine ( const char chaine [] )
```

❑ *Fonctions générales pour tableaux multidimensionnels?*

La réponse au point d'interrogation du titre de ce paragraphe est malheureusement non avec la forme actuelle de nos tableaux; il nous faudra attendre les pointeurs pour C/C++, ou la présentation de la classe *vector*[1] pour C++ afin de résoudre ce problème. Ce que nous pouvons espérer au mieux pour l'instant: laisser à l'utilisateur la liberté de choisir la taille du premier indice de son tableau, rien de plus!

Voici un petit exemple illustrant cette pseudo liberté:

```
/*
  Recherche de la plus petite valeur dans une matrice
    PLUSPETIT2
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Fonction livrant la plus petite valeur d'une matrice
    d'entiers de premiere dimension quelconque (autre que 0)
*/
int minimum ( int tab [][2], unsigned int t1,  unsigned int t2 )
{
  int plusPetit = tab [0][0];
  for ( register int i = 0; i < t1; i++ )
    for ( register int j = 0; j < t2; j++ )
      if ( tab [i][j] < plusPetit )
        plusPetit = tab [i][j];
  return plusPetit;
}


int  main ( )
{
  int t1 [3][2] = { {1, 9}, {3, 5}, {9, 2 }};
  int t2 [1][2] = { 7, 5 };
  cout << "Test de la fonction minimum" << endl << endl;
  cout << minimum ( t1, 3, 2 ) << endl;
  cout << minimum ( t2, 1, 2 ) << endl;
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

[1] Ou d'un autre conteneur!

❑ *Un peu de classe* [1]

Nous l'avons signalé dès le départ, notre but dans cette première partie de cours ne consiste pas à introduire la programmation orientée objet. Toutefois nous ne pouvons pas programmer en C++ sans en utiliser un peu; d'ailleurs nous l'avons fait sans réellement le dire avec l'utilisation de *cin* et *cout*. Nous allons continuer pour l'instant à ne pas préciser comment créer des classes et les autres concepts liés, mais nous introduisons un minimum d'informations pour pouvoir utiliser réellement nos premiers outils, en l'occurrence la classe *vector* que nous allons utiliser pour manipuler proprement et de manière sécurisée des tableaux.

Dans une première approche nous pouvons dire qu'une classe correspond à une définition de type qui comporte non seulement des données, mais aussi les fonctions applicables à ces données et appelées **méthodes**.

De plus, une classe en C++ peut être générique (**template**), c'est-à-dire qu'il s'agit d'un "moule" applicable à différents types. Au moment de la création de l'objet désiré on indique le(s) type(s) à utiliser. Ce sera le cas de la classe *vector* car nous voulons pouvoir traiter des tableaux aussi bien d'entiers, de réels ou de n'importe quel autre type que nous apprendrons encore à construire, et pourquoi pas de tableaux pour disposer ainsi de tableaux à plusieurs dimensions.

Lorsque nous disposons d'une classe, appelons-la *t_Classe*, nous déclarons une variable (un objet) de cette classe a priori comme n'importe quelle autre déclaration:

```
t_Classe monObjet
```

S'il s'agit d'une classe générique (template), comme *vector*, il faudra encore, après le nom de la classe et entre < > spécifier le(s) paramètre(s) précisant la nature des éléments que l'on veut obtenir.

Exemple:

```
vector<int> unVecteur;
```

Nous disposons maintenant d'une variable *un_Vecteur* ne comportant pour l'instant aucun élément, car pour obtenir cette variable nous avons utilisé un **constructeur** par défaut fourni par la classe. Généralement les classes mettent à disposition des constructeurs plus complexes permettant de fixer des caractéristiques de l'objet obtenu et/ou de l'initialiser avec des valeurs spécifiques.

[1] Ce paragraphe et le suivant qui servent d'introduction à la classe *vector* n'ont de sens que pour C++.

Par exemple pour la classe *vector* nous pouvons fixer la taille du tableau, mais celle-ci pourra, comme nous le verrons, changer en cours de traitement:

```
vector<int> unVecteur (10);
```

Ici nous avons créé *unVecteur* de 10 éléments initialisés à 0.

En fait un constructeur consiste en une fonction (**méthode**) un peu particulière qui porte le nom de la classe. On lui transmet les informations nécessaires (ses paramètres) entre des parenthèses après le nom de la variable.

Nous l'avons dit, une classe offre généralement d'autres fonctionnalités (méthodes) applicables à ses propres objets. Pour utiliser ces méthodes nous devons donner le nom de l'objet concerné suivi d'un "." puis le nom de la méthode désirée avec entre parenthèses ses paramètres, ou des parenthèses vides s'il n'existe aucun paramètre:

```
unObjet.uneMethode ( lesParametres );
```

Note:

- Les méthodes, comme les fonctions ordinaires peuvent se surcharger; c'est d'ailleurs très souvent le cas pour les constructeurs.
- Comme nous le verrons par la suite, les opérateurs peuvent être surchargés; la classe *vector* le fait entre autre pour "=", "[ ]". Oui il s'agit bien d'opérateurs!

Voilà pour nos premières généralités sur l'utilisation des classes; nous aurons par la suite encore bien des choses à préciser dans ce domaine!

Nous allons maintenant décrire quelques spécificités complémentaires de la classe *vector* afin que nous puissions l'utiliser concrètement, mais nous ne détaillerons pas toutes ses possibilités, ce n'est pas notre objectif ici!

La classe vector

Pour l'utiliser notre application doit comporter:

```
#include <vector>
```

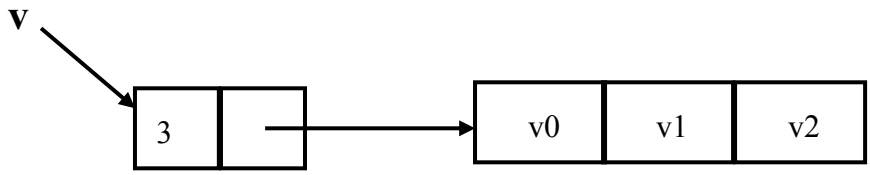
La classe *vector* fait partie de ce que l'on appelle les ***conteneurs*** qui globalement permettent de gérer des structures de données plus ou moins complexes, chacun de ces conteneurs étant spécialisé (performances!) dans un certain type de traitement (création, suppression, …). La taille (physique en mémoire) des structures gérées par les conteneurs peut varier en cours de traitement, c'est-à-dire que nous pouvons leur ajouter/enlever dynamiquement des informations.

Typiquement la classe *vector* nous assure un accès efficace à l'ensemble de ses éléments, tant en lecture qu'en écriture. Par contre l'extension de la structure (ajout de nouvelles valeurs) ou la suppression d'éléments n'offre pas des performances idéales, sauf si l'opération se réalise en fin de structure.

Nous avons appris ci-dessus à déclarer des objets de la classe *vector*:

```
vector<int> v ( 3 );
```

Contrairement aux tableaux classiques étudiés au préalable, un tel objet possède en mémoire l'apparence suivante:



Nous disposons donc non seulement de l'adresse du début des données en mémoire, mais en plus du nombre de ses données, donc de la taille de la structure. Ceci facilitera le codage des applications et permettra de réaliser des contrôles sur l'existence effective des éléments accédés.

Que pouvons-nous faire avec de tels objets? Nous allons vous présenter certaines des possibilités existantes, mais de loin pas toutes.

L'accès aux éléments peut se faire exactement de la même manière que pour les tableaux classiques, puisque nous l'avons dit la classe surcharge l'opérateur **[ ]** :

```
v[1] = 2;
```

Dans ce cas-là, comme pour les tableaux classiques, aucun contrôle ne s'effectue, vous pouvez accéder à des éléments inexistants et éventuellement détruire d'autres informations.

Heureusement la classe nous offre aussi une méthode (*at*) réalisant la même opération, mais cette fois avec un contrôle de l'existence de l'élément accédé, mais d'utilisation un peu moins agréable!

```
v.at(1) = 2;
```

Attention à la notation; ici le 1 représente le paramètre de la méthode *at*; il se met donc entre parenthèses alors que sous l'autre forme il s'agit de l'opérateur d'indexation qui se représente par des crochets.

Si l'élément accédé n'existe pas, une exception (*out_of_range*) sera levée; nous ne savons pas encore les traiter (mais cela viendra!) et donc pour l'instant le programme s'arrêtera brutalement. De toute façon ceci semble en général nettement préférable à un résultat que l'on croit juste et qui pourtant ne l'est pas!

Contrairement aux tableaux classiques dont le nom représente une adresse constante, nous pouvons effectuer une affectation globale entre 2 objets *vector* d'éléments de même type (pas forcément de même taille).

```
v1 = v2;
```

Les paramètres d'une méthode (d'une fonction en général) peuvent être d'une classe, donc de *vector* entre autres:

```
void f ( vector<int> v );
```

La méthode *size()* appliquée à un *vector* livre la taille actuelle (le nombre d'éléments) de l'objet en question. Ceci nous permet entre autres de résoudre le problème que nous avions pour transmettre à des fonctions des paramètres tableaux de tailles variables:

```
for ( register int i = 0; i < v.size(); i++ )
```

Nous pouvons ajouter dans un tel tableau un élément supplémentaire; toutefois, comme nous l'avons indiqué, seul l'adjonction en fin de structure est efficace aussi nous ne présentons que cette possibilité, réalisée par l'intermédiaire de la méthode *push_back*:

```
v.push_back ( valeur );
```

Le paramètre *valeur*, du type des éléments du tableau, représente le contenu du nouvel élément. La taille (*size()*) du tableau augmente alors de 1.

De manière symétrique nous pouvons supprimer un élément de la structure; pour l'extraire en fin nous utilisons la méthode sans paramètre *pop_back*:

```
v.pop_back ( );
```

La taille (*size()*) du tableau diminue alors de 1.

Voilà pour les notions de base; reprenons à titre d'exemple notre problème de la recherche

de la plus petite valeur dans un vecteur, auquel nous ajoutons quelques spécificités pour démontrer les possibilités de la classe:

```
/*
  Recherche de la plus petite valeur dans un vecteur
    PLUSPETIT3
*/
#include <cstdlib>
#include <iostream>
#include <vector>
using namespace std;

/* Fonction livrant la plus petite valeur d'un vecteur
    d'entiers de taille quelconque (differente de 0)
*/
int minimum ( vector<int> vecteur )
{
  int plusPetit = vecteur [0];
  for ( register int i = 1; i < vecteur.size(); i++ )
    if ( vecteur [i] < plusPetit )
      plusPetit = vecteur [i];
  return plusPetit;
}


int  main ( )
{
  vector<int> v1 ( 2 );
  vector<int> v2 ( 2 );
  v1.at(0) = 5; v1[1] = 2;
  v2 = v1;
  v2.push_back (-2);
  cout << "Test de la fonction minimum" << endl << endl;
  cout << minimum ( v1 ) << endl;
  cout << minimum ( v2 ) << endl;
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Et maintenant comment faire pour un tableau à plusieurs dimensions?

Le type des éléments d'un *vector* peut lui-même consisté en une classe quelconque donc pourquoi pas *vector*:

```
vector <vector<int> > tab;
```
[1]

Nous obtenons ainsi un tableau *tab* dont les éléments consistent en des vecteurs d'entiers.

[1] Notez bien la présence obligatoire de l'espace entre les 2 ">". Si votre compilateur compile selon la norme C++11, cet espace n'est pas obligatoire.

A priori pour l'instant *tab* ne contient aucun élément.

La méthode *resize* permet de donner une nouvelle taille à un *vector*:

```
tab.resize ( 5 );
```

Son paramètre, ici 5, représente la nouvelle taille de l'objet, qui peut être plus grande ou plus petite qu'avant. Attention toutefois, si la taille devient plus grande, il y a de forte chance que l'objet se trouve à une nouvelle adresse en mémoire. Ceci implique que:

- L'opération s'avère coûteuse en temps machine.
- Si nous avions d'autres pointeurs désignant cet objet, nous ne pouvons plus les utiliser car ils ne correspondent plus à la réalité!

Avec notre opération ci-dessus nous n'avons redimensionné que le nombre de vecteurs désignés par ce tableau, mais les vecteurs eux-mêmes ne possèdent encore aucune dimension fixée; nous devons le faire explicitement pour chacun d'eux:

```
for ( int ligne = 0; ligne < tab.size(); ligne++ )
                tab [ligne].resize ( 3 );
```

Maintenant nous avons un tableau de 5 vecteurs comportant chacun 3 éléments.

Notez que cette technique nous permet de construire des tableaux dont chaque élément peut être un tableau de taille différente; ceci sera utile par exemple pour construire des tableaux de chaînes de caractères.

A nouveau l'utilisation se fait comme pour un tableau normal:

```
cout << tab [3][2];
```

Regardons un exemple d'utilisation:

```
/*
  Exemple d'utilisation de matrices
    VECTOR2
*/
#include <cstdlib>
#include <iostream>
#include <vector>
using namespace std;
/* Dimensionnement et lecture d'une matrice */
void lireMatrice ( vector <vector<int> > &tab )
{
  unsigned int nbLignes, nbColonnes;
  cout << "Nombre de lignes: ";
  cin >> nbLignes;
  cout << "Nombre de colonnes: ";
  cin >> nbColonnes;
  tab.resize ( nbLignes );
  // Traiter tous les elements
  for ( int ligne = 0; ligne < nbLignes; ligne++ )
  {
    tab [ligne].resize ( nbColonnes );
    for ( int colonne = 0; colonne < nbColonnes; colonne++ )
    {
      cout << "Element(" << ligne << ',' << colonne << "): ";
      cin >> tab [ligne][colonne];
    }
  }
}

/* Affichage d'une matrice */
void afficheMatrice ( const vector <vector<int> > &tab )
{
  // Traiter tous les elements
  for ( int ligne = 0; ligne < tab.size(); ligne++ )
  {
    for ( int colonne = 0; colonne < tab[ligne].size(); colonne++ )
      cout << tab [ligne][colonne] << '\t';
    cout << endl;
  }
}

int  main ( )
{
  vector <vector<int> > tab;
  cout << "Utilisation de la classe vector" << endl << endl;
  lireMatrice ( tab );
  afficheMatrice ( tab );
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Notez sur cet exemple le passage du paramètre par référence:

```
void lireMatrice ( vector <vector<int> > &tab );
```

Nous avons dit que pour la classe *vector* l'adjonction et la suppression d'éléments se révèle efficace en fin de structure; nous pouvons donc facilement gérer une pile. Avant d'en donner un exemple d'utilisation, introduisons encore 2 méthodes:

La méthode *empty( )*, sans paramètre et qui livre *true* si le *vector* auquel on l'applique ne contient aucun élément et *false* sinon. Notez que l'on peut facilement s'en passer puisqu'il suffit de tester sa taille (*size( )*).

Ensuite la méthode *back( )*, elle aussi sans paramètre, qui livre la valeur du dernier élément de la structure sans l'extraire de celle-ci.

Voici un exemple d'utilisation, pas encore idéal comme nous le verrons tout de suite après:

```
/*
  Gestion d'une pile directement a partir de vector
    PILE1
*/
#include <cstdlib>
#include <iostream>
#include <vector>
using namespace std;
int  main ( )
{
  int valeur;
  vector<int> pile;
  do   // Tant que l'utilisateur le veut...
  {
    cout << "Donnez une valeur entiere (0 pour terminer): ";
    cin >> valeur;
    pile.push_back ( valeur );
  } while ( valeur != 0 );  // Peut s'ecrire: while ( valeur )!
  pile.pop_back ();  // Plus economique qu'un test dans la boucle!
  /* Tant que la pile n'est pas vide */
  while ( !pile.empty() )
  {
    cout << pile.back() << endl;
    pile.pop_back ();
  }
  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Voilà, nous avons facilement réalisé une pile, mais en pratique nous ne procéderons certainement pas ainsi car il existe une classe dite ***adaptateur*** (*stack*) qui implémente la pile.

Pour l'utiliser:

```
#include <stack>
```

Nous créons un objet pile selon le même principe que pour les *vector*, soit pour une pile d'entiers:

```
stack<int> pile;
```

Pour mettre une valeur sur la pile (en son sommet) nous utilisons la méthode *push*:

```
pile.push ( 22 );
```

Pour enlever l'élément au sommet de la pile nous utilisons la méthode *pop*:

```
pile.pop ();
```

Finalement, à part *empty* comme pour *vector*, nous utilisons la méthode *top* pour obtenir la valeur se trouvant au sommet de la pile, ceci sans l'extraire:

```
cout << pile.top();
```

Voilà donc une deuxième version de notre programme, adaptée à la classe *stack*:

```
/*
  Gestion d'une pile a partir de l'adaptateur stack
    PILE2
*/
#include <cstdlib>
#include <iostream>
#include <stack>
using namespace std;

int  main ( )
{
  int valeur;
  stack<int> pile;
  /* Tant que l'utilisateur le veut...*/
  do
  {
    cout << "Donnez une valeur entiere (0 pour terminer): ";
    cin >> valeur;
    pile.push ( valeur );
  } while ( valeur != 0 );
  pile.pop ();  // Plus economique qu'un test dans la boucle!
```

```
  /* Tant que la pile n'est pas vide */
  while ( !pile.empty() )
  {
    cout << pile.top() << endl;
    pile.pop ();
  }
  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Remarques complémentaires:

- Si notre objectif consiste à faire du calcul sur des vecteurs il sera certainement utile de prendre en considération la classe *valarray* que nous aborderons plus tard.
- Par la suite, plusieurs aspects brièvement abordés ci-dessus pourront faire l'objet de traitements plus subtils!

❑ *Compléments[1] C99*

La norme C99 a introduit une simplification dans le domaine des tableaux par rapport aux versions précédentes du langage, mais ces possibilités n'existent pas en C++.

Tout d'abord nous pouvons utiliser des variables ou des constantes dans les expressions fixant les dimensions d'un tableau[2].

Ensuite, pour les paramètres d'une fonction on peut indiquer un tableau de taille variable avec la forme ci-dessous:

```
int sp ( int n, int tab [ n ] )
```

Dans ce cas le(s) paramètre(s) de taille, ici *n* doi(ven)t venir avant celui du tableau qui l'utilise, ici *tab*!

Généralisation pour un tableau à plusieurs dimensions, par exemple 2:

```
void f ( int n, int m, int tab [n] [m] )
```

Cette nouvelle forme permet de transmettre au sous-programme, comme paramètres effectifs des tableaux dont la taille peut varier d'un appel à l'autre.

Maintenant nous pouvons imaginer dans notre problème du crible d'Eratosthène de demander à l'utilisateur la limite qu'il désire atteindre.

Voici une autre version du programme utilisant ces possibilités:

```
/*
  Calcul des nombres premiers par la methode du crible
    d'Eratosthene. Montre l'utilisation de tableaux "C99"
  ERATOSTHENEC99
*/
#include <stdio.h>
#include <stdlib.h>
void afficher ( _Bool [], int );  // Definition des prototypes
void cribler  ( _Bool [], int );


int main ( void )
{
  int maximum;
  printf ( "Crible d'Eratosthene\n\n\n" );
```

[1] Cette partie n'est pas valable pour C++!!!

[2] Il existe toutefois quelques restrictions d'utilisation que nous ne décrirons pas ici.

```
  printf ( "Donnez la limite: " );
  scanf ( "%d", &maximum );

  int borne = ( maximum -3 ) / 2 + 1;
  _Bool crible [borne];  // Tableau pour le crible lui-meme

  /* Initialiser le crible, a priori tout est premier  */
  for ( register int i = 0; i <= borne; i++ )
    crible [ i ] = 1;

  cribler  ( crible, borne );// Determiner les nombres premiers
  afficher ( crible, borne );// Afficher les resultats

  printf ( "\n\nFin du programme... " );
  system ( "pause" );
  return EXIT_SUCCESS;
}



/* Nombre de valeurs par ligne */
#define L_LIGNE 8
/* Sous-programme pour afficher les resultats        */
/* Le tableau et la limite sont passes en parametres */
void afficher ( _Bool crible [], int limite )
{
  int nb_val = 3;     // Compteur de valeurs par ligne
  /* Traiter les 2 premieres valeurs comme cas particuliers */
  printf ( "%8d%8d", 1, 2);
  /* Traiter tous les elements du tableau */
  for ( register int i = 0; i < limite; i++ )
    if ( crible [i] ) // Si c'est un nombre premier...
    {
      printf ( "%8d", 2*i+3 );   // ... l'afficher
      /* Passer a la ligne toutes les 8 valeurs    */
      if ( (nb_val++ % L_LIGNE) == 0 ) printf ( "\n" );
    }
} /* afficher */


/* Sous-programme realisant le crible a proprement parler */
/* Le tableau et la limite sont passes en parametres      */
void cribler ( _Bool crible [], int limite )
{
  /* Traiter tout le tableau */
  for ( register int i = 0; i < limite; i++ )
  if ( crible [i] ) /* Si c'est un nombre premier...*/
    /* ... enlever tous ses multiples */
    for ( register int k = 3*( i+1 ); k < limite; k += 2*i+3 )
      crible [k] = 0;
} /* cribler */
```

En réalité, dans ce programme il serait certainement préférable de déclarer une constante pour *borne*, mais alors nous ne montrerions qu'incomplètement la possibilité offerte!

Restons encore un peu dans les tableaux de taille variable et les possibilités offertes depuis la norme C99 pour signaler que dans une fonction les bornes d'une variable locale tableau peuvent se fixer par l'intermédiaire d'une valeur transmise en paramètre à cette fonction:

```
void exemple ( int borne )
{
   float tableau [ borne ];
   …
}
```

❑ Initialisation par nom

La norme C99 a introduit la possibilité de désigner dans un agrégat les éléments d'un tableau par leur nom. **Cette possibilité ne se retrouve malheureusement pas en C++.**

L'opération se réalise sous la forme:

```
[indice] = valeur
```

Ceci se généralise pour un tableau à plusieurs dimensions:

```
[indice1] [indice2]  = valeur
```

Exemple:

```
int tab [3] = { [0] = 1, [1] = 5, [2] = 7 };
```

Cette notation offre la possibilité de ne pas nécessairement donner les éléments dans l'ordre:

```
int tab [3] = { [2] = 1, [1] = 5, [0] = 7 };
```

Elle présente aussi l'avantage de ne pas devoir nécessairement initialiser tous les éléments:

```
int tab [3] = { [1] = 1 };
```

Ici les éléments 0 et 2 sont mis par défaut à 0!

Nous pouvons mélanger les notations par nom et par position:

```
int tab [3] = { 1, [1] = 5, 7 };
```

La valeur donnée par position qui suit une valeur donnée par nom correspond toujours à l'élément suivant dans le tableau.

Attention aux effets indésirables:

```
int tab [3] = { 1, [0] = 5, 7 };
```

Dans cet exemple *tab[0]* prend la valeur 5, *tab[1]* la valeur 7 et le 1 donné initialement a été écrasé par le 5!

Un exemple de tableau à 2 dimensions:

```
int tab [3] [3] = { [0] [1]  = 1, [2] [1]  = 7 };
```

❑ *Complément C++11 : Listes d'initialisateurs*

La norme C++11 introduit la possibilité d'initialiser un *vector* facilement avec un agrégat :

```
vector<int> v = { 1, 2, 3, 4, 5 };
```

Cette notation rend l'initialisation d'un vecteur similaire à l'initialisation d'un tableau normal :

```
int t[] = { 1, 2, 3, 4, 5 };
```

Dans cette forme d'initialisation la taille du *vector* est déterminée par l'agrégat. Notons qu'il n'est pas possible de spécifier la taille séparément des valeurs d'initialisation comme avec les tableaux :

```
int t[5] = { 1, 2, 3 };
```

Notons aussi que la nouvelle forme d'initialisation est possible non seulement avec les **vectors**, mais avec tous les conteneurs.

❑ *Complément C++11 : Tableaux de taille fixe en tant que conteneur*

La norme C++11 ajoute la notion de tableaux de taille fixe qui se présentent comme des

conteneurs. Rappelons qu'un **vector** est un conteneur. Ces tableaux-conteneurs sont une sorte de **vector** léger avec une taille fixe.

L'avantage comparé aux tableaux normaux est l'héritage de toutes les propriétés des conteneurs, comme par exemple la vérification des limites des indices, ou la méthode `size()` pour obtenir la taille du tableau.

Pour utiliser les tableaux-conteneurs il faut inclure l'en-tête array.

Voici un exemple :

```
#include <iostream>
#include <cstdlib>
#include <array>
using namespace std;

int main() {
    int const TAILLE = 5;
    array<int, TAILLE>  t = { 1, 2, 3, 4, 5 };
    for (int i = 0; i < t.size(); i++)
        cout << t[i] << " ";
    cout << endl;
    return EXIT_SUCCESS;
}
```

Les Pointeurs

Introduction

Comme en C, la notion de pointeur est fondamentale en C++. Elle s'avère plus générale et surtout plus utilisée que dans bien d'autres langages, souvent pour des raisons (justifiées?) d'efficacité et dans d'autres circonstances parce que nous ne pouvons pas faire autrement. Nous nous bornerons dans ce chapitre à donner des concepts de base, mais il faudrait par la suite philosopher plus longuement sur les principes et la méthodologie d'utilisation. Nous y reviendrons dans un autre chapitre; il faudra aussi regarder les aspects spécifiques liés à d'autres notions que nous introduirons par la suite.

Rappelons tout d'abord qu'un pointeur n'est rien d'autre qu'une adresse ou, plus précisément, qu'un objet dont le contenu consiste en l'adresse d'un autre objet. Nous avons déjà vu très rapidement apparaître la notion d'adresse, donc de pointeur, lorsque nous voulions dans nos sous-programmes utiliser un paramètre de "sortie".

Rappels:

- Dans la définition de l'en-tête de la fonction nous devons indiquer qu'il s'agit d'un pointeur (d'une adresse) en mettant une étoile (*) devant le nom du paramètre formel.
- Lors de l'appel de la fonction nous devons mettre un *&* devant le nom du paramètre effectif [1] car les paramètres normalement se transmettent par valeur (recopie locale de la valeur).
- Pour utiliser notre paramètre dans la fonction, nous devons indiquer l'indirection par la notation * devant chaque référence au paramètre.

[1] **Rappel**: sauf pour un tableau, dont le nom représente déjà un pointeur, donc une adresse!

Déclarations

Un pointeur, comme n'importe quel autre objet, se déclare avant utilisation. Il faut tout de suite noter qu'un pointeur contient l'adresse d'un objet d'un certain type; ceci deviendra important pour la suite.

La déclaration d'un pointeur sur un entier prendra la forme:

```
int *ptr;
```

Plus généralement:

```
type *nom;
```

La déclaration peut aussi s'écrire:

```
type * ptr;
```

Certaines personnes semblent plus facilement comprendre ainsi et cela pourra aider dans la suite!

De même que pour les autres variables, nous pouvons lui attribuer une classe:

```
static int i, *ptr;
```

Comme nous l'avions vu *i* est ici de la classe statique, mais aussi le pointeur *ptr* et non pas la variable qui sera pointée lorsque nous lui affecterons l'adresse d'une variable entière. Pour l'instant notre pointeur ne désigne aucun objet!

Par contre il n'en va pas de même des attributs **const** et **volatile**:

```
const int i = 2, *ptr = …;
```

Ici *i* est une "variable" (constante) qui ne peut pas changer de valeur; *ptr* lui désigne un objet entier constant, qui ne peut donc pas changer de valeur, mais le pointeur quant à lui peut changer, donc désigner un autre objet.

Si l'on désire que le pointeur ne puisse pas changer de valeur, c'est avec sa spécification qu'il faut associer l'attribut **const**:

```
int i, * const ptr = …;
```

Le pointeur peut aussi être constant et désigner un objet constant:

```
const int * const ptr = …;
```

Les mêmes remarques que ci-dessus pour **const** s'appliquent à l'attribut **volatile**!

Pour qu'un objet devienne un pointeur, il suffit de faire précéder son identificateur du symbole *****.

Une telle déclaration réserve le pointeur, mais pas l'objet pointé!

De plus, le pointeur lui-même ne contient pas de valeur ou plutôt il contient n'importe quelle valeur, sauf s'il est de classe **static**[1].

Comme nous l'avions déjà vu dans une même déclaration, nous pouvons définir plusieurs objets (nous vous déconseillons toutefois d'utiliser cette possibilité!):

```
char ch, *pt1, *pt2, tab [6], *pt_tab [3];
```

La déclaration ci-dessus définit:

- Une variable simple de type caractère: *ch* pouvant contenir un caractère.
- Deux pointeurs sur des objets de type caractère: *pt1* et *pt2* pouvant chacun contenir l'adresse d'un objet de type caractère.
- Un tableau de 6 caractères, donc une chaîne: *tab*, chaque élément du tableau pouvant contenir un caractère.
- Un tableau de trois pointeurs sur des objets de type caractère: *pt_tab*, chaque élément du tableau pouvant contenir l'adresse d'un objet de type caractère.

Les déclarations peuvent s'imbriquer et parfois devenir assez complexes à interpréter. Nous vous donnons ci-après, assez "brutalement", un certain nombre d'exemples à méditer:

[1] Dans ce cas il contient une valeur NULL.

```
int **pt;
```

Un pointeur contenant l'adresse d'un pointeur lui-même contenant l'adresse d'un entier (pointeur sur un pointeur d'entier).

```
int *pt( );
```

Une fonction qui livre comme résultat un pointeur contenant l'adresse d'un entier. Pour un prototype de fonction par exemple.

```
int (*pt) ( );
```

Un pointeur contenant l'adresse d'une fonction retournant comme résultat un entier. Pour passer par exemple l'adresse d'une telle fonction en paramètre à une autre fonction (en d'autres termes, pour passer une fonction en paramètre).

```
int *(*pt) ( );
```

Un pointeur contenant l'adresse d'une fonction livrant comme résultat un pointeur contenant l'adresse d'un entier.

```
int (*pt) [5];
```

Un pointeur contenant l'adresse d'un tableau de 5 entiers. A ne pas confondre avec un tableau de 5 pointeurs sur des entiers: **int** *pt [5];

```
int (*pt ( )) [5];
```

Une fonction livrant comme résultat un pointeur contenant l'adresse d'un tableau de 5 entiers.

Les combinaisons peuvent être encore plus complexes que cela, mais attention tout de même, certaines combinaisons ne sont pas légales; nous ne pouvons pas avoir des tableaux de fonctions ni des fonctions livrant comme résultat un tableau.

Règle simple pour interpréter de telles déclarations:

Déterminez l'effet des parenthèses ( ) ou [ ] de gauche à droite en allant tant que possible en profondeur au niveau de leur imbrication.

Pour illustrer ceci partons de notre dernier exemple de déclaration ci-dessus:

```
int (*pt ( )) [5];
```

- Premier pas:
  *pt ( ) : fonction qui livre comme résultat un pointeur
- Deuxième pas:
  (*pt ( )) : ce pointeur contient l'adresse de …
- Troisième pas:
  (*pt ( )) [5] : ce pointeur contient l'adresse d'un tableau de 5 éléments
- Quatrième pas:
  int (*pt ( )) [5] : ce tableau à 5 éléments entiers

D'où finalement notre conclusion initiale:

```
int (*pt ( )) [5];
```

Est une fonction livrant comme résultat un pointeur contenant l'adresse d'un tableau de 5 entiers.

Nous avons dit au début qu'un pointeur contenait l'adresse d'un objet d'un type spécifique; ceci n'est que partiellement vrai, puisque nous pouvons écrire:

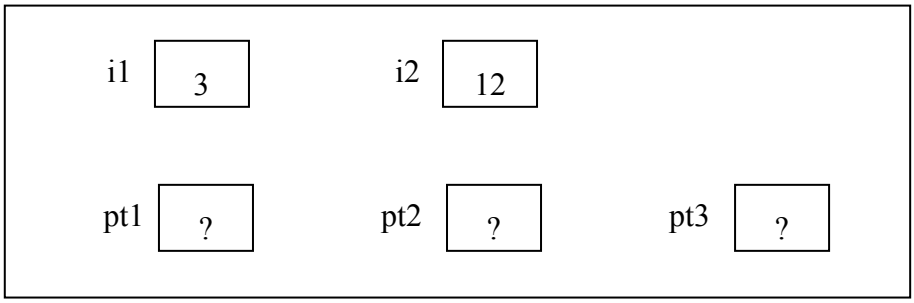
```
void *pt;
```

Pour désigner un pointeur "neutre" pouvant contenir l'adresse d'un objet de n'importe quel type, mais dans ce cas, nous perdons les propriétés spécifiques liées au calcul des adresses que nous introduirons par la suite.

Utilisation

Soient les déclarations:

```
int  i1 = 3, i2 = 12, *pt1, *pt2;
char *pt3;
```



Il existe une constante: *NULL*[1], importée de *cstdlib*[2], qui peut être utilisée en conjonction avec un pointeur sur n'importe quel type, pour indiquer qu'il ne contient pas d'adresse valable. Dans les instructions du programme, nous pouvons écrire:

```
pt1 = NULL; // ne designe aucun entier
pt3 = NULL; // ne designe aucun caractere
```

Nous utiliserons également des tests du genre:

```
if ( pt1 != NULL )
  ...; // Instructions a executer si pt1 designe un
objet
```

La constante NULL n'étant pas entièrement satisfaisante, la norme C++11 introduit une meilleure manière de désigner l'absence d'une adresse valable : le mot-clé *nullptr*. La section « Le mot-clé *nullptr* » explique son usage. Si le compilateur supporte la norme C++11, l'utilisation de *nullptr* au lieu de *NULL* est préférable.

Nous pouvons aussi affecter à un pointeur la valeur d'un autre pointeur du même type:

```
pt2 = pt1;
```

[1] Attention, la valeur effective de cette constante dépend du compilateur (souvent 0, parfois -1 et pourquoi pas encore autre chose!). Alors attention à ce que vous écrivez, même si l'on voit souvent dans des programmes ce NULL remplacé par 0!

[2] En C: *stdlib.h* et souvent aussi d'autres fichiers d'inclusion.

Mais pas d'un autre type:

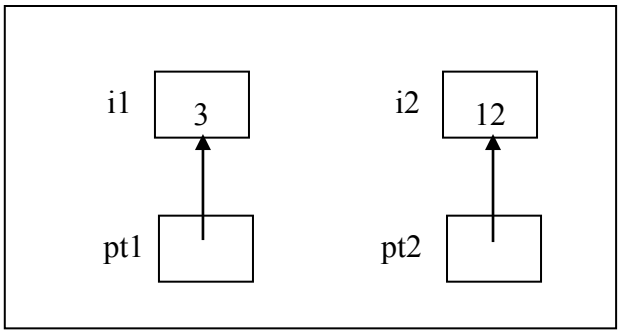
```
pt3 = pt1;  // Ceci est faux
```

Les pointeurs dans ce langage ne permettent pas seulement de manipuler des variables dynamiques (comme dans certains autres langages), mais également statiques et ceci s'utilise même très souvent pour des tableaux, comme nous le verrons.

Dans ce but, il nous faut un mécanisme permettant d'obtenir l'adresse d'un objet statique; l'opérateur **&** le fournit, comme nous le savons déjà.

Nous pouvons écrire dans les instructions d'un programme:

```
pt1 = &i1;
pt2 = &i2;
```



Maintenant *pt1* et *pt2* contiennent respectivement les adresses de *i1* et *i2*.

Pour accéder à l'objet pointé, il suffit de faire précéder l'identificateur de pointeur du symbole *, qui est l'opérateur d'indirection. Ainsi, avec les déclarations et les opérations faites ci-dessus:

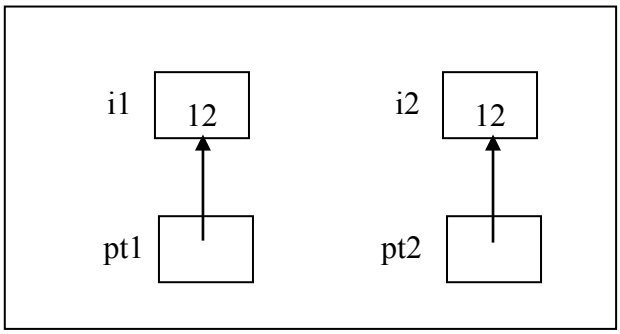
```
*pt1 = *pt2;
```

Revient au même que d'écrire:

```
i1 = i2;
```

Ou, pourquoi pas:

```
i1 = *pt2;      ou      *pt1 = i2;
```



Attention:

```
i1 = *pt1 * *pt2;
```

revient bien à écrire:

```
i1 = i1 * i2;
```

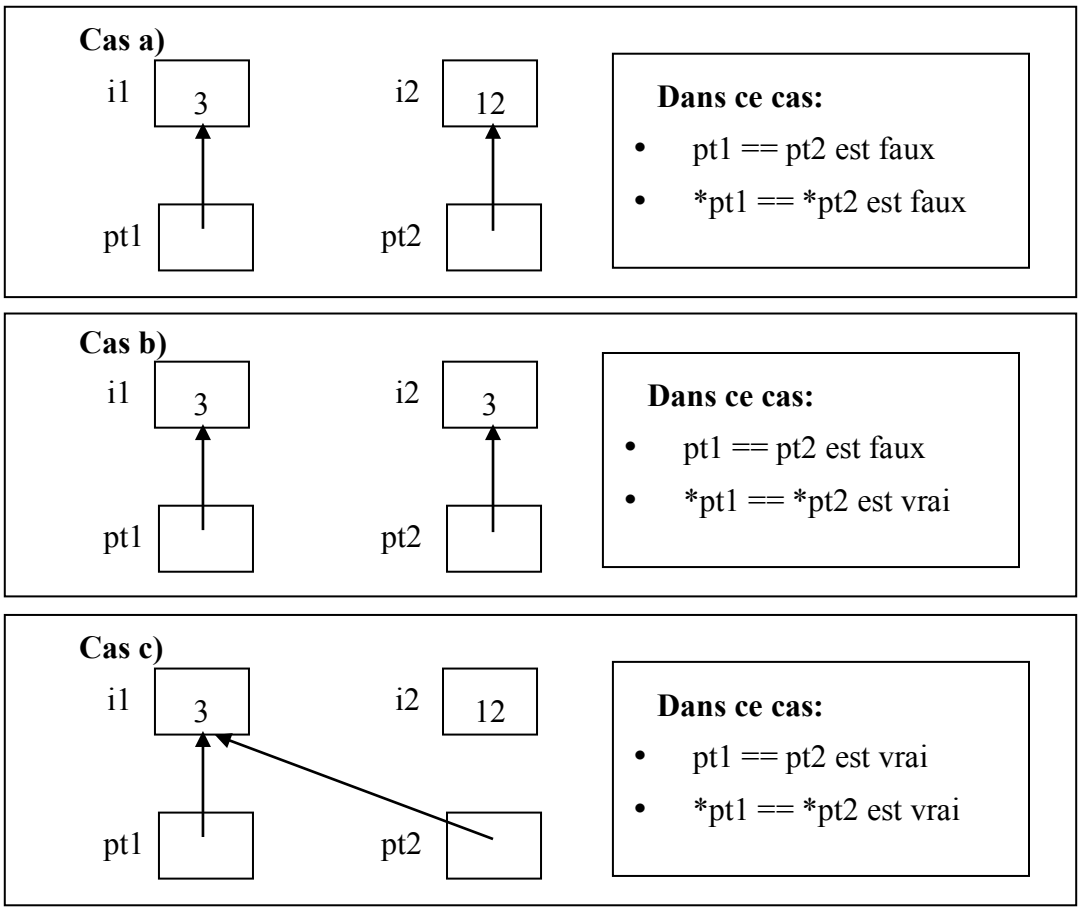
Le test:

```
if ( pt1 == pt2 )
   ...;
```

compare si les 2 pointeurs désignent la même cellule, alors que:

```
if ( *pt1 == *pt2 )
   ...;
```

compare si les 2 objets pointés respectivement par *pt1* et *pt2* contiennent les mêmes valeurs (les 2 objets pouvant éventuellement être physiquement confondus).



Partout où l'on peut utiliser l'identificateur d'un objet, on peut également utiliser un pointeur sur cet objet:

```
cout << *pt1;
```

Notons que le pointeur *NULL* est traité comme une valeur nulle; ceci implique qu'un test du genre:

```
if ( pt1 != NULL ) …
```

peut s'écrire plus simplement:

```
if ( pt1 ) …
```

mais certainement avec une lisibilité et une compréhesion moins bonne.

L'affichage du contenu d'un pointeur (et non de l'objet désigé par ce pointeur), donc d'une adresse:

```
int     * p  = &...;
...
cout << p << endl;
```

provoque l'affichage d'une valeur hexadécimale correspondant à l'adresse, par exemple:

```
0x22ff64
```

❑ Le mot-clé *nullptr* (C++11)

Dans le langage C++, la constante *NULL* est définie comme l'entier 0 (dans le langage C les définitions peuvent varier). Ceci pose problème dans le cas où le programme utilise la surcharge des fonctions, par exemple comme ceci :

```
void f(char *);
void f(int);
```

L'instruction `f(NULL)` appelle la fonction `f(int)` et non pas `f(char *)` comme on pourrait le penser.

Dans la nouvelle notation C++11, *nullptr* (du type prédéfini *nullptr_t*) prend la place de *NULL* et nous pouvons écrire :

```
pt1 = nullptr; // ne designe aucun entier
pt3 = nullptr; // ne designe aucun caractere
```

Pour tester si un pointeur designe un objet :

```
if ( pt1 != nullptr )
  ...; // Instructions a executer si pt1 designe un objet
```

Reprenant l'exemple de la surcharge des fonctions d'avant, l'instruction `f(nullptr)` appelle la fonction `f(char *)`.

Exemple:

```
/*     Programme exemple: Utilisation du type pointeur avec des
objets non dynamiques, ceci est courant en C, bien entendu, en
pratique, pas sur des cas aussi simples qu'ici
POINTEURS_STATIQUES (POINTEURS)
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  /* Definition d'une variable entiere simple et d'un tableau */
  int i1 = 1, i2 [2] = { 2, 3 };
  /* Definition de pointeurs sur des entiers */
  int  *p1  = &i1, // Pointeur initialise a l'adresse de i1
  *p2  = i2,       // Pointeur initialise au debut de i2
  **p3 = &p2;      // Pointeur: adresse de l'adresse de i2
  /* Utilisation simple de pointeurs */
  cout << "*p1 = " << *p1 << "   *p2 = " << *p2 << endl;
  cout << "**p3 = " << **p3 << endl;          // Adresse indirecte
  /* La valeur des adresses */
  cout << "&p1 = " << &p1 << "   &p2 = " << &p2 << endl;
  p1++; p2++;                        // Incrementation de pointeurs
  /* Affichage des nouvelles valeurs
       ... attention p1 designe n'importe quoi
  */
  cout << "*(++p1) = "  << *p1 << "   *(++p2) = " << *p2 << endl;
  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Résultats de l'exécution:

```
*p1 = 1   *p2 = 2
**p3 = 2
&p1 = 0x22ff64   &p2 = 0x22ff60
*(++p1) = 2293680   *(++p2) = 3
Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *Adresse des tableaux et calculs sur les pointeurs*

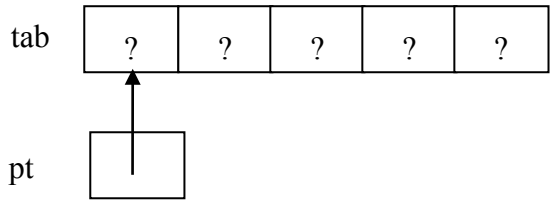
Nous pouvons réaliser certaines opérations arithmétiques sur les pointeurs.

Soient les déclarations:

```
int tab [5], *pt;
```

Nous avons déjà vu que lorsque nous passons un tableau en paramètre à une fonction, c'est son adresse qui est effectivement transmise. En fait, le nom d'un tableau, ici *tab*, est une constante de type pointeur:

```
pt = tab;  // Attention, ici pt = &tab serait faux!!
```



pour que *pt* prenne comme valeur l'adresse du tableau *tab* ou, ce qui revient au même, l'adresse de son premier élément: *tab* [0]; donc l'instruction ci-dessus peut également s'écrire:

```
pt = &tab [0];
```

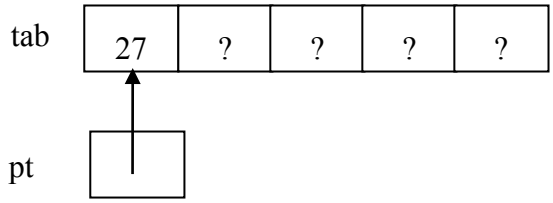
tab [0] désigne un objet simple, nous devons à nouveau utiliser l'opérateur d'adresse **&**.

Et dans les instructions qui suivent, écrire:

```
tab [0] = 27;
```

Revient au même que:

```
*pt = 27;
```

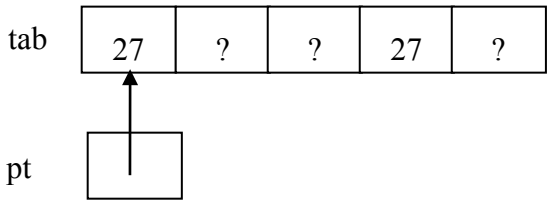


De plus l'instruction:

```
tab [3] = 27;
```

Peut également s'écrire:

```
*( pt + 3 ) = 27;
```



Ceci nous amène à introduire le principe de l'arithmétique sur les pointeurs.

Pour les tableaux à 2 dimensions, exemple:

```
int tab [3][2], * pt;
```

tab représente un tableau de pointeurs. Nous n'avons donc pas le droit d'écrire:

```
pt = tab;  // ERREUR
```

Le compilateur C++ le refusera, alors qu'un compilateur C se contentera d'un avertissement. Nous pouvons résoudre ceci de différentes manières:

```
pt = (int *)tab;
pt = tab [0];
pt = &tab [0][0];
```

Opérateurs arithmétiques sur les pointeurs

Observez attentivement l'instruction ci-dessus, un objet de type **int** occupe très certainement plus d'une adresse mémoire (le nombre dépend du matériel utilisé), nous pouvons donc affirmer que *tab* [3] se trouve à plus de 3 adresses mémoire de *tab* [0] (ou *tab*) et pourtant cela fonctionne correctement avec *(*pt* + 3) sans que nous, programmeurs, ayons à nous préoccuper de la taille des entiers sur notre machine. Pour cette raison il était essentiel dès le départ de préciser qu'un pointeur désigne toujours un objet d'un type déterminé (sauf le cas **void**); sans quoi le compilateur serait incapable de déterminer la bonne adresse, ne connaissant pas la taille des éléments du tableau.

Notons que si l'on désire lire le $i^{ème}$ élément d'un tableau *t* d'entiers, sous la forme tableau on écrit:

```
cin >> t [i];
```

Et sous la forme pointeur:

```
cin >> *( t + i );
```

Attention à bien indiquer l'indirection: `*( t + i )`; il en va de même pour l'affichage:

```
cout << *( t + i );
```

Sans les *( ) la valeur de l'adresse correspondante serait affichée.

Le prototype d'une fonction à laquelle nous passons en paramètre un tableau d'entiers s'écrit a priori, comme nous l'avions vu:

```
void exemple ( int tabl [] );
```

Dans le corps de cette fonction nous pouvons à nouveau accéder au $i^{ème}$ élément de ce tableau en écrivant:

```
tabl [i]
```

Mais aussi par:

```
*( tabl + i )
```

On peut modifier le prototype de la fonction, tout en gardant les mêmes possibilités d'utilisation, soit:

```
void exemple ( int *tabl )
```

Notez toutefois que là il est impossible de savoir si l'appelant transmet l'adresse d'une variable entière simple ou celle d'un tableau d'entiers!

D'autres opérations arithmétiques peuvent s'appliquer à des pointeurs:

- Incrémenter ou décrémenter un pointeur:

```
pt++
```

Ici encore l'incrémentation se fait de telle sorte que le pointeur désigne l'élément suivant du même type en mémoire; elle n'a de sens réel que pour un pointeur contenant l'adresse d'un tableau; pour l'adresse d'une variable simple, on va désigner n'importe quoi en mémoire (comme le montrait notre exemple de programme), et donc éventuellement faire des dégâts!

De par la priorité des opérateurs:

```
*++pt          est équivalent à      *(++pt)
```

et livre le contenu de l'objet qui vient après ***pt*** (l'élément qui suit dans le tableau!).

Note: la norme nous garantit que l'adresse qui suit le dernier élément d'un tableau correspond toujours à une adresse valable; nous pouvons donc la calculer et l'utiliser, mais bien entendu nous ne devons pas référencer l'élément désigné par cette adresse puisqu'il n'existe pas!

Il en va de même pour l'opérateur de décrémentation:

```
*--pt          est équivalent à      *(--pt)
```

qui livre le contenu de l'objet qui vient avant ***pt*** (l'élément qui précède dans le tableau!).

- Additionner, comme nous l'avons vu, mais aussi soustraire une valeur entière à un pointeur. A nouveau, le type (taille) de l'élément pointé entre en considération.

- La soustraction de 2 pointeurs, ce qui donne comme résultat le nombre d'objets du type considéré entre ces 2 adresses qui doivent logiquement appartenir au même objet tableau.

Aucune autre opération arithmétique sur les pointeurs n'est significative et permise.

Exemple

Reprenons à titre d'exemple le crible d'Eratosthène dans sa forme initiale que nous développerons plusieurs fois pour en donner diverses variantes:

```
/*
   Programme de calcul des nombres premiers par la methode du
   crible d'Eratosthene. Montre l'utilisation de tableaux et de
   pointeurs. Attention, dans cet exemple nous avons volontairement
   fait un melange complet des diverses notations pointeurs et
   tableaux, ceci  pour montrer les possibilites; cela ne veut pas
   dire qu'il faut proceder de meme dans la pratique!!!!
  ERATOSTHENE_POINTEURS
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Definition des prototypes */
void afficher ( bool *, int );
void cribler  ( bool [], int );

/* Valeur maximale a atteindre */
#define MAXIMUM 500
/* Nombre maximum d'elements */
#define Nb_Element (MAXIMUM-3)/2+1

int main ( )
{
  bool crible [Nb_Element];    // Tableau pour le crible lui-meme
  bool *pt = crible;           // Pointeur sur le tableau crible
  /* Pointeur sur le dernier element du tableau */
  const bool *fin = crible + Nb_Element;
  /* Initialiser le crible, a priori tout est premier */
  while ( pt != fin )
    *pt++ = true;
  cribler  ( crible, Nb_Element );  // Determiner les nombres premiers
  afficher ( crible, Nb_Element );  // Afficher les resultats

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

```
/* Sous-programme pour afficher les resultats        */
/* Le tableau et la limite sont passes en parametres */
void afficher ( const bool *crible, int limite )
{
  const int nbParLigne = 8;
  int nbVal = 3;      // Compteur de valeurs par ligne
  /* Traiter les 2 premieres valeurs comme cas particulier */
  cout << "\t1\t2";
  /* Traiter tous les elements du tableau */
  for ( register int i = 0; i < limite; i++ )
    if ( *(crible + i ) )    // Si c'est un nombre premier...
    {
      cout << '\t' << 2*i+3; // ... l'afficher
      /* Passer a la ligne si necessaire  */
      if ( (nbVal++ % nbParLigne) == 0 ) cout << endl;
    }
} // afficher



/* Sous-programme realisant le crible a proprement parler */
/* Le tableau et la limite sont passes en parametres      */
void cribler ( bool crible [], int limite )
{
  /* Traiter tout le tableau */
  for ( register int i = 0; i < limite; i++ )
    if ( crible [i] ) // Si c'est un nombre premier...
      /* ... enlever tous ses multiples */
      for ( register int k = 3*(i+1); k < limite; k += 2*i+3 )
        *(crible + k) = 0;
} // cribler
```

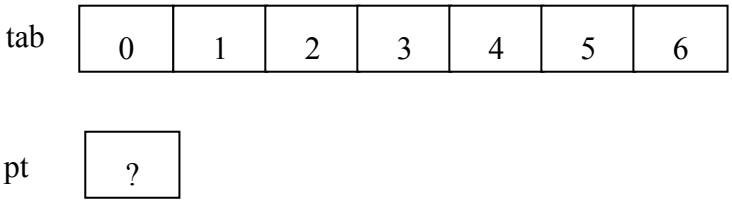
❑ *Remarques complémentaires:*

Si nous déclarons un tableau:

```
int tab [7] = { 0, 1, 2, 3, 4, 5, 6 };
```

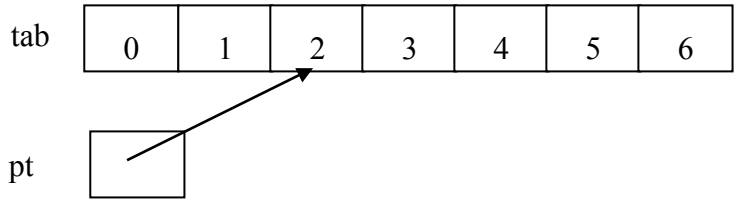
et un pointeur:

```
int *pt;
```



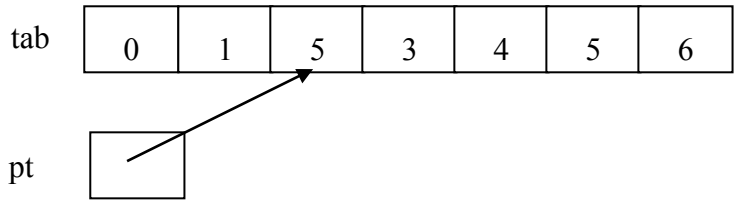
Dans les instructions, nous pouvons mettre dans *pt* l'adresse de n'importe quel élément du tableau, par exemple:

```
pt = &tab [2];
```



Comme nous pouvons mélanger les notations, rien ne nous interdit d'écrire:

```
pt [0] = 5;
```



Ce qui revient au même que:

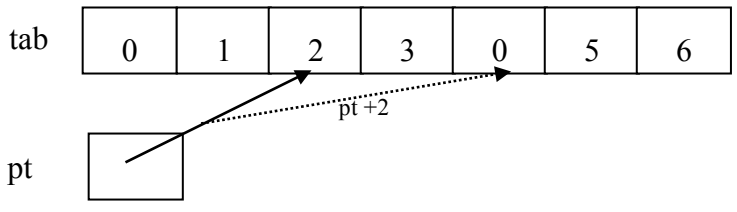
```
tab [2] = 5;
```

L'instruction:

```
pt [2] = 0;
```

revient elle au même que:

```
tab [4] = 0;
```



Comme nous l'avons dit, transmettre un tableau en paramètre à un sous-programme revient à transmettre un pointeur sur son premier élément, exemple:

```
int sp ( int tab [ ] );
```

Ceci a des conséquences sur l'utilisation de l'opérateur **sizeof**; si dans le corps du sous-programme nous écrivons:

```
… sizeof ( tab ) …
```

nous obtenons comme valeur non pas la taille du tableau, mais celle d'un pointeur sur un entier, donc cela revient au même que:

```
… sizeof ( int * ) …
```

❑ *Pointeurs sur les fonctions*

Un pointeur peut se référer à une fonction; comme pour les tableaux, le nom d'une fonction représente une adresse constante, donc un pointeur. Ceci s'avère très utile pour passer des fonctions en paramètre à d'autres fonctions (intégration numérique, recherche de zéro,...).

Exemple d'une déclaration de pointeur sur une fonction possédant un paramètre **double** et livrant un résultat **double**:

```
double (*pt)( double );
```

Nous pouvons affecter à ce pointeur l'adresse de n'importe quelle fonction correspondant à la description (un paramètre **double**, le résultat **double**). Par exemple, si dans notre programme nous disposons d'une telle fonction, disons *f*, l'affectation se réalise simplement par:

```
pt = f;
```

Maintenant si dans le programme *x* et *y* ont été déclarés **double**, nous pouvons appeler la fonction par:

```
y = (*pt) (x);
```

Ceci correspond à:

```
y = f (x);
```

Mais nous pouvons également encore réaliser cet appel sous une autre forme:

```
y = pt (x);
```

La fonction peut également venir de la bibliothèque, pourvu qu'elle possède le bon profil.

Voilà, tout ceci ne semble effectivement pas très compliqué. Nous pouvons donner maintenant un programme complet illustrant cette possibilité. Ce programme nous pourrons le réaliser plus simplement et plus proprement par la suite lorsque nous aurons étudié les définitions de types.

Le voici:

```
/*
  Exemple d'utilisation de pointeurs sur des fonctions
    PTFONCTIONS
*/
#include <cstdlib>
#include <cmath>
#include <iostream>
using namespace std;

// Quelques fonctions a titre d'exemples
double xCarre ( double x )
{
  return x * x;
}

double xPlus3 ( double x )
{
  return x + 3;
}

double xCube ( double x )
{
  return x * x * x;
}
```

```
int  main ( )
{
  const int nbElements = 5;
  double (*ptf)( double );
  char * tab [nbElements] =
                  { "x**2", "x+3", "x**3", "sinus", "cosinus" };
  double x;
  int choix;
  cout << "Evaluation d'une fonction\n\n";
  cout << "Donnez la valeur de x: ";
  cin >> x;
  cout << "Quelle fonction?" << endl;
  for ( int i = 0; i < nbElements; i++ ) // Affiche les possibilites
    cout << i + 1 << ")\t" << tab [i] << endl;
  cout << "Votre choix: ";
  cin >> choix;
  if ( choix > 0 && choix <= nbElements )
  { // Choix correct
    switch ( choix )
    { // Selectionner la fonction desiree
      case 1: ptf = xCarre; break;
      case 2: ptf = xPlus3; break;
      case 3: ptf = xCube;  break;
      case 4: ptf = sin;    break;
      case 5: ptf = cos;    break;
    }
    cout << "Valeur: " << (*ptf) (x) << endl;
    // Nous pouvons aussi ecrire:
    // cout << "Valeur: " << ptf (x) << endl;
  }
  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Résultat d'une exécution:

```
Evaluation d'une fonction

Donnez la valeur de x: 1.2
Quelle fonction?
1)      x**2
2)      x+3
3)      x**3
4)      sinus
5)      cosinus
Votre choix: 2
Valeur: 4.2
Fin du programme...Appuyez sur une touche pour continuer...
```

Remarque: le *#include <cmath>* nous met à disposition les fonctions *sin* et *cos*; dans un programme C il faudra le remplacer par: *#include <math.h>*.

❑ *Différence entre pointeur et tableau*

Nous avons fait un certain amalgame entre pointeur et tableau unidimensionnel, il faut toutefois bien considérer qu'une déclaration de pointeur ne réserve en mémoire que la place pour l'adresse d'un objet, alors qu'une déclaration de tableau réserve un objet "complet". Ceci implique que certaines choses sont faisables sous une forme et pas sous l'autre.

Soient les déclarations:

```
#define MAX ...
char texte [MAX];
const char *pt;
```

Nous pouvons écrire dans les instructions:

```
pt = "ENCORE SALUT";
```

Il n'y a pas ici création d'une nouvelle chaîne mais simplement affectation à *pt* de l'adresse de la constante!

Par contre:

```
texte = "ENCORE SALUT";     // Faux!!
```

Pas possible, car *texte* (étant un tableau qui existe) représente une constante pointeur, adresse qui ne peut pas être modifiée. Pour réaliser cette opération nous utiliserons des fonctions de la bibliothèque (par exemple *strcpy*) dont nous reparlerons dans le chapitre consacré aux chaînes de caractères.

Variables dynamiques/Retour sur les tableaux

La bibliothèque standard C++/C, dont nous reparlerons plus en détails par la suite, nous permet de gérer dynamiquement la mémoire. Cette possibilité ne s'utilise pas uniquement pour traiter des structures chaînées, comme dans certains autres langages.

Entre autres, cela nous permet de résoudre le problème des bornes de tableaux qui elles ne pouvaient pas être variables en C avant la norme C99 et qui ne le sont pas non plus en C++.

Opérateurs new et delete[1]

L'objectif consiste à créer les objets lorsque nous en avons besoin (**new**) et de restituer la place mémoire qu'ils occupent quand nous n'en avons plus besoin (**delete**). Les variables ainsi créées n'existent pas au moment de la compilation, on n'a donc pas pu les "nommer" et c'est par l'intermédiaire de pointeurs que nous accéderons à de tel objets.

L'opérateur **new** qui peut prendre 2 formes différentes, réserve en mémoire la place pour un objet du type désiré et nous livre comme résultat un pointeur sur cet objet. Nous pouvons utiliser ce pointeur comme n'importe quel autre pointeur. La première chose que nous ferons en général sera de sauver sa valeur dans une variable (pointeur) du type approprié.

Première forme:

```
new type
```

Exemple:

```
int * pt;
...
pt = new int;
```

Cette forme pour l'instant ne nous intéresse que peu puisque, en plus de la place mémoire nécessaire pour l'entier il nous faut aussi celle nécessaire au pointeur. Elle deviendra utile lorsque nous apprendrons à construire des structures chaînées et/ou dans le contexte de la programmation objet.

[1] Ces opérateurs ne sont pas disponibles en C; nous verrons dans la suite de ce chapitre comment résoudre ce même problème. La méthode qui sera alors proposée peut aussi s'utiliser en C++, toutefois nous vous conseillons de mettre en pratique les nouvelles formes.

Deuxième forme:

```
new type [taille]
```

Exemple:

```
int * pt;
...
pt = new int [20];
```

Cette forme nous permet de créer *taille* objets (dans notre exemple 20) du type spécifié (dans notre exemple **int**); en d'autres termes elle permet de construire un tableau de la taille désirée fixée en cours de traitement.

Remarques:

- **void** ne peut pas être utilisé comme type pour l'opérateur **new**.
- La zone mémoire alouée n'est pas initialisée!

Lorsque nous n'aurons plus besoin de l'objet en question nous restituerons au système la place mémoire qu'il utilisait, ceci par l'intermédiaire de l'opérateur **delete** qui lui aussi peut prendre 2 formes.

Première forme:

```
delete pointeur
```

Avec *pointeur* la valeur fournie par l'opérateur **new** sous sa première forme.

Deuxième forme:

```
delete [ ] pointeur
```

Avec *pointeur* la valeur fournie par l'opérateur **new** sous sa deuxième forme.

Pour les 2 **delete,** le pointeur garde (malheureusement !) sa valeur, il n'est pas mis à NULL!

Note:

Il est impératif d'utiliser la forme du **delete** correspondant au **new** utilisé pour la création de l'objet. De plus il ne faudra en aucun cas utiliser un **delete** avec une adresse fournie par une autre forme d'allocation de la mémoire (*malloc*, *calloc* valable en C et C++) que nous introduirons par la suite.

Prenons à titre d'exemple une nouvelle forme du crible d'Eratosthène:

```
/*
   Programme de calcul des nombres premiers par la methode du
   crible d'Eratosthene. Montre la creation de tableaux dynamiques
   ERATOSTHENE_ALLOCATION_NEW
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Definition des prototypes */
void afficher ( bool *, int );
void cribler  ( bool *, int );


int main ( )
{
  int maximum,     // Taille du tableau a reserver
      limite;      // Limite du calcul fixee par l'utilisateur
  /* Pointeurs sur le tableau du crible */
  bool *pt, *ptt;

  cout << "Crible d'Eratosthene\n\n";
  do   /* Tant que la donnee n'est pas valide */
  {
    cout << " Donnez la limite de calcul ( 3 au minimum ): ";
    cin >> limite;
  } while ( limite < 3 );

  maximum = ( limite - 3 ) / 2 + 1;
  /* Allocation dynamique et sauvegarde du pointeur */
  ptt = pt = new bool [ maximum ];

  /* Initialiser le crible, a priori tout est premier */
  while ( ptt != pt + maximum )
            *ptt++ = true;
  cribler ( pt, maximum );    // Determiner les nombres premiers
  afficher ( pt, maximum );   // Afficher les resultats

  delete [] pt;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

```
/* Sous-programme pour afficher les resultats */
/* Le tableau et la limite sont passes en parametres */
void afficher ( const bool *crible, int limite )
{
  cons int nbParLigne = 8;
  int   nbVal = 3; // Compteur de valeurs par ligne
  /* Traiter les 2 premieres valeurs comme cas particulier */
  cout << "\n\n\t1\t2";
  /* Traiter tous les elements du tableau */
  for ( register int i = 0; i < limite; i++ )
    if ( *( crible + i ) ) // Si c'est un nombre premier...
    {
      cout << '\t' << 2 * i + 3;   // ... l'afficher
      /* Passer a la ligne si necessaire */
      if ( ( nbVal++ % nbParLigne) == 0 ) cout << endl;
    }
} /* afficher */

/* Sous-programme realisant le crible a proprement parler */
/* Le tableau et la limite sont passes en parametres      */
void cribler ( bool *crible, int limite )
{
  /* Traiter tout le tableau */
  for ( register int i = 0; i < limite; i++ )
    if ( *( crible + i ) )    // Si c'est un nombre premier...
      /* ... enlever tous ses multiples */
      for ( register int k = 3*(i+1); k < limite; k += 2*i+3 )
            *(crible + k) = false;
} /* cribler */
```

Un problème peut se poser lors de la création de variables dynamiques si la mémoire disponible s'avère insuffisante. Dans ce cas, une exception *bad_alloc* est levée. Pour l'instant, comme nous ne savons toujours pas traiter ces exceptions, le programme s'arrêtera brutalement.

Nous allons introduire maintenant 2 variantes permettant de résoudre ce problème, mais elles aussi font appel à des notions que nous développerons plus en profondeur par la suite.

- Premièrement l'opérateur **new** peut se compléter de la manière suivante:

new (nothrow) type
new (nothrow) type [taille]

Sous cette forme, nous demandons que l'exception *bad_alloc* ne se propage pas, ce qui a comme conséquence qu'en cas d'impossibilité d'allouer la mémoire nécessaire, un pointeur *NULL* sera livré en retour. Nous pouvons alors facilement tester ce résultat et prendre des décisions en conséquence:

```
if ( pt != NULL )...
```

Nous vous donnons ci-dessous une nouvelle version adaptée à cette possibilité du programme principal pour notre crible d'Eratosthène; les fonctions elles ne changent pas!

```
/*
   Programme de calcul des nombres premiers par la methode du
   crible d'Eratosthene. Montre la creation de tableaux dynamiques
   et une des formes de gestion des erreurs d'allocation!!!!
   ERATOSTHENE_ALLOCATION_NEWB
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Definition des prototypes */
void afficher ( bool *pt, int limite );
void cribler  ( bool *pt, int );


int main ( )
{
  int maximum,     // Taille du tableau a reserver
      limite;      // Limite du calcul fixee par l'utilisateur
  /* Pointeurs sur le tableau du crible */
  bool *pt, *ptt;

  cout << "Crible d'Eratosthene\n\n";
  do   /* Tant que la donnee n'est pas valide */
  {
    cout << " Donnez la limite de calcul ( 3 au minimum ): ";
    cin >> limite;
  } while ( limite < 3 );

  maximum = ( limite - 3 ) / 2 + 1;
  /* Allocation dynamique et sauvegarde du pointeur */
  ptt = pt = new (nothrow) bool [ maximum ];
```

```
    // Test de la validité du pointeur
    if ( pt != NULL )
    {
      /* Initialiser le crible, a priori tout est premier */
      while ( ptt != pt + maximum )
        *ptt++ = true;
      cribler ( pt, maximum );  // Determiner les nombres premiers
      afficher ( pt, maximum ); // Afficher les resultats

      delete [] pt;
    }
    else  // Signaler le probleme
      cout << "\n\nMemoire insuffisante!!!\n";

    cout << "\nFin du programme...";
    system ( "pause" );
    return EXIT_SUCCESS;
}
```

La deuxième possibilité pour traiter autrement l'exception *bad_alloc* consiste à fixer une fonction appelée en cas de problème. Pour obtenir cet effet, il nous faut tout d'abord ajouter la directive d'inclusion:

```
#include <new>
```

Il faut aussi développer la fonction appelée en cas de problème, par exemple:

```
void gestionMemoire ( )...
```

Finalement, avant d'utiliser l'opérateur **new** nous spécifions que cette fonction doit être appelée si nécessaire. Ceci se réalise par l'intermédiaire de la fonction *set_new_handler* à laquelle nous transmettons en paramètre notre propre fonction de traitement, soit:

```
set_new_handler ( gestionMemoire );
```

Si tout se passe normalement (pas de problème de place mémoire) ces opérations n'ont aucune incidence sur le déroulement du programme.

Voilà, nous pouvons encore une fois donner une nouvelle version du programme principal pour le crible d'Eratosthène et de la fonction de gestion des erreurs, mais il faut bien reconnaître qu'elle ne fait ici que signaler le problème et arrêter brutalement l'application.

Voici cette version. A nouveau nous ne fournissons pas les fonctions *afficher* et *cribler* qui ne changent pas.

```
/*
   Programme de calcul des nombres premiers par la methode du
   crible d'Eratosthene. Montre la creation de tableaux dynamiques
   et une des formes de gestion des erreurs d'allocation!!!!
   ERATOSTHENE_ALLOCATION_NEWC
*/
#include <cstdlib>
#include <iostream>
#include <new>
using namespace std;

/* Definition des prototypes */
void afficher ( bool *pt, int limite );
void cribler  ( bool *pt, int );
void gestionMemoire ( );

int main ( )
{
  unsigned long int maximum,  // Taille du tableau a reserver
       limite;     // Limite du calcul fixee par l'utilisateur
  /* Pointeurs sur le tableau du crible */
  bool *pt, *ptt;

  cout << "Crible d'Eratosthene\n\n";

  /* Preparation de la gestion de la memoire en cas d'erreur */
  set_new_handler ( gestionMemoire );

  do   /* Tant que la donnee n'est pas valide */
  {
    cout << " Donnez la limite de calcul ( 3 au minimum ): ";
    cin >> limite;
  } while ( limite < 3 );

  maximum = ( limite - 3 ) / 2 + 1;
  /* Allocation dynamique et sauvegarde du pointeur */
  ptt = pt = new bool [ maximum ];

  /* Initialiser le crible, a priori tout est premier */
  while ( ptt != pt + maximum )
    *ptt++ = true;
  cribler ( pt, maximum );    // Determiner les nombres premiers
  afficher ( pt, maximum );   // Afficher les resultats

  delete [] pt;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

```
/* Gestion de la memoire en cas d'erreur,
    Pour l'instant nous ne faisons pas grand chose!!
*/
void gestionMemoire ( )
{
  cout << "\n\nMemoire insuffisante!!!\n";
  cout << "\nFin du programme...";
  system ( "pause" );
  exit ( -1 );
}
```

Fonctions de la bibliothèque C

Nous l'avons dit, les opérateurs **new** et **delete** n'existent pas en C; toutefois nous pouvons obtenir un résultat assez semblable en utilisant les fonctions spécifiques de la bibliothèque en incluant le fichier *stdlib.h (cstdlib)*. Bien que cela soit plutôt déconseillé, nous pouvons continuer en C++ à travailler de cette manière en incluant alors *cstdlib*, opération que nous faisons en principe de toute façon!

La partie gestion dynamique de la mémoire dans cette bibliothèque se compose de 4 sous-programmes. Il s'agit de:

```
void *malloc ( size_t taille );
```

Qui alloue un bloc mémoire de *taille* octets; l'adresse de ce bloc est livrée sous forme d'un pointeur neutre, comme résultat de la fonction. Le type *size_t* définit dans *cstdlib* correspond à un entier non signé. Si la place mémoire n'a pas pu être allouée, la fonction livre comme résultat *NULL*. La zone allouée contient n'importe quoi comme information (ce qui se trouve à ce moment-là en mémoire à l'endroit alloué!).

```
void *calloc ( size_t nb_elem, size_t taille );
```

Alloue un bloc de mémoire de *nb_elem* éléments, chacun étant de grandeur *taille* octets. Elle est généralement utilisée pour allouer des tableaux:

```
pt = calloc ( nb_elem, taille );
```

qui toutefois peut être remplacée par:

```
pt = malloc ( nb_elem * taille );
```

A nouveau si la mémoire ne peut pas être allouée, on obtient en retour *NULL*.

La zone allouée contient des octets mis à zéros; attention, ce n'est pas forcément une valeur compatible avec le type pointé!

```
void *realloc ( void *ptr, size_t taille );
```

Permet de modifier la taille d'un bloc mémoire déjà alloué par l'un des autres sous-programmes (*calloc*, *malloc*).

ptr correspond au pointeur contenant l'ancienne adresse de début du bloc.

taille est la nouvelle grandeur désirée du bloc. Si cette taille est plus petite que l'ancienne, la fin du bloc est perdue pour le programme en cours, mais récupérable par le système. Par contre, si la nouvelle taille s'avère plus grande que l'ancienne, il se peut que tout le bloc soit transféré ailleurs en mémoire, mais alors l'ancien contenu est conservé par recopie automatique. Toutefois, cela signifie bien que l'ancienne adresse contenue dans *ptr* peut ne plus être significative. On ne doit donc pas l'utiliser dans la suite du programme.

Comme toujours, si l'opération s'avère impossible, on nous retourne *NULL*.

Notez que:

```
pt = realloc ( NULL, taille );
```

revient au même que:

```
pt = malloc ( taille );
```

realloc s'utilise pour initialiser un processus où l'on sera amené dans une boucle à redimensionner la zone, car généralement on utilisera une structure de la forme:

```
…
int *pt = NULL;
…
// Boucle
   …
   pt = realloc ( pt, taille );
```

Attention: les fonctions *malloc* et *calloc* et *realloc* livrent un pointeur neutre. Si C permet d'affecter une telle valeur à une variable d'un type pointeur donné, ce n'est pas le cas de C++. Nous devons convertir explicitement (cast) dans le type destination désiré, exemple:

```
pt = (int *) malloc ( nb_elem * taille );
```

```
void free ( void *ptr );
```

Permet de restituer la mémoire préalablement allouée par l'une des *fonctions malloc, calloc, realloc*.

Attention: l'adresse dans *ptr* doit être telle qu'elle a été livrée par l'une de ces fonctions. Notez que l'on arrive au même résultat en transmettant à *realloc* une taille 0.

Attention aussi aux dangers classiques dans ce genre de traitement: ne pas utiliser *ptr* après l'opération *free*, il serait raisonnable de lui affecter *NULL* immédiatement après l'opération; ne pas tenter non plus de libérer une deuxième fois la zone, le comportement deviendrait imprévisible.

> **Attention**: nous rappelons qu'il ne faut pas utiliser le **delete** de C++ pour restituer la mémoire allouée avec: *malloc*, *calloc* ou *realloc*; ni utiliser la fonction *free* pour restituer de la mémoire réservée par l'opérateur **new**!

Reprenons, encore une fois, pour illustrer ces points, notre programme de calcul des nombres premiers et laissons à nouveau l'utilisateur fixer la limite de son calcul (les fonctions *cribler* et *afficher* ne changeant toujours pas, nous ne les redonnons pas ici!):

```
/*
   Programme de calcul des nombres premiers par la methode du
   crible d'Eratosthene
     ERATOSTHENE_ALLOCATION_DYNAMIQUE
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Definition des prototypes */
void afficher ( bool *pt, int limite );
void cribler  ( bool *pt, int limite );


int main ( )
{
  int maximum,    // Taille du tableau a reserver
      limite;     // Limite du calcul fixee par l'utilisateur
  /* Pointeurs sur le tableau du crible */
  bool *pt, *ptt;

  cout << "Crible d'Eratosthene\n\n";
  do   /* Tant que la donnee n'est pas valide */
  {
    cout << " Donnez la limite de calcul ( 3 au minimum ): ";
    cin >> limite;
  } while ( limite < 3 );

  maximum = ( limite - 3 ) / 2 + 1;
  /* Allocation dynamique et sauvegarde du pointeur */
  ptt = pt = ( bool * ) calloc ( maximum, sizeof (bool) );
```

```
  /* Initialiser le crible, a priori tout est premier */
  while ( ptt != pt + maximum )
    *ptt++ = true;
  cribler ( pt, maximum );   // Determiner les nombres premiers
  afficher ( pt, maximum );  // Afficher les resultats

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

C'est sous cette forme que l'on utilise le plus souvent les variables dynamiques en C!

❑ *Résoudre le problème des tableaux variables en paramètre*

Comme nous avions pu le constater il n'était pas possible d'écrire des sous-programmes traitant des tableaux de taille variable à plus d'une dimension en passant les tableaux sous leur forme usuelle de paramètre[1]. Moyennant quelques acrobaties, tant du côté du sous-programme que de l'appelant, nous pouvons détourner ce problème.

Tout d'abord le type du paramètre se définit comme un pointeur sur un objet du type des éléments de la matrice, par exemple si nous manipulons une matrice d'entiers:

```
int * matrice;
```

Nous devons continuer à transmettre le nombre d'éléments de chaque indice de la matrice. Ainsi, si nous voulons réaliser une procédure générale pour afficher les éléments de type **int** d'une matrice rectangulaire de taille quelconque, son prototype prend une forme du genre:

```
void ecrire ( const int *matrice, int m, int n );
```

[1] Ceci avait été rendu possible en C à partir de la norme C99 et ses tableaux de taille variable!

Avant d'approfondir ce qui peut se passer dans le corps du sous-programme, regardons ce que doit faire l'appelant. Si par exemple il a déclaré une matrice sous la forme:

```
int  m [2] [3] = { {1, 2, 3}, {4, 5, 6} };
```

il appelle la fonction *ecrire* par:

```
ecrire ( (int *) m, 2, 3 );
```

Comme par le passé, nous devons transmettre l'adresse du premier élément du tableau, ce que fait effectivement le nom du tableau lui-même (nous avons déjà utilisé cette possibilité), mais si nous le faisons ici, le compilateur signalera (ou devrait signaler!) que le type du paramètre effectif ne correspond pas à celui du paramètre formel. Pour cette raison, nous forçons la conversion du type pointeur par l'opérateur cast:

```
(int *) m
```

Nous pouvons aussi transmettre l'adresse du premier élément en le désignant explicitement, ce qui aurait donné l'appel:

```
ecrire ( &m [0][0], 2, 3 );
```

ou encore:

```
ecrire ( m [0], 2, 3 );
```

Voilà, revenons maintenant à la manière de travailler dans le sous-programme. Ce qu'il faut retenir a priori: nous devons calculer où se trouve en mémoire l'élément du tableau qui nous intéresse et ceci à partir de l'adresse du début du tableau (transmise en paramètre) et des valeurs des dimensions. Pour cela n'oubliez pas que du point de vue du rangement en mémoire des éléments du tableau, le dernier indice varie le plus rapidement. Ainsi pour la déclaration de *m* ci-dessus, nous trouvons en mémoire et dans l'ordre:

```
m[0][0], m[0][1], m[0][2], m[1][0], m[1][1], m[1][2]
```

Basons-nous pour illustrer les choses sur une matrice à 2 dimensions, bien que rappelons-le, cela n'existe pas puisqu'il s'agit d'un tableau de tableaux (nous continuerons toutefois à parler d'un tableau à 2 dimensions, 3 dimensions, etc.):

```
#define m …
#define n …
…
int tab[m][n];
```

Pour tout *i* (0<=i<m), écrire *tab [i]* est tout à fait correct et désigne un tableau de *n* entiers; *tab [i]* est donc un pointeur sur un entier!

Pour accéder à l'élément tab [i] [j], sous la forme pointeur:

```
*( (int *)tab + i*n + j )
```

C'est-à-dire qu'à l'adresse de base du tableau (*tab*), on additionne *i* (le numéro de ligne désiré) fois le nombre de colonnes du tableau (ceci nous fournit l'adresse du premier élément de la $i^{ème}$ ligne, n'oubliez pas que la borne inférieure est 0); il nous reste à ajouter *j* (la position dans cette ligne de la colonne désirée).

Si nous disposons déjà de l'adresse du début du tableau dans un pointeur sur un entier, la conversion (**int***) ci-dessus n'est pas nécessaire.

Le principe fonctionne quelles que soient les dimensions de la matrice et se généralise donc pour des tableaux à plus de 2 dimensions.

Le programme ci-dessous illustre ce que nous venons de décrire et met en évidence le fait que dans des situations connues précises, nous pouvons souvent simplifier les choses. Regardez ce programme, nous donnerons quelques explications complémentaires après:

```
/*
  Module de traitement de matrices, mettant a disposition:
    - Ecrire une matrice.
    - Additionner 2 matrices.
    - Additionner une valeur a tous les elements d'une colonne.
  Fichier des prototypes: matrices.h
*/

/* Afficher une matrice */
void ecrire ( const int *matrice, int m, int n );

/* Additionner deux matrices dans une troisieme */
void add ( const int *mat1, const int *mat2, int *mat3, int m, int n );
/* Additionner une valeur a tous les elements d'une colonne */
void add_colonne (  int *matrice, int m, int n,
                    int colonne, int valeur );
```

```
/*
  Module de traitement de matrices, mettant a disposition:
    - Ecrire une matrices.
    - Additionner 2 matrices.
    - Additionner une valeur a tous les elements d'une colonne.
  Fichier : matrices.cpp
*/
#include <iostream>
using namespace std;
#include "matrices.h"
/* Afficher une matrice */
void ecrire ( const int *matrice, int m, int n )
{
  /* Traiter tous les elements de la matrice */
  for ( int i = 0; i < m; i++ )
  { for ( int j = 0; j < n; j++ )
      cout << '\t' << *matrice++;
    cout << endl;
  }
}



/* Additionner deux matrices dans une troisieme */
void add ( const int *mat1, const int *mat2, int *mat3, int m, int n )
{
  int * const fin = mat3 + m * n;
  /* Traiter tous les elements de la matrice */
  while ( mat3 != fin )
    *mat3++ = *mat1++ + *mat2++;
}

/* Additionner une valeur a tous les elements d'une colonne */
void add_colonne (  int *matrice, int m, int n,
                    int colonne, int valeur )
{
  /* Traiter tous les elements de la matrice */
  for ( int i = 0 ; i < m; i++ )
    *(matrice + i * n + colonne ) += valeur;
}
```

```
#include <cstdlib>
#include <iostream>
using namespace std;
#include "matrices.h"
/*
  Programme de test des sous-programmes de manipulation de matrices
  TEST_MATRICES
*/
int main ( )
{
  int  m1 [2] [3] = { {1, 2, 3}, {4, 5, 6} };
  int  m2 [2] [3] = { {6, 5, 4}, {3, 2, 1} };
  int  m3 [2] [3];

  cout << " Manipulation de matrices\n\n";
  cout << "m1:" << endl;
  ecrire ( (int *) m1, 2, 3 );
  cout << "\nm2:" << endl;
  ecrire ( (int *) m2, 2, 3 );
  add ( (int *) m1, (int *) m2, (int *) m3, 2, 3 );
  cout << "\nm3 = m1 + m2:" << endl;
  ecrire ( (int *) m3, 2, 3 );
  add_colonne ( (int *) m3,  2, 3, 1, -7 );
  cout << "\nm3, colonne1 - 7:" << endl;
  ecrire ( (int *) m3, 2, 3 );

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution de ce programme donne comme résultat:

```
    Manipulation de matrices

    m1:
            1       2       3
            4       5       6

    m2:
            6       5       4
            3       2       1

    m3 = m1 + m2:
            7       7       7
            7       7       7
    m3, colonne1 - 7:
            7       0       7
            7       0       7

    Fin du programme...Appuyez sur une touche pour continuer...
```

Rappelons tout d'abord que nous n'avons pas la possibilité de contrôler dans les sous-programmes ce que nous transmet réellement l'appelant. Ainsi, si l'un des tableaux n'est pas de taille m*n, le résultat des opérations apparaîtra totalement aléatoire!

Pour la fonction *ecrire*, il suffit d'afficher les éléments les uns après les autres, d'où l'incrémentation à chaque fois de l'adresse de base du tableau (en réalité de la copie de cette adresse qui a été transmise en paramètre!); nous avons toutefois gardé les 2 boucles afin de gérer proprement le passage à la ligne.

Pour l'addition (*add*) peu importe la structure de lignes et de colonnes, d'où une seule boucle pour parcourir tous les éléments des tableaux et réaliser l'addition entre ceux de même position. Dans la boucle, chaque copie d'adresse de début de tableau est incrémentée pour passer à l'élément suivant. La constante pointeur *fin* permet de fixer la limite supérieure du tableau en mémoire et sert de test de sortie du traitement.

Le sous-programme *add_colonne* montre que l'on peut traiter aussi bien une colonne spécifique qu'une ligne; la multiplication "i*n" nous permet de passer d'une ligne à l'autre, et l'addition de *colonne*, de nous positionner à la bonne colonne dans la ligne en question.

Relevez aussi sur l'exemple le découpage réalisé:

- Un fichier ".h" contenant les prototypes des fonctions et éventuellement d'autres définitions liées.
- Un fichier ".cpp" contenant les corps effectifs de ces fonctions, ce fichier comportant un *include* du fichier ".h" qui précède, ce qui permet au compilateur de réaliser des contrôles de validité.
- Un fichier ".cpp" contenant le programme principal.

Dans une application réelle il y aura certainement plusieurs fichiers ".h" et ".cpp", mais c'est toujours sur ce même principe que nous devrions développer nos applications.

Notez aussi la forme de l'include pour notre fichier ".h":

```
#include "matrices.h"
```

La présence des "…" à la place de <…> indique au compilateur qu'il doit a priori rechercher le fichier en question dans le répertoire courant par défaut et non pas dans les répertoires spécifiques de la bibliothèque.

Lorsque nous transmettons un tableau en paramètre à un sous-programme du genre:

```
void  sp ( float tab [ ] );
```

Le contenu du tableau et le pointeur (en fait sa copie locale) sont modifiables dans le sous-programme.

```
void  sp ( const float tab [ ] );
```

Ci-dessus le contenu du tableau ne sera jamais modifiable, par contre la copie du pointeur oui.

Illustrons les précautions que l'on peut prendre lorsque l'on transmet le paramètre explicitement sous forme d'un pointeur:

```
void  sp ( float * tab );
```

Ci-dessus, aucune précaution prise: on peut modifier le contenu du tableau et le pointeur.

```
void  sp ( const float * tab );
```

Ci-dessus on ne peut pas modifier le contenu du tableau, par contre le pointeur oui.

```
void  sp ( float * const tab );
```

Ci-dessus on ne peut pas modifier le pointeur, par contre le contenu du tableau oui.

```
void  sp ( const float * const tab );
```

Finalement ci-dessus le contenu du tableau et celui du pointeur ne sont pas modifiables.

Faites attention lorsqu'une fonction livre un pointeur comme résultat, que ce pointeur ne contienne pas l'adresse d'un objet qui n'existe plus après le retour à l'appelant, le comportement du programme serait alors totalement imprévisible.

Exemple de ce qu'il **ne** faut **pas** faire:

```
char  * sp ( )
{ char tab [10];
  …
  return tab;
}
```

Il y a d'autres manières de se mettre dans une situation aussi dangereuse, mais n'exagérons pas avec les mauvais exemples!

Quelques notations complémentaires:

- Pour accéder à un élément d'un tableau à 2 dimensions, la notation: *tab [i] [j]* peut aussi s'écrire: **(*(tab + i) + j)*, mais la lisibilité est certainement moins bonne.

- Pour une fonction qui aurait un tableau à 2 dimensions en paramètre, du genre:

```
void sp ( int tab [3] [5], ... );
```

rappelons d'abord que la première dimension n'est pas nécessaire et donc que ce prototype peut aussi s'écrire

```
void sp ( int tab [] [5], ... );
```

mais il s'écrit aussi, sous forme de pointeur:

```
void sp ( int *pt [], ... );
```

❑ *Réflexion !*

Prenons à titre d'exemple l'instruction:

```
for ( i = 0; i < limite; i++ ) tab [ i ] = val;
```

en admettant que les déclarations adéquates ont été faites!

Comme nous l'avons déjà vu, cette instruction peut aussi s'écrire:

```
for ( i = 0; i < limite; i++ ) *( tab + i ) = val;
```

Si l'on prend un pointeur partant du début du tableau:

```
for ( pt = tab, i = 0; i < limite; i++ ) *pt++ = val;
```

On peut maintenant supprimer *i*:

```
for ( pt = tab; pt != tab + limite; ) *pt++ = val;
```

Ceci peut finalement se mettre sous la forme d'une boucle **while** avec initialisation préalable:

```
pt  = tab; /* Initialisation prealable des invariants */
fin = tab + limite;
while ( pt != fin )  /* La boucle */
   *pt++ = val;
```

Caractères - chaînes de caractères - string

Les chaînes de caractères

Nous nous retrouvons face au même problème qu'au chapitre qui précède, à savoir: aborder les chaînes de manière classique, compatible C/C++ ou utiliser les éléments de bibliothèque C++. Toutefois nous avons une compatibilité presque totale entre les 2 formes. Nous commençons par présenter les chaînes traditionnelles "à la C" bien que cela ne soit pas la forme que nous vous conseillons d'utiliser au final, mais cette démarche nous permet dans la deuxième partie de mettre en évidence les liens entre les 2 formes.

Chaînes traditionnelles C

Dans cette forme traditionnelle les chaînes de caractères ne sont rien d'autre que des tableaux de caractères dont le dernier effectivement (logiquement) utilisé est suivi par un caractère nul ('\0') marquant la fin de la chaîne logique.

Tout ce que nous avons dit sur les tableaux reste donc valable pour cette forme de chaînes!

Exemple de déclaration:

```
char ligne [81];
```

ligne correspond à une chaîne qui peut comporter 80 caractères effectifs, plus le nul de fin de chaîne. Puisqu'il s'agit de tableaux de caractères, nous pouvons l'initialiser de la manière suivante:

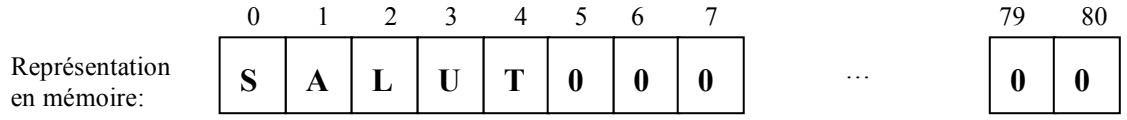
```
char ligne [] = { 'S','A','L','U','T', '\0' };
```

Représentation en mémoire:

0	1	2	3	4	5
S	**A**	**L**	**U**	**T**	**0**

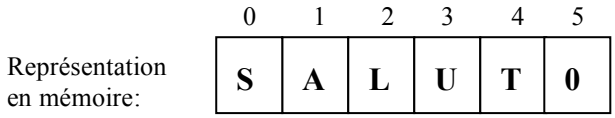
Mais si nous voulons une chaîne potentiellement plus grande, car le texte peut changer en cours de traitement:

```
char ligne [81] = { 'S','A','L','U','T', '\0' };
```



Cette écriture peu pratique se simplifie en utilisant la notion de constante chaîne de caractères:

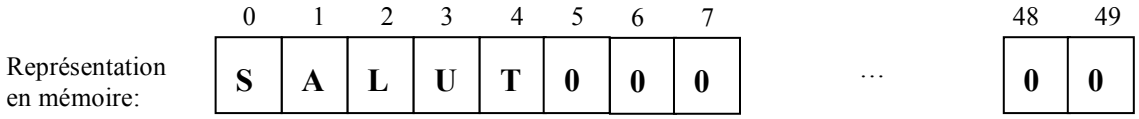
```
char ligne [] = "SALUT";
```



Ici le tableau est de 6 éléments car le compilateur ajoute automatiquement un caractère nul en fin de chaîne. La constante chaîne de caractères elle-même se termine par ce nul.

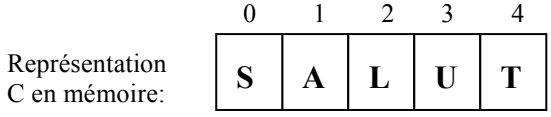
Nous pouvons bien sûr aussi écrire:

```
char ligne [50] = "SALUT";
```



Attention toutefois au cas particulier:

```
char ligne [5] = "SALUT";
```



Ici et **en C uniquement** le caractère nul n'est pas ajouté à la fin de la chaîne, car nous lui avons fixé une taille correspondant exactement à la longueur de la chaîne d'initialisation. La manipulation de cette chaîne *ligne* ne pourra pas se faire correctement avec les sous-programmes de la bibliothèque! En **C++** cette même déclaration provoque une erreur de compilation!

Attention:

L'affectation de tableaux n'étant pas permise, nous ne pouvons pas, dans les instructions d'un programme, écrire:

```
ligne = "SALUT"; // Ceci est faux
```

Vous pouvez sans problème lire et écrire des chaînes, la gestion de l'octet nul à la fin de la chaîne logique se fait automatiquement, bien malheureusement avec l'éventuel débordement (chaîne lue trop longue par rapport à la définition de la variable) non contrôlé.

Voici un tout petit exemple de programme:

```
/*
  Lecture/ecriture de chaines classiques C
  CHAINESOLD1
*/
#include <cstdlib>
#include <iostream>
using namespace std;
#define LONGUEURMAX 3
int  main ( )
{
  char prenom [LONGUEURMAX];
  cout << "Votre prenom: ";
  cin >> prenom;
  cout << "Bonjour " << prenom << endl;
  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Voilà, nous savons presque tout sur les chaînes ancienne forme; leur richesse découle essentiellement de celle des bibliothèques associées, aussi allons-nous donner quelques indications dans ce sens, sans forcément aller dans tous les détails.

Une chaîne étant a priori constituée de caractères, commençons par une information sur ceux-ci.

Catégories de caractères

Rappelons que le jeu de caractères utilisé n'est pas défini par la norme; de plus, mais nous n'en parlerons pas ici, certaines caractéristiques peuvent dépendre de la localisation.

Il existe une série de fonctions, définies dans *cctype*[1] qui permettent de savoir à quelle catégorie appartient un caractère; elles ont toutes la forme:

```
int isxxx ( int c );
```

Où xxx représente le nom de la catégorie; elles ont en paramètre d'entrée le caractère dont on désire savoir s'il appartient à la catégorie, ne vous inquiétez pas du type **int** de ce paramètre, la raison en est historique et il y a promotion automatique du char en int.

Elles livrent comme résultat: vrai ou faux!

En voici la liste:

```
int isalnum ( int c );
```

Vrai si le caractère correspond à une lettre ou un chiffre et faux dans le cas contraire.

```
int isalpha ( int c );
```

Vrai si le caractère correspond à une lettre majuscule ou minuscule et faux dans le cas contraire.

```
int iscntrl ( int c );
```

Vrai si c'est un caractère de contrôle (code 0 à 31 et 127) et faux dans le cas contraire.

```
int isdigit ( int c );
```

Vrai si le caractère correspond à un chiffre et faux dans le cas contraire.

```
int isgraph ( int c );
```

Vrai si le caractère est affichable et non blanc (code 33 à 126) et faux dans le cas contraire.

[1] Pour un programme C, il faudra inclure *ctype.h*!

```
int islower ( int c );
```

Vrai si le caractère correspond à une lettre minuscule et faux dans le cas contraire.

```
int isprint ( int c );
```

Vrai si le caractère est affichable (code 32 à 126) et faux dans le cas contraire.

```
int ispunct ( int c );
```

Vrai si le caractère est différent d'une lettre, d'un chiffre ou d'un caractère de contrôle et faux dans le cas contraire.

```
int isspace ( int c );
```

Vrai si le caractère correspond à un espace, une tabulation (verticale ou horizontale), une fin de ligne ou un retour chariot et faux dans le cas contraire.

```
int isupper ( int c );
```

Vrai si le caractère correspond à une lettre majuscule et faux dans le cas contraire.

```
int isxdigit ( int c );
```

Vrai si le caractère correspond à un chiffre hexadécimal et faux dans le cas contraire.

De plus 2 fonctions permettent de transformer respectivement une lettre majuscule en minuscule et une minuscule en majuscule:

```
int tolower ( int c );
```

Livre comme résultat le caractère *c* converti en minuscule si c'était une majuscule et sinon le caractère inchangé.

```
int toupper ( int c );
```

Livre comme résultat le caractère *c* converti en majuscule si c'était une minuscule et sinon le caractère inchangé.

Ces fonctions sont suffisamment simples pour que nous ne développions pas davantage, mais n'oubliez pas qu'elles peuvent poser des problèmes de par la définition même des catégories (et par exemple avec les lettres accentuées!).

Donnons simplement un petit exemple d'utilisation:

```
/* [1]
  Compte les lettres, les chiffres les signes de ponctuation
    et les espaces dans une chaine
  CARACTERES
*/
#include <cstdlib>
#include <iostream>
#include <cctype>
using namespace std;

int main ( )
{
  char caractere;
  /* Les compteurs: */
  unsigned int lettres = 0, chiffres = 0, ponctuations = 0,
               espaces = 0;

  /* Demander la chaine */
  cout << "Donnez la chaine a traiter:" << endl;

  /* Traiter tous les caracteres */
  while ( ( caractere  = cin.get () ) != '\n' )
  {
    /* Mettre a jour les compteurs: */
    if ( isalpha ( caractere ) ) lettres++;
    else if ( isdigit ( caractere ) ) chiffres++;
    else if ( ispunct ( caractere ) ) ponctuations++;
    else if ( isspace ( caractere ) ) espaces++;
  }

  /* Afficher les résultats */
  cout << "\n\nVotre ligne contient:\n\n";
  cout << lettres <<      '\t' << "lettres"                 << endl;
  cout << chiffres <<     '\t' << "chiffres"                << endl;
  cout << ponctuations << '\t' << "signes de ponctuations" << endl;
  cout << espaces <<      '\t' << "espaces"                 << endl;

  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

[1] Pour un programme C, en plus des entrées/sorties pour lesquelles il faudrait utiliser par exemple des *printf* et des *scanf* (comme nous l'avions signalé dès le début de ce cours), l'include <*cctype*> doit être remplacé par un include <*ctype.h*>

Notez sur cet exemple la première utilisation que nous faisons de la méthode *get* sur le flux *cin*. Elle ne possède pas de paramètre et livre en retour le prochain caractère sur ce flux d'entrée.

Relevez aussi l'utilisation que nous en avons faite dans la boucle **while**:

```
while ( ( caractere  = cin.get () ) != '\n' )
```

On affecte le résultat de la lecture à la variable caractere dans la parenthèse interne, puis on compare le résultat de cette affectation pour savoir s'il s'agit d'une fin de ligne, donc pour nous d'une fin de traitement.

La bibliothèque cstring[1]

Rappels de base

Nous l'avons déjà dit, le traitement des chaînes de caractères implique généralement l'utilisation des fonctions de la bibliothèque. Les définitions relatives aux chaînes se trouvent dans *cstring*[2] (à ne pas confondre avec *string* dont nous reparlerons dans la deuxième partie) qu'il vous faut donc inclure dans le programme.

> N'oubliez pas que tout repose sur le fait que les conventions sont systématiquement respectées, à savoir que chaque chaîne se termine par un caractère nul, si ce n'est pas le cas, le comportement des fonctions devient totalement aléatoire!

Rappel: nous ne pouvons pas, dans les instructions du programme affecter un tableau (et ces chaînes de caractères sont des tableaux), car la notion de tableau en tant que telle n'existe pas, il s'agit de l'adresse, donc d'un pointeur, et le nom d'un tableau est une constante pointeur qui ne peut pas changer de valeur.

[1] La norme C99 a introduit le mot réservé **restrict** pour un paramètre pointeur indiquant par-là qu'il s'agit du seul pointeur désignant cet objet. Ceci n'existe pas en C++ et nous ne l'utiliserons pas dans les en-têtes de fonctions que nous vous présentons.

[2] En C, vous utiliserez un #include <string.h>

Longueur

Pour obtenir la longueur courante d'une chaîne, c'est-à-dire le nombre de caractère depuis le début de la chaîne jusqu'à celui qui précède le premier '\0':

```
size_t strlen ( const char *string );
```

Copie

Pour copier un tableau dans un autre (une chaîne dans une autre), il faudra soit recopier explicitement chaque élément, soit utiliser des fonctions de la bibliothèque, qui met à disposition plusieurs variantes dans ce but:

```
char *strcpy ( char * to, const char * from );
```

Recopie la chaîne *from* dans *to* et retourne un pointeur sur *to*.

Attention: si les chaînes source et destination se chevauchent, le comportement est indéterminé.

Utilisation normale:

```
strcpy ( ch1, ch2 );
```

Il faut que *ch1* possède une taille suffisamment grande pour contenir *ch2*, sinon d'autres données seront probablement détruites.

Cas particuliers d'utilisation:

- `strcpy ( ch1, "" );`

ch1 contient maintenant la chaîne vide!

- `strcpy ( ch1, strcpy ( ch2, ch3 ) );`

De par la valeur de retour de strcpy (l'adresse de la chaîne destination), l'effet de cet appel revient à recopier *ch3* dans *ch1* et *ch2*; il serait certainement plus propre et plus lisible de le faire par 2 appels explicites.

- `strcpy ( strcpy ( ch1, ch2 ), strcpy ( ch1, ch3 ) );`

Son effet est indéterminé car on ne sait pas (cela dépend de l'implémentation) quel paramètre est évalué en premier! Alors ne le faites pas!!!

Une autre fonction permet de faire le même travail en ajoutant une sécurité complémentaire en ne traitant éventuellement qu'une partie de la chaîne source:

```
char *strncpy ( char * to,
                const char * from, size_t size );
```

Les 2 premiers paramètres sont les mêmes que pour *strcpy* avec la même signification; par contre, la fonction possède un troisième paramètre de type *size_t*[1] qui limite le nombre de caractères traités, c'est-à-dire que la copie s'arrête si l'on rencontre une fin de chaîne ou si l'on a déjà recopié *size* caractères; mais attention, dans ce dernier cas, aucune indication de fin de chaîne n'est introduite dans la destination.

Il existe aussi des fonctions pouvant faire le même genre de travail, mais sans prendre en considération la notion de chaînes, en d'autres termes des fonctions capables de copier une zone de mémoire (des octets) dans une autre zone de mémoire.

Il s'agit de:

```
void *memcpy ( void * to,
               const void * from, size_t size );
```

Recopie de toute façon *size* octets de la zone *from* dans la zone *to*, quel que soit le contenu de ces octets (y compris des nuls!) et livre comme résultat un pointeur sur la zone *to*. A nouveau les zones source et destination ne doivent pas se recouvrir.

Par contre la fonction:

```
void *memmove ( void *to,
                const void *from, size_t size );
```

réalise la même opération, mais en plus la copie se passe correctement, même si les zones *from* et *to* se chevauchent. Par contre elle sera vraisemblablement moins performante que *memcpy*.

[1] *size_t*, défini dans la bibliothèque, représente un type entier non signé!

Concaténation

Pour concaténer 2 chaînes, on va disposer des 2 mêmes possibilités que pour la copie. La première se base uniquement sur la marque de fin de chaîne pour arrêter la copie:

```
char *strcat ( char * to,
               const char * from );
```

Concatène la chaîne *from* à la suite de la chaîne *to* et livre en retour un pointeur sur *to*.

La seconde concatène au plus un nombre maximum de caractères:

```
char *strncat ( char * to,
                const char * from, size_t size );
```

Elle concatène la chaîne *from* à la suite de la chaîne *to* et livre un pointeur sur *to*, mais au plus *size* caractères de la chaîne *from* sont copiés. Ici, contrairement à la copie, un caractère nul est toujours ajouté à la fin de la chaîne quelque soit le critère qui arrête la concaténation.

Notez que pour les 2 formes les chaînes ne doivent pas se recouvrir, sinon un comportement indéterminé peut se produire.

Comparaison

Nous savons déjà que les opérateurs de comparaisons ne sont pas utilisables sur les tableaux donc sur les chaînes. A nouveau la bibliothèque vient à notre aide!

```
int strcmp ( const char *s1, const char *s2 );
```

Compare les 2 chaînes *s1* et *s2* et retourne un résultat positif si *s1* > *s2*, nul si *s1* = *s2* et négatif *si s1* < *s2* (pour les comparaisons, les caractères sont considérés comme des **unsigned char** !).

```
int strncmp ( const char *s1, const char *s2, size_t size );
```

Compare au maximum sur *size* caractères les 2 chaînes *s1* et *s2* et retourne un résultat positif si *s1* > *s2*, nul si *s1* = *s2* et négatif si *s1* < *s2*.

Etant donné la nature même de la notion de comparaison de chaînes, il n'y a pas ici de problème de recouvrement entre la source et la destination.

La bibliothèque offre aussi la possibilité de faire des comparaisons sur des zones de mémoire, sans tenir compte de la notion de chaîne; plus précisément le caractère nul est traité comme n'importe quel autre octet de la zone, donc seule la longueur fait office d'arrêt:

```
int memcmp ( const void *s1, const void *s2, size_t size );
```

Compare au maximum sur *size* octets les 2 zones *s1* et *s2* et retourne un résultat positif si *s1* > *s2*, nul si *s1* = *s2* et négatif si *s1* < *s2*.

Il existe encore d'autres fonctions liées à ce problème, mais nous n'en parlerons pas ici!

❑ Recherche

Pour la recherche d'un motif dans une chaîne, de nombreuses possibilités existent.

Pour rechercher la présence d'un caractère dans une chaîne en partant de son début, nous disposons de:

```
char *strchr ( const char *string, int c );
```

Retourne un pointeur sur la première occurrence du "caractère" *c* dans la chaîne *string*, ou *NULL* si *c* n'est pas trouvé.

Ou en partant de la fin de la chaîne:

```
char *strrchr ( const char *string, int c );
```

Retourne un pointeur sur la dernière occurrence du "caractère" *c* dans la chaîne *string*, ou *NULL* si *c* n'est pas trouvé.

Notez que dans les 2 cas, comme bien souvent, le caractère se transmet sous la forme d'un paramètre de type **int** et non pas **char**!

Pour rechercher la première occurrence d'un caractère parmi un ensemble de caractères possibles:

```
char *strpbrk ( const char *string, const char *set );
```

Retourne un pointeur sur le premier caractère de la chaîne *string* qui appartient également à la chaîne *set*. Pour la compréhension, il faut voir le deuxième paramètre comme un ensemble de caractères, exemples:

```
strpbrk ( "Salut", "ok" );
```

Livre le pointeur *NULL* car ni 'o' ni 'k' ne se trouvent dans la chaîne "Salut"

```
strpbrk ( "Toto", "ok" );
```

Livre un pointeur sur le premier 'o' de la chaîne "Toto"

Comme bien souvent dans ce genre de situation, nous pouvons aussi rechercher une sous-chaîne dans une chaîne:

```
char *strstr ( const char *string, const char *substring );
```

Retourne un pointeur sur le début de la première occurrence de la chaîne *substring* dans la chaîne *string* et *NULL* si *substring* ne se trouve pas dans *string*.

Cas particulier: si la sous-chaîne recherchée est vide, la fonction livre la chaîne principale (**string*) comme résultat!

Bien qu'impliquant une recherche, les 2 sous-programmes suivant se présentent d'une manière un peu particulière et sont certainement moins utilisés que les précédents, quoique parfois bien utiles:

```
size_t strspn ( const char *string, const char *set );
```

Retourne la position du premier caractère de *string* qui n'apparaît pas dans *set*. Autrement dit, elle livre comme résultat la longueur du "segment initial" de la chaîne *string* constituée des caractères faisant partie de l'ensemble *set*.

Exemple: `strspn ( "1234ab567", "0123456789" )` livre 4 comme résultat.

```
size_t strcspn ( const char *string, const char *set );
```

Retourne la position du premier caractère de la chaîne *string* qui appartient également à la chaîne *set*. Autrement dit, elle livre comme résultat la longueur du "segment initial" de la chaîne *string* constituée uniquement des caractères ne faisant pas partie de l'ensemble *set*.

Exemple: `strcspn ( "1234ab567", "0123456789" )` livre 0 comme résultat.

Pour la compréhension de ces 2 fonctions, il faut interpréter le deuxième paramètre comme un ensemble de caractères.

Il existe aussi, indépendamment des chaînes, la possibilité de rechercher la valeur d'un octet spécifique dans une zone de mémoire (sans tenir compte d'une éventuelle fin de chaîne):

```
void *memchr ( const void *obj, int c, size_t size );
```

Retourne un pointeur sur la première occurrence du "caractère/octet" *c* dans les *size* premiers octets de la zone *obj*; si *c* n'est pas trouvé, retourne *NULL*.

Note: même si la valeur recherchée est transmise sous la forme d'un **int**, c'est bien un octet qui est recherché, alors attention aux valeurs transmises pour ce paramètre.

❑ Décompositions

Cette opération, bien que particulière, s'avère utile dans des problèmes d'analyse lexicale par exemple.

Sa forme:

```
char *strtok ( char * string,
               const char * delims );
```

Permet de décomposer la chaîne *string* en ses différents éléments lexicaux; *delims* contenant les caractères jouant le rôle de séparateurs. Le premier appel s'effectue avec la chaîne *string* comme paramètre, les suivants avec *NULL* à la place. A chaque appel, la fonction livre un pointeur sur le début d'un nouvel élément lexical dans *string* et *NULL* si *string* est épuisé; elle saute donc tous les séparateurs initiaux s'il y en a.

Note importante: lorsque la fonction trouve un séparateur, elle le remplace par un caractère nul (fin de chaîne); donc la chaîne originale est modifiée, à vous de la sauver si vous désirez pouvoir la réutiliser sous sa forme originale.

La compréhension de cette fonction n'étant pas évidente, voici un petit programme académique montrant son fonctionnement:

```
// Exemple d'utilisation de la fonction strtok
// STRTOK
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;
int main ( )
{ char *tempo,
       chaine [ ] = "Salut Toto, comment vas-tu?",
       *ligne = chaine,
       *separateurs = ", ?-";
  cout << "Chaine initiale: " << chaine << endl;
  tempo = strtok ( ligne, separateurs );
  /* Tant que la chaine n'est pas epuisee... */
  while ( tempo )
  {  /* traitement de l'unité lexicale */
     cout << tempo << endl;
     tempo = strtok ( NULL, separateurs );
  }
  cout << "\n\nChaine initiale transformee: " << chaine << endl;
  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'affichage de ce programme:

```
Chaine initiale: Salut Toto, comment vas-tu?
Salut
Toto
comment
vas
tu

Chaine initiale transformee: Salut
Fin du programme...Appuyez sur une touche pour continuer...
```

Initialisation d'une zone en mémoire

Voici une fonction pas directement liée aux chaînes de caractères, mais qui permet de recopier la valeur d'un octet dans tous les octets consécutifs d'une zone de mémoire:

```
void *memset ( void *string, int c, size_t n );
```

Recopie l'octet *c* dans les *n* premières positions de la zone *string*. Elle livre en retour la valeur de *string*. Comme souvent, faites attention, la valeur de remplissage est de type **int**, même si c'est un octet qui est réellement utilisé.

Conversions de chaînes:

Il existe plusieurs fonctions de conversion de chaînes de caractères en valeurs numériques. Commençons par les plus anciennes, moins générales et donc plus simples, qui n'existent a priori que pour des raisons de compatibilité avec d'anciens programmes, la norme C99 ayant introduit des fonctions plus générales, mais aussi plus complexes et moins utilisées.

Attention: ces fonctions ne sont pas définies dans c*string* mais dans *cstdlib*!

Conversion vers un réel double (**double**):

```
double atof ( const char *string );
```

Convertit la chaîne de caractères *string* en une valeur de type **double**. Tout se passe, du point de vue du traitement de la chaîne, comme pour la lecture externe d'une valeur du même type, c'est-à-dire la conversion s'arrête sur le premier caractère qui ne permet plus de construire une valeur du type. Par contre la norme ne dit pas ce qui se produit si l'on ne parvient pas à construire une valeur de ce type.

Conversion vers un entier long (**long int**):

```
long int atol ( const char *string );
```

Convertit la chaîne de caractères *string* en une valeur de type **long int**. Mêmes remarques que pour *atof*.

Conversion vers un entier (**int**) avec les mêmes remarques que pour *atof* et *atol*:

```
int atoi ( const char *string );
```

Maintenant certaines fonctions nouvelles, introduites par la norme C99, que nous vous conseillons d'utiliser et qui se trouvent aussi dans c*stdlib*.

Conversion vers un réel double (**double**):

```
double strtod ( const char * string,
                char ** end );
```

Livre comme résultat la conversion de la chaîne de caractères *string* en une valeur réelle en double précision. Les espaces blancs en tête sont sautés et la conversion s'arrête au premier caractère ne faisant plus partie d'un nombre réel; c'est l'adresse de ce dernier caractère qui est livrée dans le paramètre de sortie *end* (**attention**: pointeur de pointeur) Si la conversion s'avère impossible on obtient dans *end* l'adresse du début de *string* et 0.0 est livré comme valeur. Si *end* = *NULL* au moment de l'appel, il n'est simplement pas utilisé et l'on ne saura donc pas où s'est arrêtée la lecture.

Conversion en entier long (**long int**):

```
long int strtol ( const char * string,
                  char ** end, int base );
```

Livre comme résultat la conversion de la chaîne de caractères *string* en une valeur entière longue. La représentation de la valeur entière dans la chaîne peut prendre l'une des formes usuellement admises. Les espaces blancs en tête sont sautés et la conversion s'arrête au premier caractère ne faisant plus partie d'un nombre entier; c'est l'adresse de ce dernier caractère qui est livrée dans le paramètre de sortie *end* (**attention**: pointeur de pointeur). En cas de conversion impossible, la valeur 0 est livrée. La base, qui fait l'objet du troisième paramètre, peut être comprise entre 2 et 36. Toutefois si cette base vaut 0, elle correspond par défaut à la base 10, mais rien n'empêche l'utilisateur de commencer sa valeur par un 0…, auquel cas elle sera interprétée comme étant en octal; ou de commencer par 0x… auquel cas elle sera interprétée comme étant en hexadécimal. Au sujet du paramètre *end* on peut faire les mêmes remarques que pour *strtod*.

Conversion en entier long non signé (**unsigned long**):

```
unsigned long strtoul ( const char * string,
                        char ** end, int base );
```

Les mêmes indications que pour *strtol* restent valables.

Un exemple d'utilisation:

```
/*
  Exemple d'utilisation de fonctions de conversion
  CONVERSION.cpp
*/
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;

#define base1 8
#define base2 10
#define base3 16

int main ( )
{
  char * debut = " 012 012 1a 3.5", * fin;
  cout << "Exemple d'utilisation de fonctions de conversion\n";

  cout << strtol ( debut, &fin, base1 ) << endl;
  cout << strtol ( fin, &fin, base2 ) << endl;
  cout << strtol ( fin, &fin, base3 ) << endl;
  cout << strtod ( fin, &fin ) << endl;

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Programme qui affiche comme résultat:

```
Exemple d'utilisation de fonctions de conversion
10
12
26
3.5
Appuyez sur une touche pour continuer...
```

Chaînes C++ : string

Nous allons maintenant vous donner des indications sur les *strings* en C++ (donc non utilisables en C) et bien que cette partie vienne en fin de chapitre, nous vous rappelons qu'il s'agit de la forme que nous vous conseillons d'utiliser.

Pour cela il faut que votre module comporte la clause:

```
#include <string>
```

Déclaration, initialisation: constructeurs

Une variable string se déclare de manière usuelle:

```
string maChaine;
```

Nous pouvons aussi par l'intermédiaire de divers constructeurs initialiser cette chaîne lors de sa déclaration.

Les principaux constructeurs sont:

```
string maChaine ( "Salut Toto" );
```

On initialise *maChaine* avec le texte *Salut Toto*.

```
string maChaine ( "Salut Toto", 5 );
```

On initialise *maChaine* avec les 5 premiers caractères de la chaîne *Salut Toto* donc avec *Salut*. Avec une variable cela paraît plus raisonnable!

```
string autreChaine ( maChaine );
```

On initialise *autreChaine* avec le contenu d'une autre chaîne préalablement définie.

```
char *pt = "Salut Toto";
...
string maChaine ( pt );
```

On utilise un pointeur (donc une chaîne C classique) pour initialiser *maChaine*, ici à nouveau *Salut Toto.*

```
char *pt = "Salut Toto";
...
string maChaine ( pt, pt+5 );
```

On utilise une partie de la chaîne désignée par le pointeur *pt*, ici *maChaine* vaut *Salut.*

❑ Entrées/sorties

L'affichage de variables *string* ne pose aucun problème particulier, l'opérateur << étant surchargé pour ce type; notez simplement que la chaîne s'écrit sur un nombre de positions strictement égal à sa longueur actuelle.

En entrée, il existe aussi une surcharge de l'opérateur >>, mais là il faut bien comprendre que les caractères blancs (on dit aussi espaces blancs), à savoir: les espaces, les fins de lignes, les tabulations horizontales et verticales provoquent la fin de la lecture de la chaîne; de plus, ils sont ignorés en début de lecture.

Suivant la nature de l'application à réaliser ce comportement s'avère peu pratique, aussi la classe *basic_string* nous met-elle une fonction supplémentaire à disposition: *getline*:

```
getline ( cin, maChaine );
```

Cette fonction prend comme premier paramètre un flux d'entrée (pour nous et pour l'instant *cin*) et comme deuxième paramètre un objet de type *string*. La lecture s'arrête par défaut sur une fin de ligne; tous les caractères jusqu'à cette fin de ligne sont alors significatifs. Toutefois nous pouvons préciser comme troisième paramètre le caractère provoquant la terminaison de la lecture:

```
getline ( cin, maChaine, '*' );
```

Ici la lecture s'arrête sur le premier caractère '*' rencontré, cela implique:

- Le caractère en question n'est pas consommé, tout comme la fin de ligne dans la forme par défaut.
- La chaîne lue peut donc comporter des fins de lignes.
- La chaîne lue est vide si le premier caractère était: '*'.

Vous pouvez mélanger dans une même application l'utilisation de l'opérateur >> et des appels à la fonction *getline* (de même que la fonction *get* que nous avons déjà utilisée).

Autres opérateurs surchargés

L'affectation:

Il existe fort heureusement une surcharge de l'opérateur d'affectation:

```
maChaine = autreChaine;
```

Rappelons qu'avec les tableaux, donc avec les chaînes de caractères classiques C nous ne pouvons effectuer d'affectation.

Non seulement il permet comme ci-dessus d'affecter à un *string* un autre objet *string*, mais également:

```
char c_Chaine [] = "Tutu";
...
maChaine = c_Chaine;
```

Ici une chaîne classique C.

```
maChaine = 'a';
```

Ici un simple caractère.

L'opérateur = étant surchargé, il est aussi possible de déclarer un objet *string* et de l'initialiser sous la forme:

```
string maChaine = "Toto";
```

Les comparaisons:

La classe *string* nous met à disposition tous les opérateurs de comparaison (<, <=, ==, !=, >, >=) applicables non seulement entre 2 *string*, mais également entre *string* et chaîne C classique. Le résultat est une valeur booléenne: **true** ou **false**. Seul l'ordre des codes des caractères (ordre lexicographique) entre en considération pour déterminer le résultat, sauf si tous les premiers caractères de la chaîne la plus longue se révèlent identiques à ceux de la plus courte; dans ce cas la chaîne la plus longue sera aussi la plus grande.

Accès aux caractères de la chaîne:

Il existe également une surcharge de l'opérateur [] ce qui nous permet d'accéder aux caractères individuels de la chaîne comme pour un tableau classique:

```
maChaine [2] = 'a';
```

Cela signifie, entre autres, que nous pouvons introduire des caractères nuls dans une telle chaîne, cela ne modifiera en aucun cas sa longueur, le caractère nul faisant alors partie de la chaîne comme n'importe quel autre caractère.

Comme nous l'avions signalé pour la classe vecteur, l'accès sous cette forme aux éléments n'implique aucun contrôle. Cela signifie que nous pouvons tenter d'accéder à des caractères qui n'existent pas et donc éventuellement faire d'importants dégâts au reste de notre application.

Egalement comme nous l'avions vu pour la classe vecteur, si nous désirons qu'un contrôle soit effectué nous utiliseront la méthode *at* fournie par la classe plutôt que l'opérateur []:

```
maChaine.at (2) = 'a';
```

Dans ce cas si l'indice ne correspond pas à un caractère existant, une exception sera levée et nous finirons bien par apprendre à les traiter…

❑ Concaténation

La concaténation représente une opération classique sur les chaînes et la classe surcharge l'opérateur + dans ce but:

```
maChaine = maChaine + autreChaine;
```

La concaténation peut se faire non seulement entre 2 strings, mais également avec un caractère ou une chaîne à la C.

Vouloir concaténer une chaîne (ou un caractère) à la suite d'une autre avec résultat dans la chaîne initiale représente une opération suffisamment courante pour que l'on nous mette également à disposition l'opérateur +=. Ainsi l'exemple ci-dessus peut s'écrire plus simplement:

```
maChaine += autreChaine;
```

Les méthodes de la classe

En plus des opérateurs, la classe *basic_string*[1] met à disposition un nombre important de méthodes. Nous allons décrire celles qui nous semblent les plus importantes.

Caractéristiques des chaînes

Nous obtenons la longueur courante d'une chaîne[2] par la méthode *length* ou *size* qui lui est totalement équivalente:

```
cout << maChaine.length ();
cout << maChaine.size ();
```

Bien entendu si *maChaine.length == 0* alors la chaîne est vide; toutefois nous rendrons la compréhension plus directe en utilisant la méthode *empty ()* qui livre **true** pour une chaîne vide et **false** sinon:

```
if ( maChaine.empty () ) ...
```

La grandeur de la chaîne se modifie par la méthode *resize*, qui dans sa forme simple prend un seul paramètre: une valeur entière représentant la nouvelle taille:

```
maChaine.resize ( 20 );
```

L'opération tronque simplement la chaîne si sa taille actuelle est plus grande que la nouvelle valeur. Par contre, si la nouvelle taille est plus grande, la chaîne se verra complétée par autant de caractères nuls que nécessaire[3]. Toutefois si ce caractère de remplissage par défaut ne vous convient pas, vous pouvez préciser en deuxième paramètre de la méthode le caractère à utiliser, par exemple:

```
maChaine.resize ( 20, '*' );
```

[1] Elle est générique, et string en est une instance.

[2] En pratique il se peut que la taille physique (la place mémoire déjà réservée pour d'éventuelles extensions de la chaîne) soit plus grande que la taille "logique" du contenu actuel de la chaîne; vous obtenez cette valeur par la méthode *capacity ()*. De plus, si vous y tenez vraiment, vous pouvez obtenir la taille maximale que pourrait atteindre votre chaîne par l'appel de la méthode: *max_size ()*!

[3] Ceci permet entre autres de rester compatible avec les chaînes à la C.

Parfois, pour des raisons de performance, on désire agrandir la taille physique en mémoire de la chaîne sans modifier sa grandeur logique[1]. On réalise cette opération par l'intermédiaire de la méthode *reserve* qui prend en paramètre la taille désirée, exemple:

```
maChaine.reserve ( 30 );
```

Après cette opération la méthode *length ()* livre toujours l'ancienne grandeur logique de la chaîne, alors que *capacity ()* livre la grandeur physique réservée.

❑ Affectation

Nous avions vu l'existence de la surcharge de l'opérateur =; la méthode *assign* avec ses différentes surcharges permet de faire le même travail tout en offrant des possibilités plus fines.

La forme la plus simple:

```
maChaine.assign ( autreChaine );
```

*maChain*e prend ainsi la même valeur que *autreChaine*, mais il s'agit bien d'une copie de cette valeur.

Le paramètre peut aussi prendre la forme d'une chaîne C classique.

Dans le cas où nous transmettons une chaîne de type *string* nous pouvons ajouter 2 paramètres à la méthode, permettant ainsi de n'affecter qu'une partie de la chaîne source:

```
maChaine.assign ( autreChaine, 1, 2 );
```

Le deuxième paramètre précise la position de premier caractère à prendre en considération (n'oubliez pas qu'en C++ la numérotation part de 0!) le troisième paramètre quant à lui indique le nombre de caractères à prendre en considération.

Dans le cas où nous transmettons une chaîne classique C nous ne pouvons ajouter qu'un seul paramètre qui précise alors la longueur de la chaîne à prendre en considération; on n'a pas le choix, on doit partir du premier caractère:

```
char *c_Chaine  = "Tutu";
...
maChaine.assign ( c_Chaine, 3 );
```

[1] Parce que l'on sait par exemple qu'en cours de traitement nous aurons effectivement besoin de cette taille et que nous voulons éviter des opérations de réallocation et donc éventuellement de déplacement en mémoire, opérations coûteuses en temps!

Par contre, pour une chaîne C nous arrivons à n'utiliser qu'une partie de la chaîne ceci par une surcharge d'*assign* qui nécessite un pointeur sur le début de la zone à copier et un deuxième désignant la fin de la zone:

```
char *c_Chaine  = "Tutu";
...
maChaine.assign ( c_Chaine+1, c_Chaine+3 );
```

Finalement pour cette catégorie d'opérations signalons la possibilité d'affecter à la chaîne un certain nombre de fois le même caractère, en précisant comme premier paramètre le nombre de fois à répéter et comme deuxième paramètre, le caractère désiré:

```
maChaine.assign ( 4, '*' );
```

Nous avions aussi vu la surcharge de l'opérateur +=; la méthode *append* avec ses différentes surcharges permet de faire le même travail tout en offrant des possibilités plus fines. Cette méthode offre exactement les mêmes variantes avec les mêmes paramètres que la méthode *assign* ci-dessus, nous ne la décrirons donc pas plus en détail!

❑ Comparaisons

Nous avons vu que la classe *basic_string* met à disposition tous les opérateurs de comparaisons et nous vous conseillons pour la lisibilité de les utiliser dans la mesure du possible. Toutefois elle offre aussi une méthode surchargée *compare* qui rend comme résultat une valeur entière:

- Négative si la chaîne à laquelle on applique la méthode est plus petite que celle transmise en paramètre
- Nulle si la chaîne à laquelle on applique la méthode est égale à celle transmise en paramètre
- Positive si la chaîne à laquelle on applique la méthode est plus grande que celle transmise en paramètre

```
if ( maChaine.compare ( autreChaine ) > 0 ) ...
```

Nous pouvons ne prendre en considération pour la comparaison qu'une tranche de la chaîne à laquelle on applique la méthode en précisant comme premier paramètre l'indice de début de la tranche et celui de la fin comme deuxième:

```
if ( maChaine.compare ( 1, 2, autreChaine ) > 0 ) ...
```

Nous pouvons aussi fixer une tranche pour la chaîne passée en troisième paramètre, avec les mêmes conventions que pour la première:

```
if ( maChaine.compare ( 1, 2, autreChaine, 2, 3 ) > 0 ) ...
```

Toutefois si le troisième paramètre correspond à un chaîne C classique, nous n'avons théoriquement pas le choix de la position de départ qui vaut toujours 0 et nous ne devons spécifier qu'un seul paramètre en plus représentant le nombre de caractères à prendre en considération[1].

❑ Extraction / Copie

La méthode *substr* permet d'extraire une sous-chaîne d'une chaîne et elle livre cette sous-chaîne en retour:

```
autreChaine = maChaine.substr ( 1, 2 );
```

Son premier paramètre précise la position de départ dans la chaîne source, son deuxième le nombre de caractères à prendre en considération.

La méthode *copy* permet d'affecter à une chaîne C classique le contenu d'un objet *string*; il s'agit bien d'une copie que nous pouvons par la suite manipuler indépendamment[2].

```
char c_Chaine [20];
...
maChaine.copy ( c_Chaine, maChaine.length () );
```

Le premier paramètre consiste en un pointeur sur le début de la zone où seront recopié les caractères de la chaîne source; le deuxième spécifie le nombre de caractères à copier, dans notre exemple nous avons choisit toute la chaîne.

Cette méthode comporte un troisième paramètre qui par défaut vaut 0. Il précise à partir de quel caractère de la chaîne source on commence la copie, ce qui implique par défaut: depuis son début.

[1] L'expérience pratique prouve que des compilateurs ne réagissent pas de cette manière et permettent également pour une chaîne C de préciser la position de départ et le nombre de caractères à considérer pour la chaîne donnée en paramètre.

[2] Les méthodes *c_str ()* et *data ()* livrent en retour une chaîne C classique sous la forme d'un pointeur constant. Théoriquement *c_str* livre une chaîne se terminant au premier caractère nul du *string* alors que data livre toute la chaîne source y compris les éventuels caractères nuls qui seraient présent; toutefois on constate que des implémentations ne font aucune différence!

Comme pour toutes les autres méthodes de la classe *basic_string* nous devons impérativement donner des valeurs correspondant à des caractère existant effectivement dans la chaîne source sous peine de lever une exception; par contre le compilateur ne dispose d'aucun moyen pour contrôler que la zone de destination possède une taille effective suffisante, c'est au programmeur de prendre les précautions nécessaires.

Insertion

Les différentes surcharges de la méthode *insert* permettent d'introduire de nouveaux éléments dans une chaîne existante. Elles prennent toutes comme premier paramètre la position à laquelle on réalise l'insertion.

La forme la plus directe permet d'insérer un autre *string* ou une chaîne C donnée en deuxième paramètre:

```
maChaine.insert ( 2, autreChaine );
```

Vous pouvez indiquer en plus la position du caractère de départ et le nombre de caractères à prendre en considération, ceci permettant de n'insérer qu'une partie de la chaine:

```
maChaine.insert ( 2, autreChaine, 1, 3 );
```

Insert offre aussi la possibilité d'introduire une ou plusieurs fois le même caractère dans la chaîne. Il faut alors préciser le nombre de fois que l'on désire le caractère comme deuxième paramètre et le caractère lui-même comme troisième:

```
maChaine.insert ( 2, 1, 'x' );
```

Suppression

Nous savons que nous pouvons affecter une chaîne vide à un *string*, ce qui a pour conséquence d'effacer tous les caractères qui s'y trouvaient! La méthode *clear ()* réalise le même travail:

```
maChaine.clear ( );
```

Nous pouvons aussi n'effacer qu'une partie de la chaîne pour cela la méthode *erase* permet de spécifier:

- Que le caractère à partir duquel on efface tout le reste de la chaîne:

```
maChaine.erase ( 1 );
```

- Le caractère à partir duquel on efface et le nombre à effacer:

```
maChaine.erase ( 2, 3 );
```

Remplacement

Tout d'abord la méthode *swap* permet d'échanger le contenu de 2 chaînes:

```
maChaine.swap ( autreChaine );
```

Ce qui revient évidemment au même que d'écrire:

```
autreChaine.swap ( maChaine );
```

Nous pouvons aussi remplacer une partie de chaîne par une autre chaîne sans modifier la chaîne insérée qui, elle, peut-être de type *string* ou une chaîne C classique:

```
maChaine.replace ( 1, 3, autreChaine );
```

Dans l'exemple ci-dessus *autreChaine* vient remplacer 3 caractères à partir de la position 1 dans *maChaine*.

Nous pouvons aussi remplacer par la répétition d'un certain nombre de fois un caractère donné:

```
maChaine.replace ( 1, 3, 5, '*' );
```

Dans l'exemple ci-dessus 5 étoiles viennent remplacer 3 caractères à partir de la position 1 dans *maChaine*.

Notez la similitude avec la méthode *insert*.

Rechercher

La recherche dans une chaîne représente certainement l'une des opérations les plus courantes sur les chaînes, aussi la classe *basic_string* nous met-elle plusieurs méthodes surchargées à disposition dans ce but.

Toutes ces méthodes livrent en retour la position dans la chaîne source où commence le motif recherché. En fait la valeur retournée n'est pas du type **int** (mais avec le jeu de conversions

cela fonctionne aussi!) mais du type: *string::size_type*[1]; nous pouvons donc nous déclarer explicitement des variables de ce type:

```
string::size_type position = 0;
```

Si la recherche n'aboutit pas (le motif recherché ne se trouvant dans la chaîne source) les différentes méthodes retourneront alors la valeur *string::npos* dont la valeur dépend de l'implémentation.

Toutes ces méthodes possèdent comme premier paramètre le motif à rechercher que l'on peut donner sous la forme:

- D'un objet de type *string*.
- D'une chaîne C classique, donc un pointeur.
- D'un seul caractère.

Toutes aussi possèdent un deuxième paramètre spécifiant la position dans la chaîne source à partir de laquelle il faut commencer la recherche. Nous n'avons pas nécessairement besoin de le spécifier car il comporte une valeur par défaut raisonnable, à savoir le premier caractère pour une recherche allant du début vers la fin de la chaîne, et le dernier caractère pour une recherche en sens inverse.

Si le motif se donne par l'intermédiaire d'une chaîne C classique nous pouvons encore donner un paramètre supplémentaire précisant le nombre de caractère à prendre en considération.

Les 2 principales méthodes sont respectivement *find* et *rfind*; *find* commence la recherche à partir du début de la chaîne à laquelle on applique la méthode, alors que *rfind* réalise le même travail en commençant par la fin.

Exemple:

```
maChaine.find ( autreChaine );
maChaine.rfind ( autreChaine, 10 );
```

Attention pour ce deuxième exemple, il signifie "entre le dixième et le premier caractère".

Ensuite viennent les fonctions de recherche de la position d'un caractère parmi un ensemble:

```
maChaine.find_first_of ( autreChaine );
```

Livre la position du premier caractère dans *maChaine* qui se trouve aussi dans *autreChaine*, par exemple si *autreChaine* vaut: "0123456789" nous obtiendrons la position du premier chiffre dans *maChaine*.

[1] En pratique dans tous nos exemples qui précèdent nous avons utilisé des constantes pour fixer des positions ou des grandeurs; si nous avions utilisé des variables nous devrions pour être propre les déclarer de ce type!

```
maChaine.find_last_of ( autreChaine );
```

Livre la position du dernier caractère dans *maChaine* qui se trouve aussi dans *autreChaine.*

```
maChaine.find_first_not_of ( autreChaine );
```

Livre la position du premier caractère dans *maChaine* qui **ne** se trouve **pas** dans *autreChaine.*

```
maChaine.find_last_not_of ( autreChaine );
```

Livre la position du dernier caractère dans *maChaine* qui **ne** se trouve **pas** dans *autreChaine.*

❑ Compléments

La grande majorité des méthodes de la classe *basic_string* (il y aura une exception pour les itérateurs que nous n'avons pas abordés) livre comme résultat la référence à l'objet lui-même, ce qui nous permet éventuellement "d'enchaîner les opérations" comme dans l'exemple suivant:

```
maChaine = "1234567";
cout << maChaine.append ( 4, 'a' ).length() << endl;
```

Fin de la partie 1

Table des matières

INTRODUCTION 3

- **LE LANGAGE** 3
- **LES OBJECTIFS A ATTEINDRE** 4

ELEMENTS DE BASE 7

- **PREMIER EXEMPLE** 7
- **LES COMMENTAIRES** 8
- **STRUCTURE GENERALE D'UN PROGRAMME** 10
- INSTRUCTION ET BLOC 10
- **IDENTIFICATEURS ET MOTS RESERVES** 12
- **MISE EN FORME D'UN PROGRAMME** 15
- **LES BASES DE L'AFFICHAGE** 15
- **CONCLUSIONS SUR L'INTRODUCTION** 17

LES TYPES DE BASE 18

- **LES TYPES DE BASE SONT** 18
- LES ENTIERS 19
- LES CARACTERES 20
- LES REELS 20
- LES BOOLEENS 21
- **FORME DES DECLARATIONS** 21
- **FORME DES CONSTANTES** 23
- CONSTANTES CARACTERES 23
- Caractères spéciaux 23
- CONSTANTES CHAINES DE CARACTERES 24
- CONSTANTES ENTIERES 25

CONSTANTES REELLES 26

- **EXEMPLE** 26
- **DEFINITIONS DE CONSTANTES** 28
- "CONSTANTES" CONST 28
- "CONSTANTES" #DEFINE 29
- **EXEMPLE** 30
- **LES BASES DE LA LECTURE** 31
- **LES REFERENCES** 32

OPERATEURS ET EXPRESSIONS 34

- **CONSIDERATIONS GENERALES/ AFFECTATION SIMPLE** 34
- **L'OPERATION D'ENCHAINEMENT** 36
- **LES OPERATIONS ARITHMETIQUES** 36

- **CONVERSIONS DE TYPES 37**
- CONVERSIONS EXPLICITES 38
- L'opérateur cast 38
- La forme fonctionnelle 39
- **EXEMPLE: 39**
- **OPERATEURS DE COMPARAISONS 41**
- **LES OPERATEURS LOGIQUES 43**
- **LES OPERATEURS DE MANIPULATION DE BITS 44**
- LES OPERATEURS DE DECALAGE 44
- OPERATEURS LOGIQUES BIT A BIT 46
- APPLICATION 49
- **EXEMPLE 49**
- **L'OPERATEUR CONDITIONNEL 52**
- **LES OPERATEURS D'INCREMENTATION ET DE DECREMENTATION 53**
- **LES OPERATEURS D'AFFECTATIONS 54**
- **OPERATEUR SIZEOF 56**
- **EXEMPLE 57**
- **CONCLUSION 58**

STRUCTURES DE CONTROLE 60

- **SEQUENCES ET BLOCS 60**
- **LES INSTRUCTIONS DE SELECTION 62**
- L'INSTRUCTION IF 62
- EXEMPLE 65
- L'INSTRUCTION SWITCH 66
- **LES INSTRUCTIONS DE BOUCLES 71**
- DO WHILE 71
- EXEMPLE 74
- LA BOUCLE WHILE 77
- EXEMPLE 78
- LA BOUCLE FOR 80
- EXEMPLES 84
- **INSTRUCTION BREAK 87**
- **INSTRUCTION CONTINUE 88**
- **INSTRUCTION GOTO 89**

SOUS-PROGRAMMES: FONCTIONS 90

- **INTRODUCTION 90**
- **LE "TYPE" VOID 92**
- EXEMPLE 93
- **LES PARAMETRES 94**
- PARAMETRES PAR VALEUR 94
- EXEMPLE 97
- PARAMETRES VARIABLES (ENTREE/SORTIE) 98
- Par référence 98
- Par pointeur 101

- EXEMPLE 103
- COMPLEMENT: 103
- VALEUR DE RETOUR 104
- EXEMPLE: 105
- COMPLEMENT: 106
- **RECURSIVITE 107**
- EXEMPLE COMPLET 110
- **PROTOTYPES ET FONCTIONS EXTERNES 111**
- COMPILATION SEPAREE 113
- **NOMBRE DE PARAMETRES VARIABLE 114**
- **SOUS-PROGRAMME INLINE 114**
- **VALEURS PAR DEFAUT 115**
- **LA SURCHARGE 117**

CLASSES DE DECLARATIONS ET ATTRIBUTS 120

- **VARIABLES LOCALES ET GLOBALES 121**
- **EXEMPLE 123**
- **VARIABLES EXTERNES 125**
- **CLASSE REGISTER 126**
- **VARIABLES LOCALES A UN BLOC 127**
- **ATTRIBUT CONST 128**
- **ATTRIBUT VOLATILE 128**
- **EN CONCLUSION 129**

LE PREPROCESSEUR 131

- **INTRODUCTION 131**
- **#DEFINE 131**
- REMARQUES COMPLEMENTAIRES 133
- **#UNDEF 135**
- **LES MACRO-INSTRUCTIONS 136**
- COMPLEMENT SUR LES PARAMETRES DES MACROS 137
- **MACROS PREDEFINIES 138**
- **DIRECTIVES PREDEFINIES 139**
- **INCLUSION DE FICHIERS 141**
- REMARQUES COMPLEMENTAIRES 142
- **COMPILATION CONDITIONNELLE 143**
- QUELQUES CAS PARTICULIERS POUR LES EXPRESSIONS DU #IF: 145

LES TABLEAUX 146

- **INTRODUCTION 146**
- **LES TABLEAUX UNIDIMENSIONNELS 147**
- **EXEMPLE 149**
- **INITIALISATION DE TABLEAUX 152**
- **FONCTIONS GENERALES POUR TABLEAUX A 1 DIMENSION 153**
- **LES TABLEAUX MULTIDIMENSIONNELS 154**

- TABLEAUX A 2 DIMENSIONS 154
- **PROBLEME DU PASSAGE EN PARAMETRE 157**
- **FONCTIONS GENERALES POUR TABLEAUX MULTIDIMENSIONNELS? 159**
- **UN PEU DE CLASSE 160**
- LA CLASSE VECTOR 162
- **COMPLEMENTS C99 ..... 170**
- INITIALISATION PAR NOM 172
- **COMPLEMENT C++11 : LISTES D'INITIALISATEURS 173**
- **COMPLEMENT C++11 : TABLEAUX DE TAILLE FIXE EN TANT QUE CONTENEUR 173**

LES POINTEURS 175

- **INTRODUCTION 175**
- **DECLARATIONS 176**
- **UTILISATION 180**
- LE MOT-CLE *NULLPTR* (C++11) 184
- **EXEMPLE: 186**
- **ADRESSE DES TABLEAUX ET CALCULS SUR LES POINTEURS 187**
- OPERATEURS ARITHMETIQUES SUR LES POINTEURS 189
- **EXEMPLE 191**
- **REMARQUES COMPLEMENTAIRES: 192**
- **POINTEURS SUR LES FONCTIONS 194**
- **DIFFERENCE ENTRE POINTEUR ET TABLEAU 197**
- **VARIABLES DYNAMIQUES/RETOUR SUR LES TABLEAUX 198**
- OPERATEURS NEW ET DELETE 198
- FONCTIONS DE LA BIBLIOTHEQUE C 205
- **RESOUDRE LE PROBLEME DES TABLEAUX VARIABLES EN PARAMETRE 208**
- **REFLEXION ! 215**

CARACTERES - CHAINES DE CARACTERES - STRING 216

- **LES CHAINES DE CARACTERES 216**
- **CHAINES TRADITIONNELLES C 216**
- CATEGORIES DE CARACTERES 219
- LA BIBLIOTHEQUE CSTRING 222
- Rappels de base 222
- Longueur 223
- Copie 223
- Concaténation 225
- Comparaison 225
- Recherche 226
- Décompositions 228
- Initialisation d'une zone en mémoire 229
- Conversions de chaînes: 229
- **CHAINES C++ : STRING 232**
- Déclaration, initialisation: constructeurs 232
- Entrées/sorties 233
- Autres opérateurs surchargés 234

- Concaténation 235
- LES METHODES DE LA CLASSE 236
- Caractéristiques des chaînes 236
- Affectation 237
- Comparaisons 238
- Extraction / Copie 239
- Insertion 240
- Suppression 240
- Remplacement 241
- Rechercher 241
- COMPLEMENTS 243

TABLE DES MATIERES 245