Vous apprécierez davantage la programmation
si vous la comparez à une création littéraire
destinée à être lue.
Donald E. Knuth

CHAPITRE I

Une brève introduction à la programmation en C++

Objectifs

- Apprendre les éléments du langage C++ afin de programmer quelques petits exemples.
- Se familiariser avec quelques concepts fondamentaux de la programmation : variable, type, instruction, condition, boucle, fonction, paramètre, copie vs référence.

Ce premier chapitre sert d'introduction au langage de programmation C++, dans le but de pouvoir mettre en œuvre des algorithmes de calcul. Même si vous n'avez jamais programmé, ne vous effrayez pas : nous commençons par des exemples faciles. Les notions de base présentées ici suffiront pour nos besoins modestes dans ce cours. À long terme il vous sera sans doute utile d'élargir et d'approfondir vos connaissances en programmation : vous êtes vivement invités à consulter le rayon « C++ » de votre bibliothèque universitaire afin d'y trouver l'œuvre qui vous convienne le mieux. Vous pouvez également vous renseigner sur internet, par exemple sur le site `www.cplusplus.com`.

Comment démarrer ? Dans toute la suite nous allons travailler sous le système GNU/Linux. Avant tout, loguez-vous sur une machine et ouvrez une fenêtre `xterm` pour y entrer des commandes. Afin de sauvegarder vos futurs fichiers vous pouvez créer un répertoire de travail, nommé `mae` pour fixer les idées, puis certains sous-répertoires. Il suffit pour ce faire d'utiliser la commande `mkdir` (*make directory*), en l'occurrence en tapant `mkdir mae`. Ensuite vous pouvez vous placer dans votre répertoire `mae` en tapant `cd mae` (*change directory*).

Où trouver les fichiers d'exemples ? Vous pouvez copier la plupart des fichiers d'exemples pour ne pas les retaper ; vous les trouvez tous dans le répertoire `~eiserm/mae`. Vous pouvez créez un lien symbolique (*link*) par `ln -s ~eiserm/mae source`. Vérifier par `ls` (*list*) que vous avez bien créé un lien qui s'appelle `source`. Le lien s'utilise comme un sous-répertoire ; par exemple, `ls source/chap01` montre la liste des fichiers disponibles pour le chapitre 1. Afin de travailler avec ces fichiers, surtout pour les modifier, vous les recopiez dans votre répertoire en tapant `mkdir chap01`, puis `cp -v source/chap01/* chap01`. Il sera utile de suivre la même démarche pour les chapitres à venir.

Sommaire

1. Éditer et compiler des programmes

Placez-vous dans le répertoire souhaité, par exemple en tapant `cd ~/mae`, puis tapez la commande `emacs somme.cc` suivi d'un signe '`&`'. L'éditeur de texte `emacs` s'ouvre alors dans une fenêtre. Ouvrez un fichier, disons « somme.cc » et vous pouvez commencer à taper votre texte.

1.1. Un premier programme. Recopiez le texte suivant en utilisant la touche de tabulation après chaque passage à la ligne, de façon à obtenir une indentation automatique. Tout texte après le symbole '`//`' jusqu'à la fin de la ligne sert de commentaire. (Il sert à votre information uniquement ; ne le retapez pas afin d'économiser du temps.)

Programme I.1 Un programme impeccable `somme+.cc`

```
#include <iostream>           // fichier en-tête iostream pour l'entrée-sortie
using namespace std;          // accès direct aux objets et fonctions standard

int ma_fonction( int n )      // définition d'une fonction à un paramètre n
{                             // cette accolade commence le corps de la fonction
  return n*n;                 // cette instruction calcule et renvoie la valeur n^2
}                             // cette accolade termine le corps de la fonction

int main()                    // définition de la fonction principale
{                             // cette accolade commence le corps de la fonction
  cout << "Calcul de la somme de ma fonction f(k) pour k=a,..,b" << endl;
  cout << "Donnez les valeurs des bornes a et b svp : ";
  int min, max;               // définition de deux variables appelées min et max
  cin >> min >> max;          // lecture des deux valeurs du clavier
  int somme= 0;               // définition et initialisation de la variable somme
  for( int k= min; k <= max; k= k+1 )             // boucle allant de min à max
    {                                             // début du bloc de la boucle
      cout << somme << "+" << ma_fonction(k);     // afficher les deux valeurs
      somme= somme + ma_fonction(k);              // recalculer la somme
      cout << " = " << somme << endl;             // afficher la nouvelle somme
    }                                             // fin du bloc de la boucle
  cout << "La somme vaut " << somme << endl;      // afficher le résultat final
}                             // cette accolade termine le corps de la fonction
```

Pour sauvegarder votre texte cliquez sur `save` dans le menu `file`. Pour lancer la compilation cliquez sur `compile` dans le menu `tools`, puis remplacez `make -k` par la commande `g++ somme.cc` au bas de la fenêtre emacs. La fenêtre se scinde en deux parties ; l'une contient votre programme, l'autre vous montre l'évolution de la compilation et les erreurs éventuelles. En cas de succès, la compilation se termine par le message `Compilation finished at ...` Vous avez obtenu un fichier exécutable, nommé `a.out`, dans votre répertoire de travail. Revenez dans la fenêtre initiale `xterm` et lancez la commande `a.out` (il faut éventuellement taper `./a.out`).

Exercice/P 1.1. Essayez, par exemple, de calculer $\sum k^3$ à la place de $\sum k^2$. Puis, si vous êtes courageux ou impatient, essayez de calculer $\sum k!$. (Pour ce faire, il faudra introduire une boucle pour calculer $n!$ dans `ma_fonction`. Devinez-vous déjà comment le faire ? Si non, veuillez patienter ... et lire la suite.)

Remarque 1.2. Notez qu'il y a toujours *un seul* fichier `a.out` dans votre répertoire de travail. Conserver des exécutables n'est pas toujours une bonne idée : vérifiez la taille de ce fichier en tapant `ls -l`. Si vous voulez conserver un exécutable, vous pouvez le renommer à l'aide de la commande `mv` (*move*), par exemple en tapant `mv a.out somme`, avant de compiler un autre programme dans le même répertoire. De manière alternative, la compilation avec `g++ somme.cc -o somme` compile le fichier source `somme.cc` et produit un exécutable nommé `somme`.

1.2. Un programme erroné. Vous pouvez maintenant éditer votre second programme (volontairement faux afin de provoquer des erreurs de compilation). Cliquez sur `files` puis sur `open` et entrez le nom `faux.cc` du fichier à ouvrir dans la ligne de commande. Tapez alors le texte du programme I.2 et lancez la compilation. Vous obtenez, entre autre, les messages suivants :

```
faux.cc: In function ‘int main()’:
faux.cc:10: ‘cout’ undeclared (first use this function)
faux.cc:10: ‘i’ undeclared (first use this function)
faux.cc:11: ‘endl’ undeclared (first use this function)
faux.cc:11: parse error before ‘int’
faux.cc:12: non-lvalue in assignment
```

Si vous placez la souris sur la ligne `faux.cc:10` et que vous cliquez deux fois avec le bouton du milieu, le curseur se placera automatiquement sur la ligne correspondante du texte source.

Programme I.2 Un programme erroné `faux.cc`

```
 1  int carre( int n )
 2  {
 3    return n*n;
 4  }
 5
 6  int main()
 7  {
 8    for( int p=0; p<10; p++ )
 9      {
10        cout << carre(i) << endl
11          int k;
12        5= p;
13      }
14  }
```

Essayons de rectifier le programme ci-dessus. Le flot `cout` est déclaré par exemple dans le fichier `iostream` : nous devons ajouter la directive `#include <iostream>` en début de fichier. Recompilez, puis ajouter `using namespace std;` pour voir la différence.

Dans la ligne 10, la variable `i` n'est pas déclarée : il convient de remplacer `i` par `p`. L'erreur suivante provient de l'absence du point virgule après l'instruction

`cout << carre(i) << endl`

(Remarquez à ce propos la mauvaise indentation de la ligne suivante).

La dernière erreur vient de ce que le signe ‘`=`’ est le signe d'affectation et non le signe d'égalité. On ne peut affecter de valeur à une constante : on dit que ce n'est pas une valeur à gauche (*left value*). Une fois les erreurs corrigées la compilation se déroule normalement.

1.3. Structure globale d'un programme en C++. Comme on a déjà vu dans le programme I.1, un programme en C++ se compose de certains éléments typiques. Avant de parler des détails dans le paragraphes suivants, donnons un aperçu de sa structure globale :

Variables: Les variables représentent les données et l'état du logiciel. Pendant l'exécution, leurs valeurs subissent certains changements, prescrits par les instructions du programme. Le comportement et la capacité d'une variable sont déterminés par son *type* (voir §2).

☞ La définition d'une variable peut, en C++, être faite n'importe où dans le code (mais, bien entendu, avant l'utilisation). Cela permet de la définir aussi près que possible de l'endroit où elle est utilisée : on améliore ainsi la lisibilité du code.

Instructions: Les actions décrites dans le code source du programme sont nommées *instructions*. Il s'agit par exemple des calculs, des affectations, des opérations d'entrée-sortie, etc. Les instructions conditionnelles, avant tout, seront discutées en §3. Un programme consiste ainsi d'une liste d'instructions, séparées par des point-virgules.

☞ On a souvent intérêt à regrouper plusieurs instructions en un *bloc* en les mettant entre deux accolades ‘`{`’ et ‘`}`’. Vous pouvez considérer un bloc comme une seule instruction, dite *complexe* car elle est composée de plusieurs instructions plus élémentaires. Le point-virgule obligatoire après chaque instruction est facultatif après un bloc $\{\ldots\}$.

Fonctions: Pour augmenter la lisibilité d'un logiciel, il est en général indispensable de le découper en sous-programmes, appelés *fonctions*. Il s'agit d'un bloc d'instructions (le *corps* de la fonction) qui porte un nom (figurant à la *tête* de la fonction). On peut ainsi appeler la fonction de n'importe où dans le programme.

☞ Vous pouvez regarder un programme comme une longue liste d'instructions, tout comme un roman, aussi long qu'il soit, consiste une suite de mots. Cependant, à partir d'une certaine taille, il est nécessaire d'introduire plus de structure : des paragraphes, des sections, des chapitres, ... En C++ une façon de ce faire est le regroupement en fonctions (puis en classes et modules).

☞ Une fonction peut avoir des *paramètres* et peut renvoyer une *valeur*, ce qui fait l'objet du §4. Le programme principal est une fonction dont le nom doit être `main`.

Directives: Ce sont des instructions spéciales (précédées du caractère dièse `#`) qui sont traitées avant la compilation. Le préprocesseur modifie le texte source du programme en y incluant, par exemple, le contenu des fichiers en-tête. L'usage des fichiers en-tête est une façon archaïque mais éprouvée pour déclarer les fonctions d'une bibliothèque que l'on utilisera dans le programme. C'est le cas pour le fichier `iostream`, qui déclare l'entrée-sortie standard (voir §5).

Commentaires: Il est souhaitable, voire nécessaire, d'inclure des commentaires pour expliquer le fonctionnement du programme (voir §7). Ces commentaires doivent être compris entre `/*...*/`. On peut aussi commenter une ligne en la faisant débuter par `//`.

Soulignons finalement la fameuse distinction entre *définition* et *déclaration* :

Définitions: Tout objet utilisé (variable, constante, fonction) doit être défini *exactement une fois* dans le programme, ni plus ni moins. La définition d'une variable réserve la mémoire nécessaire ; la définition d'une constante spécifie sa valeur ; la définition d'une fonction génère le code associé.

Déclarations: Tout objet (variable, constante, fonction) doit être déclaré *au moins une fois* avant d'être utilisé dans le programme. Ceci permet au compilateur de connaître déjà le type de l'objet afin d'y faire référence, alors que l'objet lui-même peut être défini plus tard.

Toute définition est aussi une déclaration, mais non réciproquement.

Programme I.3 Déclaration vs définition `affiche.cc`

```
1   #include <iostream>                // directive pour inclure iostream
2   using namespace std;               // accès direct aux fonctions standard
3
4   void affiche( int i );             // déclaration de la fonction affiche
5
6   int main()                         // définition de la fonction main
7   {
8     int n;                           // définition de la variable n
9     cout << "donnez un entier : ";    // première utilisation du flot cout
10    cin >> n;                        // première utilisation de cin et n
11    affiche(n);                      // appel de la fonction affiche
12  }
13
14  void affiche( int i )              // définition de la fonction affiche
15  {
16    cout << "la valeur est " << i << endl;
17  }
```

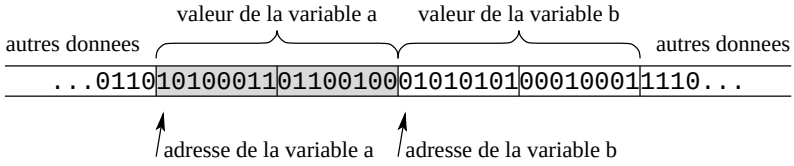
Question/P 1.3. Pourquoi la déclaration de la fonction `affiche()` est-elle nécessaire dans la ligne 4 du programme I.3 ? Pourrait-on s'en passer ? Que se passe-t-il si vous la mettez en commentaire ? Le comparer avec le programme I.1 : où s'effectue la déclaration des fonctions dans ce dernier ?

Remarque 1.4. La déclaration séparée d'une fonction, si elle n'est pas toujours nécessaire dans de petits programmes, est indispensable si l'on veut appeler une fonction avant de l'avoir définie, comme c'est le cas dans le programme I.3. La déclaration séparée devient obligatoire si deux fonctions s'appellent mutuellement ou si l'on veut appeler une fonction dans des modules distincts (voir à ce propos la « programmation modulaire »).

2. Variables et types primitifs

De manière générale, un programme travaille de l'information : pendant l'exécution, cette information est stockée dans la mémoire et subit des changements suivant les instructions du programme. (Relire le programme I.1 pour un exemple.) En C++, l'information est organisée par des *variables*. Chaque variable est d'un certain *type* et porte un *nom* (son *identificateur*).

Mémoire. Sur un ordinateur, l'unité élémentaire de l'information est le *bit* qui ne prend que les deux valeurs 0 et 1. La mémoire de l'ordinateur peut être vue comme une très longue suite de bits. Une séquence de huit bits consécutifs est appelée un *octet* (ou *byte* en anglais) : il peut représenter $2^8 = 256$ valeurs différentes. De manière analogue, deux octets peuvent représenter $2^{16} = 65536$ valeurs, etc. Regardons par exemple comment sont stockées deux variables `a` et `b`, de taille 2 octets disons :



À titre d'illustration, suivons l'humble vie de la variable `a` durant l'exécution de la fonction suivante :

```
int test()
{                       // Au début de ce bloc la variable a n'existe pas encore
  short int a;          // Création de la variable a: allocation de deux octets
  a= 3 + ( 4 * 5 );     // Évaluation d'une expression, puis affectation du résultat
  return a;             // Renvoyer (copier) la valeur de a  à l'instance appelante
}                       // Destruction de la variable a devenue inutile, puis retour
```

La variable `a` est créée lors de sa *définition* ; le compilateur lui réserve alors deux octets à une certaine *adresse* dans la mémoire. Le *nom* « `a` » fait ensuite référence à cette adresse. Les instructions du programme ne changent que la *valeur* stockée à cet endroit, qui, lui, ne bouge plus. Quand la variable n'est plus utilisée, à la sortie du bloc (ou fonction ou programme) où elle est définie, elle est *détruite*. La mémoire occupée jusqu'ici est rendue au recyclage, pour ensuite stocker d'autres données. Dans notre exemple la *valeur* de la variable `a` est renvoyée comme résultat de la fonction `test()` à la fonction appelante :

```
int b;                  // Création de la variable b: allocation de deux octets
b= test();              // Appel de test(), puis affectation (stockage) du résultat
```

☞ Soulignons que pendant toute sa vie, la variable `a` occupe deux octets exactement, jamais plus. La raison est simple : c'est lors de sa création qu'il faut décider de son *type*, disons `short int`, ce qui fixe en particulier la taille. *Le type reste le même durant toute la vie d'une variable !* Dans notre cas, la taille de deux octets limite forcément la capacité à 2^{16} valeurs seulement. L'interprétation des valeurs possibles et les opérations disponibles sont également définies par le type, comme discuté plus bas.

☞ Comme on le voit dans ces exemples, l'affectation suit la syntaxe `variable= expression`. La sémantique est la suivante : d'abord l'expression est évaluée, puis la valeur qui en résulte est affectée à la variable. Pour ceci le résultat de l'expression doit être du même type que la variable, sinon le compilateur essaiera d'effectuer une conversion (implicite) ou émettra un message d'erreur (en cas d'incompatibilité).

Noms. Le nom d'une variable sert à l'identifier : il permet en particulier de trouver l'emplacement dans la mémoire, d'accéder à la valeur stockée et de la modifier.

Un nom peut être constitué de lettres, de chiffres et du caractère blanc souligné « _ » (*underscore* en anglais). Un nom ne doit pas commencer par un chiffre et doit éviter les mots clés réservés du langage.

☞ Notez que le C++ fait la différence entre majuscules et minuscules : les deux noms `toto` et `Toto` ne représentent donc pas la même variable. En C++ les noms de variables peuvent être aussi longs que l'on désire, toutefois le compilateur ne tiendra compte que des 32 premiers caractères.

☞ Vous pouvez donner n'importe quel nom pourvu qu'il respecte les règles précédentes. Veillez tout de même à ce qu'un nom de variable soit en rapport avec l'utilité quelle aura dans votre programme : cela aide à une meilleure compréhension de celui-ci, surtout dans un code source long et complexe (voir §7).

Types. Le C++ est un langage *typé* : le type d'une variable détermine sa capacité (sa taille dans la mémoire et les valeurs possibles) ainsi que son comportement (les opérations disponibles dans le langage).

Une des particularités du C/C++ est que ce langage se veut proche de la machine. Il ne fournit donc que des types directement existant sur le microprocesseur. Cette approche a pour avantage une grande rapidité à l'exécution. Pour en profiter, par contre, il faut comprendre un peu comment travaille la machine.

Le C++ fournit un très petit nombre de *types primitifs*, dont la capacité et le comportement sont prédéfinis par le compilateur. Les types primitifs sont omniprésents dans la programmation en C++. On discutera par la suite chacun de ces types, sa capacité et ses opérations, ainsi que ses limitations :

`bool:` booléen, ne prend que deux valeurs : `true` (=1) et `false` (=0).
`char:` caractère (taille 1 octet typiquement), options : `signed, unsigned`
`int:` nombre entier (taille 2 ou 4 octets typiquement), options : `signed, unsigned`
`short int:` nombre entier (taille 2 octets typiquement), options : `signed, unsigned`
`long int:` nombre entier (taille 4 octets typiquement), options : `signed, unsigned`
`long long int:` nombre entier (taille 8 octets typiquement), options : `signed, unsigned`
`float:` nombre à virgule flottante simple précision (typiquement 4 octets)
`double:` nombre à virgule flottante double précision (typiquement 8 octets)
`long double:` nombre à virgule flottante triple précision (typiquement 12 octets)

☞ Afin d'éviter d'éventuelles déceptions, soyons clairs : les types du C++ sont très loin des concepts idéalisés des mathématiques. Par exemple, le type `int` ne modélise que les « petits entiers ». De manière analogue, le type `float` n'a rien à voir avec les nombres réels : la mémoire finie fixée implique que l'on ne peut stocker que des développements binaires *tronqués* ce qui produit forcément des erreurs d'arrondi.

2.1. Le type int. Le type `int` a été conçu pour modéliser les entiers (plus précisément les « petits entiers », voir la capacité plus bas). En C++ on pourrait écrire par exemple :

```
int p,q;          // définition de deux variables p et q de type int
p= 10;            // affectation d'une valeur (ici une constante)  : p vaut 10
q= 5*(p+1);       // évaluation d'une expression, puis affectation : q vaut 55
p= q;             // affectation (ici copie d'une valeur)          : p vaut 55
q= q+1;           // affectation (ici augmentation par 1)          : q vaut 56
```

Après exécution, les variables `p` et `q` ont les valeurs `55` et `56`, respectivement. Définition et affectation peuvent être combinées. Ainsi les lignes suivantes produisent le même résultat qu'avant :

```
int p(10);        // définition de p avec initialisation à la valeur 10
int q= 5*(p+1);   // définition de q et affectation de la valeur 55
p= q;             // affectation (à noter que p est déjà défini)
q= q+1;           // augmentation (à noter que q est déjà défini)
```

Capacité du type int. À cause des limitations de taille, une variable de type `int` ne peut stocker que des valeurs entre $-2^{31} = -2147483648$ et $2^{31} - 1 = 2147483647$ incluses. Il existe des variantes `short` et `long` et `long long`, dont chacune peut être `signed` (l'option par défaut) ou `unsigned`. Le tableau suivant en donne les capacités correspondantes (implémentées par GNU C++ sur un processeurs à 32 bits ; elles peuvent varier d'un compilateur à un autre, et même d'une machine à une autre).

type	taille		valeurs	plage des valeurs possibles	
	octets	bits	possibles	minimum	maximum
`unsigned short int`	2	16	2^{16}	0	65535
`signed short int`	2	16	2^{16}	-32768	32767
`unsigned int`	4	32	2^{32}	0	4294967295
`signed int`	4	32	2^{32}	-2147483648	2147483647
`unsigned long int`	4	32	2^{32}	0	4294967295
`signed long int`	4	32	2^{32}	-2147483648	2147483647
`unsigned long long int`	8	64	2^{64}	0	18446744073709551615
`signed long long int`	8	64	2^{64}	-9223372036854775808	9223372036854775807

☞ Si l'on veut faire des calculs avec des entiers plus grands, les types primitifs ne suffiront plus. Dans ce cas il faut une solution sur mesure ; on discutera une implémentation au chapitre II.

Comportement du type int. Les valeurs prises par une variable de type `int` sont interprétées comme des nombres entiers, signés ou non, suivant le schéma ci-après :

représentation binaire sur 2 octets = 16 bits	interprétation ‘`unsigned`’	interprétation ‘`signed`’
$0000000000000000_{\text{bin}}$	0_{dec}	0_{dec}
$0000000000000001_{\text{bin}}$	1_{dec}	1_{dec}
$0000000000000010_{\text{bin}}$	2_{dec}	2_{dec}
$0000000000000011_{\text{bin}}$	3_{dec}	3_{dec}
…	…	…
$0111111111111110_{\text{bin}}$	32766_{dec}	32766_{dec}
$0111111111111111_{\text{bin}}$	32767_{dec}	32767_{dec}
$1000000000000000_{\text{bin}}$	32768_{dec}	-32768_{dec}
$1000000000000001_{\text{bin}}$	32769_{dec}	-32767_{dec}
…	…	…
$1111111111111100_{\text{bin}}$	65532_{dec}	-4_{dec}
$1111111111111101_{\text{bin}}$	65533_{dec}	-3_{dec}
$1111111111111110_{\text{bin}}$	65534_{dec}	-2_{dec}
$1111111111111111_{\text{bin}}$	65535_{dec}	-1_{dec}

En C++ ce schéma correspond aux valeurs possibles d’une variable de type `short int`, aussi appelé `short` tout simplement.

Il admet deux variantes :
`unsigned short` allant de 0 à 65535, puis `signed short` allant de -32768 à 32767.

Peut-être la deuxième moitié de l’interprétation `signed` vous semble un peu arbitraire, voire bizarre. Mais elle devient très naturelle si l’on calcule modulo 2^{16} : ainsi $32768 \equiv -32768$, puis $32769 \equiv -32767$, etc … finalement $65534 \equiv -2$ et $65535 \equiv -1$.

Les types `int` et `long int` sont représentés de la même manière mais sur 4 octets, et le type `long long int` sur 8 octets.

Opérateurs arithmétiques. Les `int` admettent les opérateurs de comparaison usuels : égal `==`, différent `!=`, inférieur `<`, inférieur ou égal `<=`, supérieur `>`, supérieur ou égal `>=`. Ce sont des opérateurs binaires qui comparent deux `int` et renvoient la valeur booléenne qui en résulte. Les opérateurs arithmétiques `+`, `-`, `*` ont leur sens usuel pour les `int`, avec une exception importante : si la capacité est dépassée, seul le reste modulo 2^{32} est retenu ! (Voir la représentation interne esquissée plus haut.) Autrement dit :

☞ *Les variables de type `int` avec les opérations `+`, `-`, `*` modélisent non les entiers mais les entiers modulo* 2^{32}. ✍

Quant à la division des `int`, l’opérateur `/` calcule la partie entière du quotient, alors que l’opérateur `%` en calcule le reste. À noter donc que `17/3` vaut `5`, et `17%3` vaut `2`. Si l’on veut calculer la fraction comme une valeur à virgule flottante, expliquée plus bas, il faut d’abord transformer les deux opérants en `float` puis utiliser la division prévue pour les `float`.

Opérateurs mixtes. Au delà de l’opérateur d’affectation `=`, les opérateurs `+=`, `-=`, `*=`, `/=`, `%=` composent l’affectation avec un opérateur arithmétique : `a+=b` équivaut à `a=a+b`, puis `a-=b` équivaut à `a=a-b` etc. Pour afficher les valeurs `0,5,10,...,95,100`, par exemple, on peut ainsi écrire :

```
for( int i=0; i<=100; i+=5 ) cout << i << " ";
```

Dans de telles boucles on utilise également l’incrémentation `++` ou la décrémentation `--`. La version préfixe `++i` équivaut à `i=i+1` ou encore à `i+=1`. La version postfixe `i++` change la variable `i` de la même façon, mais renvoie l’ancienne valeur de `i` comme résultat, tandis que la version préfixe en renvoie la nouvelle. Donc `int i=5; cout << i++; cout << i;` affiche `56`, tandis que la variante `int i=5; cout << ++i; cout << i;` affiche le résultat `66`.

Exercice/P 2.1. En reprenant le programme I.1 comme modèle, écrire un programme qui lit au clavier un nombre naturel *n* et calcule la factorielle *n*!. Jusqu’à quelle valeur de *n* le calcul est-il correct ?

Exercice/P 2.2. Vous pouvez calculer et afficher les puissances successives 2^0, 2^1, 2^2, 2^3 … et ainsi déterminer vous-mêmes la capacité du type `unsigned short int` avec la boucle suivante :

```
for ( unsigned short int entier=1, bits=0; entier!=0; entier*=2, ++bits )
  cout << "2^" << bits << "=" << entier << endl;
```

La boucle s’arrête lorsque la variable `entier` vaut `0`, ce qui arrive après 16 itérations. Expliquez ce phénomène. D’une manière analogue, deviner puis vérifier le résultat du calcul suivant :

```
unsigned int p=0;       cout << "p = " << p << ", p-1 = " << p-1 << endl;
signed   int q=(p-1)/2; cout << "q = " << q << ", q+1 = " << q+1 << endl;
```

Vous pouvez ainsi déterminer *empiriquement* les limites du type `int`, et, après modification, des autres types entiers. Une implémentation de cette idée est disponible dans le fichier `limites.cc`.

2.2. Le type bool. Le type `bool` modélise les variables booléennes. Une telle variable ne peut prendre que deux valeurs : vrai (=1) ou faux (=0). Ainsi deux constantes de type `bool` sont prédéfinies : `true` et `false`. Les opérations de la logique booléenne sont réalisées par la négation `!b`, la conjonction « et » `a&&b`, et la disjonction inclusive « ou » `a||b`, où `a` et `b` sont deux expressions de type `bool` et les opérateurs renvoient à nouveau une valeur de type `bool`. On dispose aussi des comparaisons usuelles, égalité `a==b` et inégalité `a!=b`.

Le type `bool` est particulièrement important pour les instructions conditionnelles, expliquées en §3 plus bas. L'opérateur conditionnel en est une variante : il suit la syntaxe

⟨booléen⟩ `?` ⟨expression⟩ `:` ⟨alternative⟩ `;`

Si ⟨booléen⟩ est vrai, cet opérateur retourne la valeur de ⟨expression⟩, autrement il retourne la valeur de ⟨alternative⟩. Ainsi la fonction

```
int max( int a, int b ) { return ( a>=b ?  a :  b ); }
```

calcule le maximum de `a` et `b`.

Question 2.3. Déterminer les valeurs des variables booléennes après les définitions suivantes :

```
bool a= (  1 < 2 );
bool b= ( -1 > 0 );
bool c= ( a && b );
bool d= ( a || b ) != ( 3 == 4 );
```

Dans les trois premières lignes, les parenthèses ne sont pas nécessaires mais facilitent la lecture. Dans la dernière ligne les parenthèses deviennent importantes. De manière générale, si vous avez le moindre doute sur la priorité de différents opérateurs, mettez des parenthèses qui expriment ce que vous voulez dire.

Remarque 2.4. Dans une expression logique, un entier `i` est implicitement converti en `bool`. Explicitement `bool(i)` vaut `false` si `i` vaut zéro, et `true` sinon. Par exemple `!5||4` vaut `true`. Réciproquement, une valeur booléenne `b` peut être converti en un entier : `int(b)` vaut `0` si `b` vaut `false`, et `int(b)` vaut `1` si `b` vaut `true`. Ceci est fait, par exemple, pour l'affichage : `cout << false << true;` affiche `01`.

2.3. Le type char. Une variable de type `char` contient un caractère, par exemple une des lettres `a...z` ou `A...Z`, mais aussi des chiffres `0...9` ou bien d'autres symboles comme `!"#$%&()*+,-.` etc. A priori, il n'y a rien de numérique dans cette notion. Pourtant, dans un ordinateur, tout caractère est codé par un entier. Il existe ainsi une table de « traduction » entre valeur entière et caractère : le plus souvent c'est la table ASCII (*American Standard Code for Information Interchange*). Le compilateur se charge de faire la transition entre les deux formes d'écriture :

Programme I.4 Conversion entre `int` et `char` — `intchar.cc`

```
1   #include <iostream>                  // déclarer l'entrée-sortie standard
2   using namespace std;                 // accès direct aux fonctions standard
3
4   int main()
5   {
6     int i= 65;                         // 65 est le code ASCII du caractère 'A'
7     char c= char(i);                   // conversion explicite de int en char
8     cout << c << ":" << i << endl;     // affichage du caractère et de son code
9     c= 'B';                            // le caractère 'B' correspond au code 66
10    i= c;                              // conversion implicite de char en int
11    cout << c << ":" << i << endl;     // affichage du caractère et de son code
12    i= 67;                             // 67 est le code ASCII du caractère 'C'
13    cout << char(i) << ":" << i << endl; // conversion et affichage à la fois
14    c= 'D';                            // le caractère 'D' correspond au code 68
15    cout << c << ":" << int(c) << endl;  // conversion et affichage à la fois
16  }
```

Comme on le voit dans cet exemple, la conversion d'une variable `i` de type `int` dans le type `char` est assez naturelle : on écrit simplement `char(i)`. Réciproquement, `int(c)` convertit la variable `c` de `char` en `int`. La conversion est effectuée implicitement si nécessaire.

Capacité et comportement du type char. Le type `char` occupe un octet, ce qui veut dire que le code ASCII ne peut prévoir que 256 caractères (en fait seuls les 128 premiers sont standardisés). De manière interne, le type `char` est représenté comme un entier de taille 1 octet. On peut donc effectuer tous les calculs que nous avons vus plus haut pour les entiers. Par exemple la différence `'c'-'a'` correspond à l'entier 2, et `'a'+7` donne `'h'`, le 8ème caractère de l'alphabet. Logiquement il existe les deux variantes `signed char` et `unsigned char`. Bref, la seule différence entre `char` et `int`, outre la taille, est le comportement à l'entrée-sortie.

Remarquons finalement qu'en C++ une constante littérale de type `char` s'écrit comme `'a'`. Par contre une chaîne de caractères s'écrit comme `"ceci est une chaîne"`. On les a déjà utilisées pour l'affichage des messages, et on les reconsidérera au §6.2.

Exercice/P 2.5. Écrire un programme qui affiche les codes ASCII de 32 à 255 avec les caractères associés dans un joli tableau à 16 colonnes et 14 lignes. Pour le passage à la ligne après un groupe de 16 caractères, vous pouvez regarder le reste modulo 16. (Il s'agit d'un petit bricolage qui illustre un constat fondamental : bien formater la sortie peut prendre un bon moment. Vous trouvez une solution dans le fichier `ascii.cc`.)

Remarque 2.6. Comme le code ASCII fut développé aux États Unis pour les États Unis, les caractères codés par 32…127 ne contiennent que l'alphabet latin standard, sans accents ni caractères spéciaux. Vous voyez le résultat dans l'affichage de l'exercice précédent. Pour coder l'alphabet latin élargi, on se sert de la plage 128…255 (dont l'interprétation semble pourtant moins standardisée). On en a déjà profité dans nos programmes, par exemple dans les chaînes de caractères destinées à l'affichage. Pour d'autres alphabets encore, on peut utiliser *unicode*, un jeu de caractères international standardisé. La taille d'un caractère unicode est forcément plus grande, à savoir 16 bits. Consultez `www.unicode.org` pour en savoir plus.

2.4. Le type float. Les variables du type `float`, aussi appelées *nombres à virgule flottante*, ont été conçues pour modéliser des nombres réels d'une manière approchée, voir plus bas. Le programme I.5 ci-dessous en donne un exemple. Les types à virgule flottante sont toujours signés : il est donc inutile d'utiliser les mots clés `signed` et `unsigned` dans un tel cas.

Programme I.5 Solution numérique d'une équation quadratique `quadratique.cc`

```
#include <iostream>             // déclarer l'entrée-sortie standard
#include <cmath>                // déclarer les fonctions mathématiques
using namespace std;            // accès direct aux fonctions standard

int main()
{
  cout << "Entrez les coefficients de ax^2+bx+c svp:" << endl;
  double a,b,c;
  cout << "a = "; cin >> a;
  cout << "b = "; cin >> b;
  cout << "c = "; cin >> c;
  cout << "L'équation est " << a << "x^2 + " << b << "x + " << c << endl;
  double d= b*b - 4*a*c;        // NB: Un programme sérieux devrait...
  double r= sqrt(d);            // ... tester si d n'est pas négatif !
  double x1= (-b+r)/(2*a);      // ... tester si a est non nul !
  double x2= (-b-r)/(2*a);      // ... tester si a est non nul !
  cout << "Voici les deux solutions :" << endl;
  cout << "x1 = " << x1 << endl << "x2 = " << x2 << endl;
}
```

À noter que la représentation des nombres réels n'est possible qu'avec une précision (très) limitée. Dans ce genre de calculs approchés, on aura donc toujours affaire à des erreurs d'arrondi. Les résultats sont à interpréter avec la plus grande prudence !

Exercice/P 2.7. Tester le programme I.5 sur l'exemple $x^2 + 10^{10}x + 1$. Il affichera, sans honte, deux solutions qui sont manifestement fausse. C'est scandaleux ? Certes, mais le C++ n'y peut rien. Comment expliquer ce phénomène ? (Lire la suite.)

Capacité du type float. Comme pour les petits entiers du type `int`, comprendre le comportement du type `float` nécessite (malheureusement) la connaissance de la représentation interne. Le tableau suivant en donne un résumé (valable pour notre compilateur GNU C++) :

type	taille		précision	capacité (grandeurs possibles)	
	mantisse	exposant	(erreur relative)	minimum	maximum
`float`	24 bits	8 bits	$2^{-24} \approx 5 \cdot 10^{-8}$	$2^{-128} \approx 10^{-38}$	$2^{127} \approx 10^{38}$
`double`	53 bits	11 bits	$2^{-53} \approx 1 \cdot 10^{-16}$	$2^{-1024} \approx 10^{-308}$	$2^{1023} \approx 10^{308}$
`long double`	64 bits	15 bits	$2^{-64} \approx 5 \cdot 10^{-20}$	$2^{-16384} \approx 10^{-4932}$	$2^{16383} \approx 10^{4932}$

Exemple 2.8. Comme les arrondis sont un problème très fréquent, nous prenons ici le temps de le discuter un peu plus en détail. Le principe est exemplifié dans le schéma ci-après. Étant donné un nombre réel (par exemple $31,9$) on le transforme d'abord en représentation binaire. Puis la virgule est rendue flottante de sorte que tous les bits '1' soient placés à droite de la virgule ; en compensation on introduit un facteur 2^e convenable. Finalement la mantisse est tronquée à la taille prescrite, disons 24 bits pour une variable de type `float`. Seule ce développement tronqué et l'exposant sont stockés dans la mémoire :

$$\begin{aligned} 31,9_{\text{dec}} &= 11111.11100110011001100110011\ldots_{\text{bin}} \\ &= .11111111001100110011001100110011\ldots_{\text{bin}} \cdot 2^5 \\ &\approx .\underbrace{11111111001100110011001}_{\text{mantisse de 24 bits}}{}_{\text{bin}} \cdot \underbrace{2^5}_{\text{décalage}} \end{aligned}$$

Exercice/M 2.9. Si vous savez le faire, vérifiez la transformation de $31,9$ en binaire. En particulier essayez de vous convaincre que la représentation binaire est périodique, comme indiqué. Sinon, vous pouvez reprendre cet exemple au chapitre IV où l'on discutera de tels changements de base.

Comportement du type float. Une variable de type `float` occupe typiquement 4 octets, soit 32 bits, dont 24 bits pour la mantisse et 8 bits pour l'exposant. La longueur limitée de la mantisse entraîne forcément des erreurs d'arrondi. Même dans notre innocent exemple $31,9$ la représentation binaire est trop longue pour tenir entièrement dans 24 bits (en base 2 elle est périodique donc infinie). Pour en comprendre les effets possibles, souvent inattendus, tester le programme suivant puis expliquer le résultat. (Le vérifier en affichant la différence `somme-1.0`. Quel est le développement binaire de $0,1_{\text{dec}}$?)

Programme I.6 Un résultat surprenant ? `compte.cc`

```
1   #include <iostream>      // déclarer l'entrée-sortie standard
2   using namespace std;     // accès direct aux fonctions standard
3
4   int main()
5   {
6     float somme= 0.0;
7     for ( int i=1; i<=10; ++i ) somme+= 0.1;
8     cout << "La somme vaut " << somme << "." << endl;
9     if ( somme == 1.0 ) cout << "Le compte est bon." << endl;
10    else cout << "Vous êtes accusé de détournement de fonds." << endl;
11  }
```

Les nombres à virgule flottante ne représentent qu'un ensemble fini de nombres de manière exacte ; tous les autres nombres réels ne peuvent qu'être stockés de manière tronquée. Typiquement il faut s'attendre à une erreur relative de $2^{-24} \approx 0.000005\%$, ce qui n'est pas mal, mais tout de même une erreur.

L'usage naïf des nombres à virgule flottante nuit à la correction des résultats.
Consommez avec modération.

Exercice/M 2.10. Esquisser le fonctionnement de l'addition pour les nombres à virgule flottante. Dans la représentation décrite plus haut, que donne l'addition de 2^{60} et 1 ? de 1 et 2^{-60} ?

Exercice/P 2.11. Pour explorer les types `float` et `double` puis `long double` vous pouvez regarder le programme `precision.cc`. Il provoque des erreurs et détermine ainsi la longueur de la mantisse et de l'exposant.

Opérations du type float. Les variables du type `float` admettent les opérateurs de comparaison usuels : égal `==`, différent `!=`, inférieur `<`, inférieur ou égal `<=`, supérieur `>`, supérieur ou égal `>=`. Ce sont des opérateurs binaires qui comparent deux `float` et renvoient la valeur booléenne qui en résulte.

Les opérateurs arithmétiques `+`, `-`, `*`, `/` ont leur sens usuel pour les `float`. De plus, les fonctions usuels sont disponibles après inclusion du fichier en-tête `cmath`. Par exemple : `fabs`, `sqrt`, `exp`, `log`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, etc.

Exercice/P 2.12. Les erreurs d'arrondi peuvent se produire dans les calculs les plus simples, même avec des *nombres rationnels* ! Ainsi le calcul suivant effectué avec les `float` ne donne pas 0, comme il serait mathématiquement correct, mais produit une erreur d'arrondi. Le tester puis expliquer son résultat :

```
float a= 10.0 / 3.0;  // calcul exact : a vaut 10/3
float b=    a - 3.0;  //                b vaut 1/3
float c=    b * 3.0;  //                c vaut 1
float d=    c - 1.0;  //                d vaut 0
cout << d << endl;    // Que vaut le résultat approximatif ?
```

Effectuer aussi le calcul similaire `a=10.0/4.0; b=a-2.0; c=b*2.0; d=c-1.0;` Cette fois-ci le résultat est-il exact ? Comment expliquer ce phénomène ?

Exercice/P 2.13. Incroyable mais vrai : l'addition des `float` n'est même pas associative ! Pour `a=-1e30; b=1e30; c=1.0;` comparer `(a+b)+c` et `a+(b+c)`. Expliquer la différence.

Exercice/P 2.14. Écrire un programme qui lit au clavier un nombre n puis calcule la somme $\sum_{k=1}^{k=n} \frac{1}{k^2}$ de deux manières différentes : d'abord dans le sens des indices croissants, puis dans le sens inverses. Les résultats sont-ils identiques ? Comment expliquer la différence ? Quelle méthode est préférable ? (Pour varier, vous pouvez comparer les résultats calculés avec `float`, `double`, et `long double`. Même exercice avec $\sum_{k=1}^{k=n} \frac{1}{k}$.)

☞ Les erreurs d'arrondi, leur propagation dans les calculs, et les techniques pour éviter le pire, forment toute une théorie faisant partie de l'analyse numérique. Avant d'entamer une programmation numérique sérieuse, consultez un des livres spécialisés à ce sujet, et privilégiez des méthodes éprouvées au lieu de solutions ad hoc.

Conversion de type. Il est parfois nécessaire de changer explicitement le type d'une valeur, on parle alors d'une *conversion* comme déjà vu plus haut. Ainsi une valeur `f` de type `float` peut être converti en entier par `int(f)` ; le résultat est la partie entière de `f`. (À noter qu'il s'agit de *tronquer* et non *d'arrondir* la valeur.) La conversion est implicite dans une affectation comme `int i= f`.

Réciproquement, rappelons que pour deux valeurs `p` et `q` de type `int` l'expression `p/q` donne un entier, à savoir la partie entière du quotient. Si l'on veut calculer une valeur à virgule flottante approchée on écrit explicitement `float(p)/float(q)` ou bien implicitement `float(p)/q`. Dans ce cas la division de deux variables de type `float` donne un résultat de type `float`, comme souhaité.

2.5. Constantes. Comme nous le voyons dans les exemples, l'usage des *constantes littérales* est très fréquent. Une constante littérale du type `char` s'écrit entre apostrophes, par exemple `char c='a'`. Plus généralement, une chaîne de caractères, comme on l'a déjà vue pour la sortie, s'écrit entre guillemets, par exemple `cout << "Bonjour!"`. Les constantes du type `int` s'écrivent comme `83` (décimal) ou bien `0123` (octal) ou bien `0x53` (hexadécimal). Les constantes à virgule flottante sont notées comme `0.23` ou `.23` ou `2.3e-1` (de type `double` par défaut) ou bien `0.23f` (de type `float`).

Constantes nommées. Si une constante apparaît à plusieurs reprises dans le programme, il est souvent utile de définir une *constante nommée*. Pour ce faire le mot-clé `const` peut être ajouté à la déclaration d'un tel objet pour en faire une constante plutôt qu'une variable. Par exemple :

```
const int version=7;      // variable figée de type int
const float pi=3.1415f;   // variable figée de type float
```

De telles constantes peuvent être utilisées comme des variables usuelles, par exemple l'expression `2*pi*rayon` calcule $2\pi r$ avec $\pi = 3,1415$. Évidemment une constante, une fois définie, ne peut plus servir de valeur à gauche, c'est-à-dire on ne peut pas affecter de valeurs : l'instruction `pi=3` produira un message d'erreur du compilateur. (Cette restriction est la raison d'être des constantes.) Pour la même raison, une constante doit être initialisée lors de sa définition, car une affectation ultérieure est impossible. Il est donc impossible de définir une constante `const int c;` sans spécifier sa valeur.

2.6. Références. Une *référence* sur un objet introduit un synonyme, un alias de l'objet référencé. En termes de mémoire, on ne crée pas une copie, mais une référence sur l'adresse de l'objet. En termes d'identificateurs, on déclare un nouveau nom pour un vieil objet. (Si, si, ceci peut être très utile.)

```
int a= 0;          // définition d'une variable nommée a
int & b= a;        // référence nommée b sur la variable a
a= 1; cout << "a=" << a << ", b=" << b << endl;   // a=1, b=1
b= 2; cout << "a=" << a << ", b=" << b << endl;   // a=2, b=2
```

Dans l'exemple précédent on définit d'abord une variable `a` ; le compilateur lui réserve donc une partie de la mémoire à une certaine adresse. Ensuite on crée une référence qui s'appelle `b` ; ce nom fait maintenant référence à la même adresse que le nom `a`. En particulier le compilateur ne crée pas de nouvel objet, on déclare juste un alias.

☞ Le signe `&` est utilisé dans la définition du nom `b` — est dans la définition seule !. Il indique que `b` ne désigne pas une nouvelle variable mais un synonyme de la variable `a`. Après sa définition, le nom `b` s'utilise comme le nom d'une variable ordinaire. Cependant, quand vous testez ce bout de code, vous verrez que tout changement sur `b` est répercuté sur `a`, et réciproquement. Les deux se comportent alors comme un seul objet, adressable sous deux noms différents.

☞ Une référence doit être initialisée lors de sa définition : il est impossible de définir une référence `int &c;` sans spécifier la variable sur laquelle elle fait référence ! De même, une fois l'identification entre `a` et `b` est faite, on ne peut plus la changer : il s'agit effectivement d'un seul et même objet. (Que se passe-t-il par contre si l'on enlève le signe `&` ci-dessus ?)

Références constantes. Tout ce que nous venons de dire s'applique également aux constantes :

```
const int a= 10;   // définition de la constante a à valeur 10
const int &b= a;   // définition d'une référence constante sur a
```

Dans l'exemple suivant, `a` est une variable alors que `b` est constante :

```
int a;             // définition d'une variable a
const int &b= a;   // définition d'une référence constante sur a
```

C'est une situation intéressante, car on ne peut pas changer la valeur en utilisant le nom `b` ; ceci n'est possible que par le nom `a`. On réalise ainsi une restriction d'accès qui est fréquente et très utile pour les paramètres de fonctions (voir §4 plus bas).

```
const int a= 10;   // définition de la constante a à valeur 10
int &b= a;         // définition d'une référence non constante --> erreur
```

Ce dernier exemple est refusé par le compilateur. Pourquoi ?

3. Instructions conditionnelles

Les instructions conditionnelles permettent de ramifier l'exécution des instructions suivant l'état des variables. On parle ainsi de *branchement*, dont une variante est *l'itération*. Dans ce but le C++ offre les instructions suivantes :

3.1. L'instruction if. Branchement suivant un booléen. *Syntaxe :*

`if (` ⟨expression⟩ `)` ⟨instruction⟩`;`
`if (` ⟨expression⟩ `)` ⟨instruction⟩ `else` ⟨alternative⟩`;`

Sémantique : Si l'expression booléenne donne la valeur `true`, alors l'instruction suivante est exécutée. Dans la deuxième forme, la valeur `false` entraîne l'exécution de l'alternative.

Remarque 3.1. La comparaison `==` est facilement confondue avec l'affectation `=`. De telles fautes de frappe sont très courantes et elles s'avèrent en général catastrophiques pour le fonctionnement du programme ! (Heureusement un bon compilateur émettra un avertissement.) Changeons par exemple le programme I.7 en remplaçant la condition `(b==0)` par l'affectation `(b=0)`. Pourquoi le comportement du programme change-t-il radicalement ?

Question 3.2. Où est l'erreur dans le code suivant : `if x<y min=x else min=y;` Comment le rectifier ?

Question 3.3. Où est l'erreur dans le code suivant ? Comment le corriger ?

```
if ( a <= b <= c ) cout << "suite croissante" << endl;
if ( a >= b >= c ) cout << "suite décroissante" << endl;
```

Programme I.7 Division de deux entiers `division.cc`

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    cout << "\nDonnez deux entiers a et b (avec b non nul) svp : ";
7    int a, b;
8    cin >> a >> b;
9    cout << "Vous avez entré a=" << a << " et b=" << b << endl;
10   if ( b==0 ) cerr << "La division par 0 n'est pas définie\n" << endl;
11   else cout << "Leur quotient entier vaut a/b=" << (a/b) << endl
12             << "et le reste vaut a%b=" << (a%b) << endl << endl;
13 }
```

3.2. L'instruction switch. Branchement suivant un entier. *Syntaxe :*

`switch(` ⟨expression⟩ `)`
{
`case` ⟨constante_1⟩ `:` ⟨instructions_1⟩ `break;`
...
`case` ⟨constante_n⟩ `:` ⟨instructions_n⟩ `break;`
`default:` ⟨instructions par défaut⟩
}

Sémantique : D'abord l'expression est évaluée en un entier. Si cette valeur figure dans la liste des constantes, alors les instructions suivantes sont exécutées. (Elles sont typiquement terminées par `break` pour terminer et sortir du bloc.) Si la valeur n'y figure pas, les instructions qui suivent le mot clé `default` sont exécutées. Le programme I.8 en donne un exemple.

Programme I.8 Évaluation d'une expression arithmétique `switch.cc`

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    cout << endl << "Donnez une expression binaire svp : ";
7    float x, y;
8    char op;
9    cin >> x >> op >> y;
10   cout << "Calcul de " << x << op << y << " ... ";
11   switch( op )
12     {
13     case '+': cout << "résultat : " << x+y << endl; break;
14     case '-': cout << "résultat : " << x-y << endl; break;
15     case '*': cout << "résultat : " << x*y << endl; break;
16     case '/': if (y!=0) cout << "résultat : " << x/y << endl;
17               else      cerr << "division par 0" << endl;
18               break;
19     default:  cerr << "seules + - * / sont permises" << endl;
20     }
21   cout << "Au revoir.\n" << endl;
22 }
```

☞ On a tout intérêt à remplacer, dans la mesure du possible, une suite de `if` par un `switch` : en effet `switch` est plus lisible (et plus rapide à l'exécution) qu'une longue succession de `if` .

Exercice/P 3.4. Écrire une fonction `jour` qui prend comme paramètres un entier *j* compris entre 1 et 7, disons, et qui affiche le jour de la semaine : « lundi », « mardi », …, « dimanche ».

3.3. Boucle while. Itération selon la syntaxe suivante :

`while (` ⟨condition⟩ `)` ⟨instruction⟩`;`
`do` ⟨instruction⟩ `while (` ⟨condition⟩ `);`

Sémantique : L'instruction est itérée tant que la condition est vraie. Dans la première forme, la condition est testée *avant* l'exécution de l'instruction. Dans la seconde forme, la condition est testée *après* l'exécution de l'instruction, de sorte que l'itération soit exécutée au moins une fois.

3.4. Boucle for. Boucle selon la syntaxe suivante :

`for(` ⟨initialisation⟩`;` ⟨condition⟩`;` ⟨progression⟩ `)` ⟨instruction⟩`;`

Sémantique : Au début de la boucle est effectuée l'initialisation. Tant que la condition est vraie, l'instruction est exécutée, ensuite la progression est effectuée, et la boucle recommence avec le test de la condition. Ainsi la sémantique des boucles suivantes est la même :

`for(` ⟨initialisation⟩`;` ⟨condition⟩`;` ⟨progression⟩ `)` ⟨instruction⟩`;`
⟨initialisation⟩`; while(` ⟨condition⟩ `) {` ⟨instruction⟩`;` ⟨progression⟩`; }`

On pourrait donc se passer de la boucle `for`, mais son utilisation améliore souvent la lisibilité.

☞ Il est possible de définir une variable dans l'initialisation d'une boucle, par exemple
`for( int i=0; i<10; ++i ) cout << i << endl;`
Ceci a été utilisé dans le programme I.1. Par convention, la portée de la variable `i` est restreinte à la boucle.

Exemple 3.5 (Chronométrage d'une boucle)**.** Dans une boucle typique, chaque itération ne prend qu'une fraction d'une seconde, mais elle n'est pas instantanée. Ceci se fait sentir pour un grand nombre d'itérations.

Pour avoir un exemple concret, le programme `chrono.cc` chronomètre le calcul de la somme $1+2+\cdots+n$ par la formule $\frac{1}{2}n(n+1)$, puis par une boucle allant de 1 à n. Si l'on choisit par exemple $n=10^9$, la deuxième variante nécessite l'exécution d'un milliard d'itérations. Afin de vous faire une intuition sur la performance des ordinateurs, lisez puis testez ce petit programme.

Il sera instructif de chronométrer de la même manière vos programmes plus élaborés.

3.5. Les instructions break et continue. Pour gérer les boucles et les itérations il y a deux instructions supplémentaires : `break` sort immédiatement de la boucle alors que `continue` termine l'itération en cours et recommence avec la prochaine itération, en passant par la progression et la condition d'arrêt.

Comme on a vu plus haut, la commande `break` s'utilise très naturellement dans les branchements du type `switch` ; sauf exception elle y est logiquement nécessaire. Dans les boucles l'usage est plus rare, mais peut parfois faciliter la programmation (à consommer avec modération).

Exemple 3.6. Le programme suivant lit une suite d'entiers positif. Tout entier négatif est rejeté, puis la lecture continue. L'entier `0` sert à signaler la fin de la suite.

Programme I.9 Exemple de `break` et `continue` `break.cc`

```
1  #include <iostream>          // déclarer l'entrée-sortie standard
2  using namespace std;         // accès direct aux fonctions standard
3
4  int main()
5  {
6    cout << "Entrez une suite d'entiers positifs (0 pour terminer) :\n";
7    for( int i=1; ; ++i )      // Ici pas de condition d'arrêt
8      {
9        cout << "L'entier no " << i << " : ";
10       int n;                 // définition d'une variable locale
11       cin >> n;              // lecture d'une valeur du clavier
12       if ( n == 0 ) break;   // sortir immédiatement de la boucle
13       if ( n < 0 ) continue; // terminer l'itération en cours
14       cout << "L'entier no " << i << " vaut " << n << endl;
15     }
16   cout << "Au revoir\n" << endl;
17 }
```

4. Fonctions et paramètres

Pour augmenter la lisibilité d'un programme il est en général indispensable de le découper en sous-programmes, appelés *fonctions*. Nous allons voir tout au long de ce cours que cette technique apporte un réel confort dans la vie du programmeur. Ainsi il est indispensable de comprendre ce concept fondamental *en détail*, en particulier le passage des *paramètres* et le renvoie des *résultats*.

4.1. Déclaration et définition d'une fonction. En C++ déclaration et définition d'une fonction sont, respectivement, de la forme

`type_de_retour nom_de_la_fonction(` ⟨liste des paramètres⟩ `);`
`type_de_retour nom_de_la_fonction(` ⟨liste des paramètres⟩ `) {` ⟨instructions⟩ `}`

On précise ainsi au compilateur le type du résultat retourné, le nom de la fonction, ainsi que le nombre des paramètres et leur type. Voici un exemple :

Programme I.10 Exemple d'une fonction avec paramètres

```
int calcul( int x, char op, int y )
{
  switch( op )
    {
    case '+': return x+y;  // effectuer une addition
    case '-': return x-y;  // effectuer une soustraction
    case '*': return x*y;  // effectuer une multiplication
    default:  exit(1);     // erreur (sortie du programme)
    }
}
```

Les paramètres et la valeur renvoyée réalisent la communication entre la fonction et le monde extérieur : ils constituent *l'interface* de la fonction. Dans notre exemple, l'instruction `int r= calcul(2,'+',3);` appelle la fonction `calcul` et déclenche ainsi les actions suivantes :

Passage des paramètres: D'abord les paramètres sont passés à la fonction `calcul`. Pour l'appel `calcul(2,'+',3);` ceci équivaut aux définitions `int x=2; char op='+'; int y=3;` en entrée. Les paramètres reçoivent ainsi leurs valeurs dictées par *l'instance appelante*.

Exécution de la fonction: Ensuite est exécutée la fonction proprement dite. Ici on utilise les paramètres comme des variables usuelles. Leur seule particularité est qu'elles viennent d'être initialisées lors de l'appel de la fonction.

Renvoie du résultat: L'instruction `return` ⟨expression⟩`;` permet de renvoyer une valeur à l'instance appelante. À noter que ceci provoque la sortie immédiate de la fonction.

☞ Une fonction de type `void` ne retourne pas de valeur ; elle se termine donc par `return;` (facultatif) et ne peut pas comporter d'instruction `return` ⟨expression⟩. Toute autre fonction doit obligatoirement renvoyer une valeur du type spécifié.

4.2. Mode de passage des paramètres. Suivant le contexte et les besoins de l'application, les paramètres d'une fonction lui sont passés soit *par copie* soit *par référence*.

Passage par référence: Lors du passage par référence la fonction appelée reçoit une référence sur l'objet original. (Pour les références voir §2.6.) Toute modification de la variable-paramètre est effectuée sur l'original, et persiste après la sortie de la fonction. Ce mode de passage est signalé dans la liste des paramètres par le symbole `&` entre le type et le nom du paramètre.

Passage par copie: Lors du passage par copie, la fonction ne reçoit qu'une copie de l'objet original, et cette copie mène une vie indépendante. Ainsi toute modification de la variable-paramètre est effectuée sur la copie et n'est visible que dans la fonction. À la sortie de la fonction la copie est détruite et ne laisse aucune trace ; l'objet original, quant à lui, reste inchangé. (Ce mode de passage est l'option par défaut en C++.)

Programme I.11 Passage par copie vs passage par référence

```
int puissance4( int x )     // passage par copie
{
  x= x*x;       // calculer d'abord le carré...
  x= x*x;       // ... puis la puissance 4
  return x;     // renvoyer la valeur calculée
}

int incrementer( int& x )  // passage par référence
{
  int y= x;     // stocker la valeur initiale de x
  x= x+1;       // incrémenter la variable x
  return y;     // renvoyer l'ancienne valeur de x
}
```

Exercice/P 4.1. Essayez de comprendre en détail le fonctionnement du programme `passage.cc`. Que prédiriez-vous comme résultat ? Le compiler puis l'exécuter afin de vérifier votre prévision. Modifier le programme afin de varier le passage des paramètres. Quels résultats prédiriez-vous ? Les vérifier.

☞ La valeur renvoyée par une fonction est un objet d'un certain type donné. Si l'on veut qu'une fonction transmette *plusieurs valeurs* comme résultat, sans pour autant définir un type complexe à cet effet, il suffit de lui passer des paramètres par référence (c'est-à-dire modifiables) pour stocker les résultats :

```
void eudiv( const int& a, const int& b, int& q, int& r );
```

On suppose dans cet exemple que `eudiv` effectue une division euclidienne : le quotient et le reste sont stockés dans les variables `q` et `r`, nécessairement passés par référence. (Expliquer pourquoi.)

Question 4.2. Analyser les fonctions suivantes. Quel est leur résultat ? Expliquer la nécessité du signe '`&`'.

```
void swap( int& a, int& b ) { int c=a; a=b; b=c; }
void noswap( int a, int b ) { int c=a; a=b; b=c; }
```

Exercice/P 4.3. Écrire une fonction `trier` qui prend comme paramètres trois entiers a, b, c et les échange de sorte qu'à la fin on ait $a \leq b \leq c$. Choisir un mode de passage convenable : copie ou référence ?

Passage par référence constante. Pour un gros objet il est souvent plus efficace de le passer par référence que par copie. Pensez à une image numérique de plusieurs méga-octets : la création d'une copie nécessite de la mémoire et du temps non négligeables, tandis qu'une référence n'introduit qu'un synonyme sans aucun travail de copie. Cependant, d'éventuels changements seront effectués sur l'original, ce qui peut ou non être souhaitable. Pour cette raison, les paramètres peuvent être déclarés constants :

Passage par référence constante: L'argument est passé par *référence* afin d'éviter une copie inutile, et déclaré `const` pour éviter toute modification. (Le passage par référence en absence de `const` est considéré comme l'intention de modifier la variable.)

Passage par copie constante: Il est possible de passer un objet par copie tout en le déclarant `const`. Ceci combine l'inconvénient d'une copie avec la restriction d'une constante. Bien que théoriquement possible ce mode de passage n'a donc pas d'intérêt pratique.

Surcharge des fonctions. En C++ il est possible de définir des fonctions différentes portant le même nom, à condition qu'elles se distinguent par leurs paramètres. Voici un premier exemple :

```
void eudiv( const int& a, const int& b, int& q );
void eudiv( const int& a, const int& b, int& q, int& r );
```

Dans cet exemple c'est le nombre des paramètres qui diffère : le compilateur saura appeler la bonne fonction pour `eudiv(10,3,quot)` et pour `eudiv(10,3,quot,reste)` car le contexte détermine son choix. De même dans le deuxième exemple, où le type des paramètres diffère :

```
void swap( int& a, int& b ) { int c=a; a=b; b=c; }
void swap( float& a, float& b ) { float c=a; a=b; b=c; }
```

Dans les deux cas, le compilateur saura choisir la bonne fonction à partir des paramètres passés. Ceci n'est plus le cas dans l'exemple suivant, qui lui est erroné :

```
int   sqrt( int a ) { ... }
float sqrt( int a ) { ... }
```

L'intention du programmeur était sans doute de fournir deux variantes pour le calcul d'une racine carrée : la première pour calculer la partie entière, de type `int`, la deuxième pour calculer une valeur approchée, de type `float`. Pour le compilateur, par contre, il est impossible de deviner lors de l'appel `sqrt(2)` laquelle des deux fonction est souhaitée. Il refuse donc de compiler ce code.

Remarque 4.4. Vous pouvez donner des valeurs par défaut aux derniers paramètres d'une fonction. Dans ce cas, si vous appelez cette fonction avec un paramètre manquant, sa valeur par défaut sera utilisée pour l'initialisation. Par exemple, supposons que la fonction `rationnel( int numer, int denom= 1 )` construit le nombre rationnel `numer/denom`. L'appel `rationnel(3,2)` donne $\frac{3}{2}$, alors que l'appel `rationnel(3)` donne $\frac{3}{1} = 3$.

4.3. Les opérateurs. Un opérateur n'est rien d'autre qu'une fonction — avec un nom et une notation particulière. L'avantage de l'écriture sous forme d'opérateur est une meilleure lisibilité : au lieu d'écrire `add(a,b)` on préfère la notation `a+b`, au lieu de `mult(a,b)` on préfère `a*b`, et au lieu de `assign(a,b)` on préfère `a=b`. Dans ce but le C++ offre une vingtaine d'opérateurs, qui prennent un ou deux (voire trois) paramètres. Parmi les opérateurs que nous avons déjà vus, on distingue les opérateurs arithmétiques, les opérateurs d'affectation, d'incrémentation, de comparaison, de logique, et d'entrée-sortie.

Surcharge d'opérateurs. Comme vu plus haut, les opérateurs arithmétiques `+`, `-`, `*`, `/` sont définis pour les types entiers comme `short int`, `int`, `long int` etc, mais aussi pour les types numériques comme `float`, `double`, `long double` etc. Évidemment ces opérateurs sont lourdement surchargés, car ils prennent un sens différent dans chaque instance. À chaque appel le compilateur choisira l'opérateur approprié au vu du type des opérandes données. Dans tous les cas, un tel opérateur prend deux arguments du même type et renvoie un certain résultat du dit type.

Mode de passage. Regardons les opérateurs fournis par le C++ sous l'angle du passage par copie ou par référence. Comme les opérateurs arithmétiques ne changent pas les valeurs de leurs paramètres, on s'attend à une déclaration comme

```
type operator + ( type a, type b );      ou bien
type operator + ( const type& a, const type& b );
```

Il en est de même pour les autres opérateurs arithmétiques `-`, `*`, `/`, `%`. Les opérateurs de comparaison `==`, `!=`, `<`, `<=`, `>`, `>=` pourraient être déclarés de manière similaire :

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

L'opérateur d'affectation `=` prend lui aussi deux paramètres du même type, mais cette fois-ci une variable à gauche et une valeur quelconque à droite. Il ne change pas le paramètre à droite, mais il modifie bien sûr le paramètre à gauche en y affectant sa nouvelle valeur :

```
type& operator = ( type& variable, const type& valeur );
```

Il en est de même pour les variantes `+=`, `-=`, `*=`, `/=`, `%=`. Les opérateurs d'incrémentation `++` et de décrémentation `--` prennent un seul paramètre, et bien sûr ils le modifient :

```
type& operator ++ ( type& variable );
```

Ceci correspond à notre fonction `incrementer()` dans le programme I.11.

C'est ainsi que l'on devrait définir ces opérateurs. Bien sûr ils sont déjà prédéfinis par le compilateur pour les types primitifs `int`, `float`, etc. On verra plus tard, quand nous définissons nos propres types en C++, comment implémenter des opérateurs qui suivent les modèles ci-dessus. (Par exemple les grands entiers au chapitre II, les permutations au chapitre VI, ou les polynômes au chapitre XII).

4.4. Portée et visibilité. Une variable n'existe que dans le bloc dans lequel elle est définie, et ceci seulement après sa définition. On dit alors que la variable est *locale* : elle est créée lors de sa définition, et détruite lors de la sortie du bloc. La partie entre création et destruction est appelée la *portée* de la variable. Une variable définie en dehors de tout bloc est dite *globale*. Le programme I.12 en donne un exemple.

Pour un exemple un peu plus complexe, essayez de comprendre le programme I.13. Notez qu'une variable peut en cacher une autre : s'il y a deux variables de même nom et de portées imbriquées, la variable locale a priorité sur la variable globale. Tester le programme pour le vérifier.

Programme I.12 Portée des variables `portee.cc`

```
 1  int g;        // définition de la variable globale g
 2
 3  int main()
 4  {             // début du bloc de la fonction main()
 5    g = 0;      // ok : nous sommes dans la portée de g
 6    i = 1;      // erreur : nous ne sommes pas dans la portée de i
 7    int i;      // la portée de i commence par sa définition
 8    i = 2;      // ok : nous sommes maintenant dans la portée de i
 9    {           // début du sous-bloc
10      g = 3;    // ok : nous sommes toujours dans la portée de g
11      i = 4;    // ok : nous sommes toujours dans la portée de i
12      j = 5;    // erreur : nous ne sommes pas dans la portée de j
13      int j;    // la portée de j commence par sa définition
14      j = 6;    // ok : nous sommes maintenant dans la portée de j
15      i = 7;    // ok : nous sommes toujours dans la portée de i
16      g = 8;    // ok : nous sommes toujours dans la portée de g
17    };          // fin du sous-bloc et fin de la portée de j
18    j = 9;      // erreur : nous ne sommes plus dans la portée de j
19    i = 10;     // ok : nous sommes toujours dans la portée de i
20    g = 11;     // ok : nous sommes toujours dans la portée de g
21  }             // fin du bloc et fin de la portée de i
22
23  void une_autre_fonction()
24  {             // début du bloc de la fonction
25    g = 12;     // ok : nous sommes dans la portée de g
26  }             // fin du bloc de la fonction
```

Programme I.13 Visibilité des variables `visible.cc`

```
 1  #include <iostream>              // déclarer l'entrée-sortie standard
 2  using namespace std;             // accès direct aux fonctions standard
 3
 4  int i=-1, j=-2, a=0, b=0;        // définition globale de i,j,a,b
 5
 6  void affiche( int a, int b )     // définition de la fonction affiche(a,b)
 7  {                                // début du bloc et de la portée de a et b
 8    cout << a << " " << b << endl; // ici les paramètres a et b sont locaux
 9  }                                // fin du bloc et de la portée de a et b
10
11  int main()                       // définition de la fonction main()
12  {                                // début du bloc de la fonction main()
13    affiche(i,j);                  // ici i est globale et j est globale
14    for ( int i=3; i<=5; ++i )     // définition locale de la variable i
15      {                            // début du sous-bloc de la boucle
16        affiche(i,j);              // ici i est locale et j est globale
17        int j=i*i;                 // définition d'une variable locale j
18        affiche(i,j);              // ici i est locale et j est locale
19      };                           // fin du sous-bloc et de i,j locales
20    affiche(i,j);                  // ici i est globale et j est globale
21  }                                // fin du bloc de la fonction main()
```

☞ Il est possible d'adresser la variable globale en faisant précéder son nom de l'opérateur de portée ' `::` '. (Remplacer `i` par `::i` dans la ligne 16 ou 17 ou 18, par exemple.)

4.5. La fonction main. La fonction `main` est le point d'entrée de votre programme, elle est donc obligatoire et un peu particulière. Elle aussi peut prendre des paramètres, c'est-à-dire des options dans une ligne de commande : l'instance appelante est le système d'exploitation de votre ordinateur, qui passe ces options à votre logiciel. Ainsi votre programme pourra utiliser une liste de paramètres passée par le système d'exploitation et lui retourner une valeur, dit code de retour, attestant de sa bonne exécution.

Programme I.14 La fonction `main` peut prendre des paramètres `main.cc`

```
1  #include <iostream>
2  using namespace std;
3
4  int main( int nombre_de_parametres, char* liste_des_parametres[] )
5  {
6    for( int i=0; i<nombre_de_parametres; ++i )
7      cout << "paramètre " << i << " = " << liste_des_parametres[i] << endl;
8    return 0;
9  }
```

☞ Comme on le voit dans cet exemple, la fonction `main` reçoit deux paramètres : le premier est le nombre d'options, le deuxième est la liste des options, dont chacun est une chaîne de caractères.

☞ Un programme C/C++ rend toujours un code de retour de type `int`. Le système d'exploitation considérera que le programme s'est bien déroulé si la valeur de retour est nulle (la valeur par défaut). Pour toute autre valeur, le système diagnostiquera une erreur. Ce mécanisme est bien utile lorsque vous utilisez un script shell : ce dernier pourra selon la valeur de retour lancer une action adaptée.

5. Entrée et sortie

5.1. Les flots standard. Un programme doit en général communiquer avec le monde extérieur. Typiquement votre logiciel affichera des données ou des messages sur l'écran, et l'utilisateur entrera des données ou des commandes au clavier. Pour cet effet quatre flots d'entrée-sortie (*input-output streams* en anglais) sont déclarés après inclusion du fichier `iostream` :

Le flot `cin` correspond à l'entrée standard (typiquement le clavier)
Le flot `cout` correspond à la sortie standard (typiquement l'écran).
Le flot `cerr` est utilisé pour envoyer des messages d'erreur (typiquement sur l'écran)
Le flot `clog` est utilisé pour protocoler l'avancement du programme (typiquement sur l'écran)

L'opérateur `<<` permet d'envoyer (écrire) une valeur dans un flot de sortie.
L'opérateur `>>` permet d'extraire (lire) une valeur dans un flot d'entrée.

Envoyer le caractère spécial '`\n`' commence une nouvelle ligne.
Envoyer le caractère spécial '`\r`' retourne au début de la même ligne.
L'instruction `cout << flush;` vide la mémoire tampon et force l'affichage immédiat.
L'instruction `cout << endl;` vide la mémoire tampon et commence une nouvelle ligne.

5.2. Écrire dans un flot de sortie. L'inclusion du fichier en-tête `iomanip` permet de formater les flots. Dans le programme I.15 ci-dessous, par exemple, `setprecision(int n)` définit le nombre de décimales affichées, alors que `setw(int n)` (pour *set width* en anglais) définit la largeur du champ.

Programme I.15 Formater les flots avec `iomanip` `iomanip.cc`

```
1  #include <iostream>    // déclarer l'entrée-sortie standard
2  #include <iomanip>     // manipulateurs pour formater les flots
3  #include <cmath>       // fonctions mathématiques comme sqrt()
4  using namespace std;   // accès direct aux fonctions standard
5
6  int main()
7  {
8    for( int n=0; n<=100; n+=10 )
9      cout << "n=" << setw(3) << n << ", n^2=" << setw(5) << (n*n) << endl;
10   cout << "sqrt(2) = " << setprecision(10) << sqrt(2.0) << endl;
11 }
```

Exercice/P 5.1. Pour augmenter la précision d'un calcul on pourrait être tenté d'afficher plus de chiffres, par exemple comme dans le code suivant :

```
double x= sqrt(2);
cout << setprecision(50) << x << endl;
```

Trouver l'erreur logique dans cette approche. Combien de chiffres sont valables ? Pour simplifier vous pouvez remplacer `sqrt(2)` par `1.0/3.0`. Expliquer ce phénomène.

Exercice/P 5.2. Deviner puis vérifier ce que donnent les instructions suivantes :

```
int i=5; cout << i << i++ << i-1 << endl;
```

Expliquer pourquoi une telle écriture est fortement déconseillée. Trouver une écriture plus claire. ▼

5.3. Lire d'un flot d'entrée. En principe la lecture d'un flot d'entrée est aussi simple que l'écriture dans un flot de sortie. Il y a quand même une différence intrinsèque : lors de l'écriture le logiciel connaît déjà les données à afficher, mais pendant la lecture on ignore ce qui va être lu (logique, non ?). Il faut donc prévoir plusieurs cas, y inclus des entrées erronées.

Exercice/P 5.3. Écrire une fonction qui lit une suite d'entiers positifs terminée par 0 et qui retourne le maximum, le minimum, et la moyenne. Pensez aux cas limites, voire erronés : Que faire avec une entrée négative ? Que faire avec une liste vide, c'est-à-dire de longueur zéro ?

☞ La difficulté de programmer l'entrée-sortie augmente avec la complexité des données. Pour ne pas perdre trop de temps et pour procéder directement au noyau mathématique, nos projets comporteront, comme bout de code initial, une entrée-sortie prête à utiliser. C'est commode mais peu réaliste.

5.4. Écrire et lire dans les fichiers. La communication entre programme et monde extérieure peut se faire par d'autres voies que l'écran ou le clavier. Le programme I.16 ouvre le fichier `lecture.txt` pour lire une liste d'entiers. En parallèle il ouvre le fichier `ecriture.txt` pour écrire les valeurs absolues. Facile, non ? Notons que l'on peut interroger l'état d'un flot par les fonctions `eof()` – *end of file* = fin du flot, `good()` – opération réussie, `fail()` – opération échouée, `bad()` – erreur grave = flot corrompu.

Programme I.16 Lire et écrire dans des fichiers `fichiers.cc`

```
#include <iostream>     // déclarer l'entrée-sortie standard
#include <fstream>      // manipuler les flots associés aux fichiers
using namespace std;    // accès direct aux fonctions standard

int main()
{
  clog << "J'ouvre le fichier \"lecture.txt\" pour lecture ... ";
  ifstream entree("lecture.txt");   // ifstream = input file stream
  if ( !entree ) { clog << "echec." << endl; return 1; };
  clog << "ok." << endl;

  clog << "J'ouvre le fichier \"ecriture.txt\" pour écriture ... ";
  ofstream sortie("ecriture.txt");  // ofstream = output file stream
  if ( !sortie ) { clog << "echec." << endl; return 1; }
  clog << "ok." << endl;

  int nombre_in, nombre_out;
  while( !entree.eof() )
    {
      entree >> nombre_in;
      if( entree.eof() ) break;
      nombre_out= abs(nombre_in);
      sortie << nombre_out << " ";
      clog << "J'ai lu " << nombre_in << " --> j'ai écrit " << nombre_out << endl;
    }
  clog << "Je ferme les fichiers \"lecture.txt\" et \"ecriture.txt\"." << endl;
  entree.close(); sortie.close();
}
```

Ici le flot `clog` n'est utilisé que pour « protocoler » l'avancement du travail ; on pourrait aussi bien le supprimer. Mis à part ouverture et fermeture, les fichiers s'utilisent comme les flots standard `cin` et `cout`.

On pourrait imaginer que pour `cin` et `cout` l'ouverture s'effectue automatiquement lors du lancement du logiciel, ainsi que la fermeture quand le programme se termine.

Dévier l'entrée-sortie. Le programme I.16 précédent peut être réécrit comme suit :

Programme I.17 Lire et écrire les flots standard `devier.cc`

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

int main()
{
  int nombre_in, nombre_out;
  while( !cin.eof() )
    {
      cin >> nombre_in;
      if( cin.eof() ) break;
      nombre_out= abs(nombre_in);
      cout << nombre_out << " ";
      clog << "J'ai lu " << nombre_in << " --> j'ai écrit " << nombre_out << endl;
    }
}
```

Par défaut `cin` lit du clavier et `cout` écrit sur l'écran. On peut néanmoins dévier ces flots standard en appelant le programme comme suit :

```
a.out < input.txt
```

Ainsi le flot d'entrée `cin` lit le fichier `input.txt`. Si celui-ci contient le texte `+1 -2 +3 -4 +5 -6`, par exemple, alors le programme affiche `1 2 3 4 5 6` sur l'écran. Bien sûr, on peut aussi dévier la sortie :

```
a.out > output.txt
```

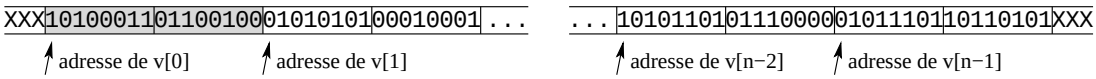
Dans ce cas le flot de sortie `cout` écrit dans le fichier `output.txt`. (Son ancien contenu est écrasé ; à utiliser avec prudence.) On peut finalement dévier entrée et sortie à la fois :

```
a.out < input.txt > output.txt
```

Ainsi notre programme transforme les données du fichier d'entrée `input.txt` et écrit le résultat dans le fichier de sortie `sortie.txt`. Génial, non ?

6. Tableaux

Il arrive fréquemment que l'on veuille stocker une famille de données de même type, par exemple une suite finie de nombres, une liste de noms, d'adresses, etc. Pour cela le C/C++ prévoit comme construction primitive les *tableaux*. Un tableau de longueur n est une famille $v_0, v_1, \ldots, v_{n-1}$ de n variables du même type, indexées par $0, 1, \ldots, n-1$. Sur ordinateur ceci se réalise par n variables stockées à n adresses consécutives dans la mémoire. Pour chacune il faut réserver la mémoire nécessaire (selon le type) :



Au lieu des tableaux primitifs du C/C++, il sera plus commode et plus sûr d'utiliser des implémentations sophistiquées comme les vecteurs ou les chaînes de caractères. Le principe est le même, mais ces classes offrent plus de confort et de fonctionnalité.

6.1. La classe vector. La bibliothèque STL (*Standard Template Library*) fournit la classe générique `vector` avec une fonctionnalité assez naturelle : un vecteur d'entiers de type `int` est défini par

```
vector<int> mon_vecteur;
```

Jusqu'ici c'est un vecteur vide, de longueur 0. Sa taille peut être adaptée par

```
mon_vecteur.resize(10);
```

Pour définir un vecteur et spécifier sa taille, on utilise la définition

```
vector<int> mon_vecteur(10);
```

On détermine la taille par la fonction `mon_vecteur.size()` et on accède à l'élément numéro `i` par `mon_vecteur[i]`. Le programme I.18 illustre d'autres opérations disponibles.

Programme I.18 Exemple d'utilisation de la classe `vector` — `vector.cc`

```
#include <iostream>              // déclarer l'entrée-sortie standard
#include <vector>                // définir la classe générique vector
using namespace std;             // accès direct aux fonctions standard

ostream& operator << ( ostream& out, const vector<int>& vec )
{
  out << "( ";                          // parenthèse ouvrante pour faire joli
  for ( size_t i=0; i<vec.size(); ++i ) // boucle parcourant les indices
    out << vec[i] << " ";               // affichage de la valeur vec[i]
  out << ")";                           // parenthèse fermante pour faire joli
  return out;                           // on rend le flot comme il se doit
}

int main()
{
  cout << "\nVoici quelques opérations sur des vecteurs:\n" << endl;
  vector<int> mon_vecteur(5);    // définir un vecteur de longueur 5
  cout << "initialisation(5) : " << mon_vecteur << endl;
  mon_vecteur.resize(20,99);     // rallonger le vecteur en remplissant par 99
  cout << "adaptation(20,99) : " << mon_vecteur << endl;
  mon_vecteur.resize(10);        // raccourcir le vecteur à la longueur 10
  cout << "adaptation(10) :    " << mon_vecteur << endl;
  for ( int i=0; i<10; ++i )     // boucle parcourant les indices du vecteur
    mon_vecteur[i]= 10+i;        // affecter une valeur à la place indexée i
  cout << "affectation :       " << mon_vecteur << endl;
  vector<int> une_copie;         // définir un deuxième vecteur (encore vide)
  cout << "un vecteur vide :   " << une_copie << endl;
  une_copie= mon_vecteur;        // affectation (par recopiage des éléments)
  cout << "copie du vecteur :  " << une_copie << endl << endl;
  mon_vecteur.push_back(90);     // rajouter la valeur 90 à la fin
  cout << "prolongation :      " << mon_vecteur << endl;
  mon_vecteur.pop_back();        // raccourcir en effaçant la dernière place
  cout << "troncature :        " << mon_vecteur << endl;
  mon_vecteur.clear();           // effacer le vecteur tout entier
  cout << "délétion complète : " << mon_vecteur << endl << endl;
  cout << "copie retenue :     " << une_copie << endl << endl;
}
```

De la même façon on utilise un vecteur de n'importe quel autre type, comme `vector<bool>` ou `vector<char>` ou `vector<double>`. Pour cette raison on appelle la classe `vector` une *classe générique* (ou *template* en anglais, ou *patron de classe*). À noter que la classe `vector` n'est définie qu'après inclusion du fichier en-tête correspondant par la directive `#include <vector>`.

☞ La classe générique `vector` de la STL ne fournit pas d'opérateurs d'entrée-sortie. Ceci n'est pas bien grave : pour afficher un vecteur du type `vector<int>` on pourrait aisément écrire une fonction

```
void affiche( ostream& out, const vector<int>& vec );
```

Nous préférons un *opérateur* qui poursuit le même but. Il est implémenté et utilisé dans le programme I.18. Rappelons à ce propos qu'un opérateur n'est rien d'autre qu'une fonction avec une écriture particulière.

☞ Le fichier `vectorio.cc` implémente l'entrée-sortie des vecteurs d'une manière un peu plus générale. Essayez de comprendre son fonctionnement.

Attention aux indices ! Avant d'utiliser un vecteur on doit obligatoirement spécifier sa taille et ainsi réserver la mémoire nécessaire. Le compilateur ne peut pas deviner quels indices seront utilisés durant l'exécution du programme. Or, en dehors des indices légitimes résérvés on accéderait aux données voisines, qui appartiennent à d'autres variables voire d'autres programmes.

Pour cette raison la lecture d'une case non définie donne des résultats erronés, et l'écriture peut détruire des données irréparablement. Un tel logiciel se compile sans problème, mais lors de l'exécution il s'arrêtera brutalement signalant une *erreur de segmentation*. Cette situation est, hélas, assez fréquente. En particulier on se trompe facilement du dernier indice : un vecteur de taille 10 n'a pas d'élément indexé par 10 !

☞ *Un vecteur de taille n est toujours indexé par* $0, 1, \ldots, n-1$.
Lire un élément d'indice illégitime donne des résultats imprévisibles ;
son écriture risque d'écraser d'importantes données stockées à cet endroit ! ✍

6.2. La classe string. Très souvent, même dans les petits programmes, on doit manipuler les chaînes de caractères. La classe `string` de la bibliothèque standard permet de ce faire avec la plus grande facilité. Elle peut être vue comme une spécialisation des tableaux, avec une fonctionnalité sur mesure pour les chaînes de caractères. Sa déclaration se fait par la directive `#include <string>`.

Une variable du type `string` peut être définie par

```
string s;
```

On peut affecter une constante littérale par

```
s= "un exemple";
```

L'opérateur `+` est surchargé et réalise la concaténation :

```
s= "voici " + s + " !";
```

Après ces instructions `s` contient le texte « `voici un exemple !` ». On accède à la *n*ième lettre par `s[n]`, c'est-à-dire on utilise l'indexation connue des vecteurs. Par exemple, `s[0]` vaut `'v'`, et l'affectation `s[0]= 'V'` redéfinit la première lettre, de sorte que `s` contienne le texte « `Voici un exemple !` ». La classe `string` offre beaucoup d'autres fonctions, dont le programme I.19 ne montre que quelques exemples.

Programme I.19 Exemple d'utilisation de chaînes de caractères `string.cc`

```
#include <iostream>        // déclarer l'entrée-sortie standard
#include <string>          // déclarer la classe string
using namespace std;       // accès direct aux fonctions standard

int main()
{
  cout << "\nBonjour et bienvenue.\nComment t'appelles-tu ? ";
  string nom, s;           // définition de deux chaînes (encore vides)
  cin >> nom;              // lire un mot (délimité par des espaces)
  cout << "Salut " << nom << " !" << endl;
  getline(cin,s);          // effacer le reste de la ligne d'entrée

  string t("Ceci est une chaîne de caractères.");
  cout << t << endl;       // la chaîne de caractères initiale
  cout << string(t,9,14) << t.size() << " caractères.\n";
  cout << t << endl;       // la chaîne de caractères inchangée
  s= t + " On peut y rajouter du texte.";
  cout << s << endl;       // la chaîne de caractères rallongée
  s.insert(54,"ou insérer ou glisser ");
  cout << s << endl;       // la chaîne de caractères après insertion
  s.replace(68,7,"remplacer");
  cout << s << endl;       // la chaîne de caractères après remplacement
  s.erase(65,13);
  cout << s << endl;       // la chaîne de caractères après délétion

  cout << "Retape la permière phrase stp : " << endl;
  getline(cin,s);          // lire toute une ligne, espaces inclus
  cout << "Tu as entré la phrase\n\"" << s << "\"" << endl;
  if ( s == t ) cout << "Sans aucune faute de frappe, bravo !" << endl;
  else cout << "Ceci n'est pas la phrase\n\"" << t << "\"" << endl;
  cout << "première sous-chaîne \"ou\" : pos=" << s.find("ou") << endl;
  cout << "dernière sous-chaîne \"ou\" : pos=" << s.rfind("ou") << endl;
  cout << "première ponctuation : pos=" << s.find_first_of(".,:;!?") << endl;
  cout << "dernière ponctuation : pos=" << s.find_last_of(".,:;!?") << endl;
  cout << "Au revoir, " << nom << ".\n" << endl;
}
```

Conversion entre string et flot. Il sera parfois commode de lire ou d'écrire dans une chaîne de caractères comme on écrit ou lit dans un flot. Ceci est possible comme illustrés dans le programme suivant : `istringstream` prend une chaîne de caractères et en fait un flot d'entrée (*input string stream*), alors que `ostringstream` fournit un flot de sortie qui écrit dans une chaîne de caractères (*output string stream*).

Programme I.20 Conversion entre string et flot `sstream.cc`

```
#include <iostream>            // entrée-sortie standard
#include <sstream>             // conversion entre string et flot
using namespace std;           // accès direct aux fonctions standard

int main()
{
  // Définir la chaîne à analyser, puis lire ses termes
  string t= " test 12.5 + 87.5 ";
  cout << "string initial : \"" << t << "\"" << endl;
  istringstream iss(t);
  string mot; double x,y; char op;
  iss >> mot >> x >> op >> y;

  // Afficher les termes, puis les écrire dans une chaîne de caractères
  cout << mot << endl << x << endl << op << endl << y << endl;
  ostringstream oss;
  oss << " ### " << mot << " ### " << x << " ### " << op << " ### " << y << " ### ";
  string s= oss.str();
  cout << "string terminal : \"" << s << "\"" << endl;
}
```

7. Quelques conseils de rédaction

Le code source d'un bon programme sera très souvent relu, analysé, modifié, corrigé, élargi, amélioré, réutilisé, etc. Pour cette raison une bonne structuration et des commentaires détaillés sont indispensable. Ils seront le meilleur guide pour tout futur lecteur, et ne serait-ce que pour le programmeur lui-même, qui essaie de retrouver le sens de son programme écrit quelques mois auparavant…

⇒ Il convient de bien présenter le code source et de le commenter sans retenue.

⇒ Il ne faut surtout pas croire que le seul lecteur sera le compilateur, au contraire !

⇒ Lors de la rédaction d'un programme il faut écrire pour l'homme et non pour la machine.

Contempler à ce propos le mot savant de Knuth cité au début de ce chapitre.

7.1. Un exemple affreux. Le langage C++ se prête facilement à la programmation de code obscure. Regardons le programme I.21, qui est correct mais humainement incompréhensible. Le compilateur C++ accepte ce code sans aucun problème et en produit un logiciel exécutable.

Programme I.21 Un programme incompréhensible — à éviter ! `obscure.cc`

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

int main()
{
  const int a=10000; int b=52514; vector<int> c(b); int d=0, e=0, f, g, h;
  for( ; (f=b-=14); d=1, cout << setw(4) << setfill('0') << g+e/a << flush )
    for( g=e%=a; (h=--f*2); e/=h ) e=e*f+a*( d ? c[f] : a/5 ), c[f]=e%--h;
}
```

Pouvez-vous deviner ce que fait ce logiciel ? Si l'on vous disait que ces trois lignes calculent 15000 chiffres de π, seriez-vous convaincu ? Feriez-vous confiance en un tel programme ? Seriez-vous capable de calculer ainsi 20000 chiffres de π ? La réponse est non, non, non, et non !

Ces questions correspondent d'ailleurs à des critères de qualité très importantes : l'utilisabilité, la compréhensibilité, la vérifiabilité, et la réutilisablitilé. Pour ces raisons il faut à tout prix éviter de code source comme ci-dessus : le programme I.21 n'est qu'une curiosité pour amuser les étudiants, mais pour tout autre objectif il est inutilisable.

Remarque 7.1. Le code source du programme I.21 est une adaptation en C++ d'un programme de J. Arndt et C. Haenel, donné dans leur livre π *unleashed*, Springer-Verlag, Berlin 2001. Cette méthode de calculer π a été découverte par S. Rabinowitz en 1991. On l'expliquera au chapitre IV, où l'on développe aussi une preuve de sa correction. Vous serez ainsi capable de l'implémenter d'une façon plus digne.

7.2. Le bon usage. Pour obtenir des programmes compréhensibles, même lors de la création de programmes assez brefs, il faut respecter certaines règles. Les remarques suivantes sont loin d'être exhaustives ; elles essaient simplement d'anticiper quelques difficultés et mauvaises habitudes répandues. Même si ce baratin ne vous parle pas trop lors de la première lecture, il serait bon de le relire de temps en temps.

Utiliser des noms de variables et fonctions qui aient un sens. Vous pouvez opter pour des noms courts ou longs. Il est préférable d'utiliser un nom court (une unique lettre par exemple) pour un compteur de boucle, ou un autre usage relativement ponctuel. Par contre si votre variable doit être utilisée en divers points de votre programme, il est alors préférable d'opter pour un nom plus long et surtout plus explicite. En effet, si à la seule lecture de la variable on sait à quoi elle correspond, cela sera une aide précieuse.

Commenter soigneusement tout fichier source. Dans les exemples présentés ici, qui se veulent didactiques, les commentaires expliquent le fonctionnement du langage C++. Dans un programme réel, par contre, de telles explications seraient inappropriées : on supposera que le lecteur envisagé connaît déjà le langage, inutile alors de lui expliquer « ceci est un commentaire » ou « cela est une variable ». Il faut, par contre, expliquer tout ce qui n'est pas immédiatement évident.

Commenter toute fonction et tout bloc sémantique. Pour ne pas encombrer le corps d'une fonction avec de longs commentaires, on peut écrire un commentaire détaillé immédiatement avant la fonction : quelles sont les données d'entrée et de sortie ? les hypothèses ? les garanties ? les limitations ? la méthode utilisée ? Pour des implémentations plus complexes il est avantageux de se référer à une documentation externe, par exemple « voir Knuth §5.3.1 », ou une spécification comme « voir ISO C++ 14882 : §27.3 ». Ensuite des commentaires courts d'une ligne suffiront dans le corps, en renvoyant éventuellement aux plus amples explications précédentes.

Créer une documentation du programme indépendante des fichiers sources. Afin de garder à jour, *dans un seul fichier*, et le code source et la documentation, il existe des systèmes de documentation comme Doxygen (`www.doxygen.org`) ou JavaDoc (`java.sun.com/javadoc`). Ils produisent une documentation à partir des commentaires du code source.

Optimiser la lisibilité du code source. Pour cela il est souhaitable de bien formater le fichier source :

- Ne pas écrire plus d'une instruction par ligne (sauf raisons contraires).
- Mettre les accolades ouvrantes et fermantes seules sur une ligne.
- Indenter correctement (ceci est automatique sous `emacs` avec la touche de tabulation).
- Laisser des lignes blanches afin de regrouper les blocs sémantiques.

☞ Bien sûr des exceptions à ce format sont possibles, et différents styles se sont établis. L'essentiel est d'en choisir un et de l'utiliser d'une manière cohérente. (Dans ces notes je n'ai pas toujours respecté ces règles, dans le souci d'une meilleure mise en page. Ne faites donc pas comme je fais, faites plutôt comme je dis. `;-)`

Relire votre code source. Plusieurs fois. Avec du recul. Pour un projet sérieux il faut soigneusement vérifier chaque fonction : le code écrit traduit-il bien votre intention ? Peut-il être plus clair ? plus efficace ? Vérifier, améliorer, rerédigez en même temps les commentaires qui l'accompagnent.

Concluons par le méta-conseil formulé par Bjarne Stroustrup :

☞ *« Ne suivez les conseils que lorsqu'ils vous semblent logiques.*
Il n'existe pas de substitut à l'intelligence,
pas plus qu'à l'expérience, au bon sens et au bon goût. » ✍

Spécifier chaque fonction et son interface. Quand vous introduisez une fonction, même courte, prenez soin de spécifier pour quel usage elle est faite, quelle donnée elle requiert dans chaque paramètre, et quelle(s) donnée(s) elle renvoie. Mettez le tout dans un joli commentaire :

```
//-----------------------------------------------------------------------
// Calcul du coefficient binomial
// Entrée : deux entiers n et k
// Sortie : renvoie le coefficient binomial (n,k)
//
// Pour rappel : le coefficient binomial (n,k) est le nombre
// des sous-ensembles de cardinal k d'un ensemble de cardinal n.
// On suppose donc 0 <= k <= n, et la fonction renvoie 0 sinon.
//
// On utilise ici le type Integer qui modélise les entiers,
// positifs ou négatifs ou nuls, sans restriction de taille.
//-----------------------------------------------------------------------
Integer binomial( Integer n, Integer k )
{ ... }
```

L'utilisation de la fonction est ainsi clarifiée. En s'y conformant, le fonctionnement interne sera relativement facile à corriger, adapter, ou optimiser : il suffira de modifier convenablement le code de la fonction (voir chapitre II, §1.3). Ceci est un changement *local* et sans aucun risque, pourvu que la fonction et tous ses utilisateurs respectent la spécification.

☞ Lors de la conception d'un programme non trivial, la spécification et l'interface de chaque fonction doivent être claires et précises dès le début. Il est assez pénible, dans un état avancé du projet, de chercher puis modifier tous les usages d'une fonction, dispersés dans le code. Ce genre de changements *globaux* est coûteux et provoque souvent des erreurs — à éviter.

Introduire des fonctions auxiliaires. Si les mêmes actions apparaissent à plusieurs endroits dans votre programme, songez à en faire une fonction. De manière semblable, si une fonction devient trop longue et difficile à comprendre, essayez de la couper en sous-problèmes de taille modérée (si possible ni trop petits ni trop grands). Ainsi on encapsule des tâches bien précises et les rend accessibles aux vérifications et tests séparés. De plus, munies de commentaires comme ci-dessus, la lecture sera plus aisée qu'un long enchaînement d'instructions sans structure.

Finalement, pour un plus grand programme, il est recommandable de couper le fichier source en plusieurs modules (sous-programmes dans des fichiers séparés) de taille convenable, qui regroupent certaines familles de fonctions appartenant à un thème commun. Ceci a été fait, par exemple, pour la bibliothèque standard, qui est regroupée par thème : `iostream`, `iomanip`, `fstream`, `cmath`, `vector`, `string`, etc.

Introduire des variables auxiliaires. Si votre programme calcule deux fois la même quantité, songez à introduire une variable auxiliaire afin d'y stocker la précieuse valeur intermédiaire. Ceci devient obligatoire si le calcul est coûteux. Voici un mauvais exemple :

```
cout << "le résultat vaut " << calcul_long_et_laborieux(a,b,c) << endl;
if ( calcul_long_et_laborieux(a,b,c) > 0 )
  cout << "ce résultat est positif" << endl;
```

Supposons que votre fonction longue et laborieuse nécessite une heure pour calculer la réponse souhaitée. Alors le programme ci-dessus y mettra *deux* heures ! Il va sans dire que la deuxième heure est un gaspillage injustifiable. Utilisez plutôt la variante suivante :

```
int resultat= calcul_long_et_laborieux(a,b,c);
cout << "le résultat vaut " << resultat << endl;
if ( resultat > 0 ) cout << "ce résultat est positif" << endl;
```

Vérifier les résultats. Ne vous fiez jamais aveuglement à un programme, même si c'est le vôtre ! Chaque fois que cela vous est possible, recoupez les résultats intermédiaires ou finaux de votre programme avec des résultats dont vous êtes sûr. Ces tests de bon sens, souvent peu chers, détectent parfois des erreurs cachées qui autrement se montreraient seulement plus tard, et souvent de manière plus vicieuse.

8. Exercices supplémentaires

8.1. Exercices divers. Les exercices suivants sont à peu près dans l'ordre de difficulté croissante.

Exercice/P 8.1. Écrire un programme qui lit au clavier une durée en secondes et la convertit en jours, heures, minutes et secondes. Même exercice pour la conversion réciproque.

Exercice/P 8.2. Compléter le programme I.5 esquissé plus haut pour qu'il résolve correctement *tous* les cas possibles d'une équation de degré ≤ 2.

Exercice 8.3. Trouver les solutions $a,b,c \in \mathbb{N}$ des équations $ab \equiv 1 \pmod{c}$, $bc \equiv 1 \pmod{a}$, $ac \equiv 1 \pmod{b}$. En quoi l'ordinateur peut-il y être utile ? Trouve-t-il de solutions ? Dans quelle mesure peut-il résoudre le problème ? Formuler une conclusion précise des expériences sur ordinateur.

Exercice/P 8.4. Soit $f : \mathbb{N} \to \mathbb{N}$ définie par $f(n) = n/2$ si n est pair, et $f(n) = 3n+1$ si n est impair. Écrire un programme qui lit un entier positif n et affiche les termes successifs de la suite récurrente définie par $u_0 = n$ et $u_{k+1} = f(u_k)$. Empiriquement on observe qu'une telle suite arrive toujours à la valeur 1, indépendamment de la valeur initiale n. Après quelques essais, on pourrait conjecturer que *toute* valeur initiale $n \in \mathbb{N}$ mène à $f^k(n) = 1$ pour un certain rang k. Cette conjecture, dite *conjecture de Syracuse*, reste toujours ouverte. Voilà une question qui est facile à formuler mais incroyablement difficile à résoudre.

Exercice/P 8.5. Écrire une fonction qui calcule l'angle entre deux coordonnées sphériques. Ajouter un programme qui lit au clavier deux coordonnées géographiques et affiche leur distance sur terre. Par exemple, quelle est la distance entre $45°N, 5°E$ et $50°N, 8°E$?
Remarque. — Vous pouvez développer les formules nécessaires vous-mêmes. Essayez ensuite d'écrire une entrée-sortie confortable, qui vérifie aussi la validité des données entrées.

8.2. Un jeu de devinette.

Exercice/P 8.6. Écrire un programme qui réalise le jeu « trop petit, trop grand » : le logiciel choisit un nombre aléatoire que l'utilisateur doit deviner. L'utilisateur propose un nombre et le programme répond « trouvé » ou « trop petit » ou « trop grand », puis on réitère si nécessaire. ▼

Programme I.22 Début d'un programme de devinette `devinette0.cc`

```
#include <iostream>             // déclarer l'entrée-sortie standard
#include <cstdlib>              // pour déclarer random() et srandom()
#include <ctime>                // pour déclarer time()
using namespace std;            // accès direct aux fonctions standard

int main()
{
  cout << "\nBienvenue au jeu \"trop petit, trop grand\" !" << endl;
  cout << "\nEntrez un nombre maximal svp : ";
  int max; cin >> max;
  srandom( time(NULL) );      // initialiser le générateur aléatoire
  int secret= random()%max+1; // produire un nombre aléatoire entre 1 et max
  cout << "J'ai choisi un nombre secret entre 1 et " << max << "." << endl;

  // *** compléter le programme en implémentant ici les règles du jeu.
  cout << "Mon nombre secret vaut " << secret << "." << endl;
}
```

Exercice/P 8.7. En reprenant le jeu précédent, écrire un programme qui devine un nombre choisi par l'utilisateur. Celui-ci répond, toujours honnêtement, en tapant 't' ou 'p' ou 'g', respectivement.

8.3. Calendrier grégorien.

Exercice/P 8.8. Écrire un programme qui lit au clavier une date du type `j/m/a`, teste sa validité et affiche le jour de la semaine ainsi que le nombre des jours qui se sont écoulés depuis le `01/01/0001`, date aussi commode que fictive. Combien de jours avez-vous aujourd'hui ?

Réciproquement, écrire une fonction `date(int n, int& j, int& m, int& a)` qui calcule la date `j/m/a` à partir de son numéro `n`, cumulatif depuis le `01/01/0001`. (Expliquer le signe `&` dans la liste des paramètres.) Quand fêterez-vous votre 10000ème journiversaire ? ▼

8.4. Vecteurs.

Exercice/P 8.9. Écrire une fonction `inverse` qui prend comme paramètre un vecteur, passé par référence, et qui le remplace par le vecteur dans l'ordre inverse. La tester avec un programme qui lit au clavier une suite d'entiers positifs terminée par 0 et qui affiche la suite inverse. *Indication.* — On pourrait utiliser une fonction `swap(a,b)` qui échange les valeurs des variables `a` et `b`.

Exercice/P 8.10. Écrire une fonction `pascal` qui prend un vecteur $(v_0, v_1, \ldots, v_n)$, passé par référence, et le remplace par le vecteur $(v_0, v_0+v_1, \ldots, v_{n-1}+v_n, v_n)$. *Attention.* — Le nouveau vecteur est plus grand. Vaut-il mieux le calculer de gauche à droite ou de droite à gauche ? Testez votre fonction par un programme qui engendre ainsi successivement les lignes du triangle de Pascal.

Exercice/P 8.11. Pour un vecteur d'entiers $v = (v_1, \ldots, v_n)$ on définit le vecteur dérivé $v' = (v'_1, \ldots, v'_n)$ par $v'_1 = |v_1 - v_n|$ et $v'_i = |v_i - v_{i-1}|$ pour $i = 2, \ldots, n$. Écrire une fonction `deriver` qui prend comme argument un vecteur v, passé par référence, et le remplace par le vecteur v'. Le tester par un programme qui lit au clavier un vecteur v puis affiche les dérivés $v, v', v'', \ldots, v^{(k)}$ jusqu'à $k = 20$ disons.

Exercice/M 8.12. En jouant avec le programme de l'exercice précédent, on tombe sur des observations inattendues : pour un vecteur (v_1, v_2, v_3, v_4) donné, arrive-t-on toujours au vecteur nul ? Même question pour un vecteur de longueur $n = 2^k$. Comment expliquer ce phénomène ? Que se passe-t-il pour un vecteur de longueur $n \neq 2^k$? Tombe-t-on toujours sur le même cycle ? Que vaut la période en fonction de n ?

Exercice 8.13. Voici un challenge : écrire un programme qui lit une liste d'entiers et détermine la médiane. (Définir d'abord ce que c'est.) Il vous faudra éventuellement une méthode de tri, ce qui sera l'objectif du chapitre V. Bien-sûr, la question devient triviale dès que la liste est triée. Voyez-vous une autre méthode ?

8.5. Chaînes de caractères.

Exercice/P 8.14. Écrire une fonction `palindrome` qui prend comme paramètre une chaîne de caractères et détermine s'il s'agit d'un palindrome. Par exemple, « anna » est un palindrome, « nana » ne l'est pas. Quel mode de passage convient le mieux : par copie, par référence ou par référence sur une constante ? Tester votre fonction avec un programme qui lit un mot au clavier puis affiche son verdict.

Exercice/P 8.15. Écrire une fonction `anagramme` qui prend comme paramètres deux chaînes de caractères et qui détermine si les deux forment un anagramme. Par exemple « marie » et « aimer » forment un anagramme, mais non « tester » et « rester ». Quel mode de passage convient le mieux ? Tester votre fonction avec un programme qui lit deux mots au clavier puis affiche s'il s'agit d'un anagramme ou non.

Exercice/P 8.16. Pour un minimum de sécurité on exige qu'un mot de passe ait entre 6 et 8 caractères, qu'il contienne au moins une lettre et au moins un chiffre. Écrire une fonction qui teste si ces conditions sont vérifiées. (Si vous voulez, rajoutez qu'au moins un des caractères soit ni lettre ni chiffre.) Écrire un programme qui lit au clavier un mot de passe proposé par l'utilisateur et qui lui rappelle les règles si nécessaire.

Exercice/P 8.17. Écrire une fonction `minuscule` qui prend comme paramètre une chaîne de caractères, passée par référence, et qui convertit tout en lettres minuscules. Écrire une fonction analogue `majuscule`.

8.6. Topologie du plan. Voici un challenge plus osé, si vraiment vous vous ennuyez. Cet exercice vous propose d'analyser puis de programmer quelques questions de la topologie du plan. Tout sera affine par morceaux pour éviter d'éventuelles pathologies sauvages, mais surtout pour être accessible au calcul sur ordinateur. Malgré son apparence élémentaire, les questions mathématiques soulevées sont assez intéressantes.

Deux points $p, q \in \mathbb{R}^2$ définissent un *segment* $[p,q] \subset \mathbb{R}^2$ que l'on peut paramétrer par la fonction $s\colon [0,1] \to \mathbb{R}^2$, $s(t) = (1-t)p + tq$. Plus généralement, toute famille $C = (p_0, p_1, \ldots, p_n)$ de points $p_k \in \mathbb{R}^2$ spécifie une *courbe polygonale* $c\colon [0,n] \to \mathbb{R}^2$ définie par $c(k+t) = (1-t)p_k + tp_{k+1}$ pour $k = 0, 1, \ldots, n-1$

et $0 \leq t \leq 1$. C'est une application continue, affine sur chaque intervalle $[k, k+1]$, réliant les points donnés $c(k) = p_k$. Son image est donc la réunion des segments $[p_0, p_1]$, $[p_1, p_2]$, …, $[p_{n-1}, p_n]$. Dans la suite on supposera que la courbe c est *fermée* dans le sens que le point de départ $c(0) = p_0$ et le point d'arrivée $c(n) = p_n$ coïncident.

Vous pouvez imaginez les fonctions C++ suivantes dans un logiciel de dessin. Pour simplifier nous n'allons regarder que la partie calculatoire. La liste C peut être implémentée comme un vecteur de longueur $2n+2$, à coefficients entiers de type `int`, ou un type flottant si vous ne craignez pas des erreurs d'arrondi. Pour l'entrée-sortie des vecteurs, vous pouvez vous servir du fichier `vectorio.cc`.

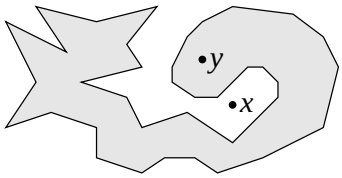


FIG. 1. Un polygone dans le plan : qui est « in », qui est « out » ?

Voici quelques tâches que l'on voudrait implémenter ; vous pouvez en rajouter d'autres :

(1) Déterminer si le polygone est simple, c'est-à-dire les segments ne se recoupent pas.

(2) Déterminer la longueur du polygone, puis l'aire de la région qu'il borde.

(3) Déterminer si le polygone est convex (ou étoilé, mais c'est plus difficile).

(4) Déterminer si globalement le parcours du polygone tourne à gauche ou à droite.

(5) Déterminer si un point donné $x \in \mathbb{R}^2$ est à l'intérieur ou à l'extérieur du polygone.

Pour ces questions il faudra d'abord préciser soigneusement ce que cela veut dire. Ensuite la solution mathématique n'est pas toujours évidente, mais peut-être vous y trouvez un certain plaisir de rechercher. Soyez assuré, il existe des solutions élégantes et efficaces…

Quelques réponses et indications

[Exercice 5.2, affichage tordu] À la surprise générale, les instructions

```
int i=5; cout << i << i++ << i-1 << endl;
```

affichent `654` et non `555`. Pour comprendre ce résultat, il faut savoir que l'opérateur de sortie est évalué de droite à gauche, donc contrairement au sens de lecture. (Ne me démandez pas pourquoi ; c'est juste une convention.) Il renvoie comme résultat une référence sur le flot de sortie, ce qui permet d'enchaîner plusieurs opérateurs. Le parenthésage implicite équivaut à l'écriture explicite

```
int i=5; (((cout << i) << i++) << i-1) << endl;
```

Cette écriture, bien que plus précise, reste peu lisible. Il est préférable de la couper en plusieurs instructions, dont chacune est facilement compréhensible.

[Exercices 8.6 et 8.7, devinette « trop petit, trop grand »] Ce jeu a deux facettes intéressantes : quant à l'apprentissage du C++ il oblige à réfléchir sur la communication entre utilisateur et logiciel. Quant à l'aspect algorithmique, c'est une préparation ludique à la recherche dichotomique (discutée au chapitre V). Si vous avez le temps, ne vous privez pas de ce plaisir ; sinon, vous trouvez une proposition de solution dans les programmes `devinette.cc` et `deviner.cc`.

[Exercices 8.8, calendrier grégorien] Cet exercice nécessite un peu de préparation. Précisons que l'on applique ici uniformément les règles du calendrier grégorien, bien qu'elles ne prissent effet que le 15 octobre 1582. Tout d'abord, comment déterminer si une année donnée est bissextile ? (C'est cette règle raffinée qui est due au Pape Grégoire, d'où le nom.) Ensuite, comment calculer le nombre des jours entre le `01/01/0001` et le `01/01/a`, début de l'année courante ? (Pour ceci il faut appliquer la règle des années bissextiles. À titre d'exemple, si le `01/01/0001` est le jour numéro `1`, alors le `01/01/2001` est le jour numéro `730486`.) Finalement, comment calculer le nombre des jours écoulés pendant l'année courante, c'est-à-dire du `01/01/a` au `j/m/a` ?

On pourra écrire une fonction `int numero( int jour, int mois, int annee )` qui réalise ce calcul et renvoie `0` si la date n'est pas valide. Pour le calcul réciproque on pourra écrire une fonction `void date( int num, int& jour, int& mois, int& annee )`. Si après réflexion vous ne trouvez pas de meilleure solution, vous pouvez regarder le fichier `gregorien.cc`. Bonne lecture !

Rien n'est fait
quand il reste à faire.

PROJET I

Tester la conjecture d'Euler concernant $x^4+y^4+z^4=w^4$

Objectifs

- Résoudre une question mathématique par une énumération exhaustive.
- Reconnaître, puis contourner, des problèmes typiques des types `int` et `double`.
- Comprendre les limitations mathématiques et informatiques inhérentes d'une telle approche.

Le problème et son histoire. Comme vous savez, l'équation de Pythagore $x^2+y^2=z^2$ admet une infinité de solutions $(x,y,z)\in\mathbb{Z}_+^3$ telles que x,y,z soient premiers entre eux. La plus petite est $(3,4,5)$, puis on a $(5,12,13)$, et on sait même produire toutes les solutions de manière systématique.

En 1769 Euler montra que l'équation $x^3+y^3=z^3$, par contre, n'admet aucune solution $(x,y,z)\in\mathbb{Z}_+^3$. Il s'agit du premier cas du grand théorème de Fermat, qui dit que $x^n+y^n=z^n$ avec $n\geq 3$ n'admet pas de solutions $(x,y,z)\in\mathbb{Z}_+^3$. Après sa découverte, Euler conjectura que $x^4+y^4+z^4=w^4$ n'admet pas de solutions $(x,y,z,w)\in\mathbb{Z}_+^4$, et plus généralement que $z_1^n+z_2^n+\cdots+z_{n-1}^n=z_n^n$ avec $n\geq 3$ n'admet pas de solutions dans $\mathbb{Z}_+^n$. Il venait d'établir le cas $n=3$. Il a fallu deux siècles environ pour en connaître la réponse pour $n=5$ (L.J. Lander et T.R. Parkin en 1966), puis pour $n=4$ (N.D. Elkies et R. Frye en 1988).

☞ Pour ne pas gâcher le plaisir de la découverte, ne cherchez pas tout de suite la réponse sur internet.

1. Préparation : calcul d'une racine

Afin de programmer ce problème, il sera utile de disposer des fonctions $a\mapsto a^4$ et $a\mapsto\lfloor\sqrt[4]{a}\rfloor$. Le type en C++ à utiliser pour les entiers sera nommé `Integer` dans la suite. Pour introduire cet alias en C++, il suffit d'écrire `typedef int Integer;` au début de votre programme. Ceci définit le type `Integer` comme synonyme avec `int`. Vous n'utilisez ensuite que le type `Integer` ; si jamais vous voulez changer de `int` à `long long int` il suffit de mettre à jour la ligne `typedef`.

Exercice/P 1.1. Écrire une fonction `Integer puissance4( const Integer& a )` qui calcule a^4 avec deux multiplications seulement. Pour quelle plage de paramètres votre fonction sera-t-elle correcte ? Expliquer le signe '`&`' devant le paramètre. Quelles alternatives conviennent ici ?

Réciproquement on cherche une fonction `racine4` qui calcule $\lfloor\sqrt[4]{a}\rfloor$, c'est-à-dire l'unique nombre naturel r tel que $r^4\leq a<(r+1)^4$. Pour ce faire deux méthodes naïves viennent à l'esprit : la première est correcte mais lente, la deuxième est rapide mais fausse :

Exercice/P 1.2. Écrire une fonction `Integer racine4lente_mais_correcte( const Integer& a )` qui calcule $\lfloor\sqrt[4]{a}\rfloor$ par une boucle parcourant $r=0,1,2,\ldots$. Justifier votre fonction dans les commentaires. À quelle plage de paramètres s'applique-t-elle ? Expliquer en quoi elle est inefficace quand r est grand. (Revoir le chronométrage de l'exemple 3.5. On discutera une nette amélioration au projet III.)

Exercice/P 1.3. Dans un monde idéal, sans erreur d'arrondi, on utiliserait sans hésitation

```
Integer racine4rapide_mais_fausse( const Integer& a )
  { return Integer( exp( log(double(a))/4 ) ); }
```

Vérifier les résultats de `racine4rapide_mais_fausse( puissance4(a) )` pour quelques petites valeurs de `a`. Lesquels sont corrects ? Expliquez ce phénomène. (Revoir §2.4.) Pourquoi et dans quels cas une petite erreur d'arrondi peut-elle entraîner une grosse erreur dans le résultat final ?

Comme la valeur cherchée est un entier, il est hors de question d'accepter une quelconque imprécision dans le résultat : on exige la valeur exacte ! Voici une façon de s'en tirer :

Exercice/P 1.4. En tenant compte des deux exercices précédents, écrire une fonction `racine4` qui soit à la fois *correcte* et *efficace* et *claire*. À partir de la valeur initiale `r = racine4rapide_mais_fausse(a)`, on corrige `r`, le cas échéant, afin de *garantir* l'inégalité $r^4 \leq a < (r+1)^4$: tant que r est trop petit on l'augmente, tant que r est trop grand on le diminue. Implémentez cette idée, et justifiez la correction dans les commentaires. À quelle plage de paramètres votre fonction s'applique-t-elle ?

2. Énumération exhaustive

On se propose de tester la conjecture d'Euler dans la mesure du possible.

Exercice/M 2.1 (préparation)**.** D'abord pour $n=4$ on essaiera une recherche exhaustive de toutes les solutions $z_1^4+z_2^4+z_3^4=z_4^4$ avec la restriction $1 \leq z_1 \leq z_2 \leq z_3 \leq N$. Jusqu'à quel N peut-on aller avec le type `int` ? `long long int` ? Montrer que le nombre des cas à traiter est $\binom{N+2}{3}$. Est-ce un polynôme en N ? de quel degré ? Est-il réaliste de chercher toutes les solutions jusqu'à $N=10^2$? $N=10^3$? $N=10^4$? $N=10^5$? $N=10^6$? Spécifier une borne N qui vous semble raisonnable.

Exercice/P 2.2 (implémentation)**.** Écrire une fonction `void euler4( Integer min, Integer max )` qui affiche toutes les solutions $z_1^4+z_2^4+z_3^4=z_4^4$ avec $1 \leq z_1 \leq z_2 \leq z_3$ et $\min \leq z_3 \leq \max$. Naïvement on pourrait penser à 4 boucles imbriquées, mais la fonction `racine4` permet de se ramener à 3 boucles. (Le projet V expliquera comment les réduire à 2 boucles seulement en utilisant des méthodes de tri.)

☞ *Étapes intermédiaires :* Il sera utile que le programme affiche de temps en temps l'avancement du travail. (Pas trop souvent car l'affichage, lui aussi, prend du temps. . ..) On pourrait ainsi écrire

```
cout << "en train de tester " << ... << "\r" << flush;
```

De même il sera intéressant d'afficher toute solution dès qu'elle est trouvée :

```
cout << "solution trouvée :  " << ... << endl;
```

L'affichage de ces messages permettra, le cas échéant, d'interrompre le programme avec la touche `CTRL c` sans perdre pour autant toute l'information.

Exercice/P 2.3 (généralisation)**.** Refaire le développement précédent pour l'équation $z_1^5+z_2^5+z_3^5+z_4^5=z_5^5$ avec la restriction $1 \leq z_1 \leq z_2 \leq z_3 \leq z_4 \leq N$. Jusqu'à quel N peut-on aller avec le type `int` ? `long long int` ? Combien de cas doivent être traités ? Est-ce un polynôme en N ? de quel degré ? Cette approche est-elle réaliste pour $N=300$? $N=1000$? $N=3000$? $N=10000$? Écrire des fonctions `puissance5` et `racine5` puis une fonction `euler5`.

Exercice/P 2.4 (consolidation)**.** Quand votre programme semble marcher, vérifiez à nouveau sa correction logique (non seulement syntaxique : c'est déjà fait par le compilateur). Explicitez dans les commentaires une spécification pour chaque fonction, en précisant à quelle plage de paramètres elle s'applique. Essayez de justifier que chaque fonction satisfait sa spécification, en remontant de l'élémentaire au complexe (*bottom-up*). Certifiez-vous finalement que votre logiciel est 100% fiable ?

☞ *Rédaction finale :* Après s'être convaincu de sa correction, nettoyer le code source et rédiger la version finale des commentaires. Commencer par quelques lignes de commentaires comportant le nom du programme, l'auteur, la date, ainsi qu'une brève description. Essayer de produire un code qui soit à la fois correct et efficace, concis et compréhensible. Relire à ce propos les conseils du §7.

Exercice/P 2.5 (conclusion)**.** Exécutez la version finale de votre logiciel et rédigez-en un protocole. Combien de temps prend-il ? Quel en est le résultat ? Formulez soigneusement une conclusion : que peut-on dire des conjectures d'Euler ? Quelles questions sont laissées en suspense ?

☞ *Compilation finale :* Afin d'optimiser un peu, compilez avec `g++ -O3 -static`.

- L'option `-O3` demande au compilateur d'optimiser la traduction en langage machine : la compilation sera plus complexe et plus lente, mais l'exécution sera un peu plus rapide.
- L'option `-static` fait inclure les bibliothèques d'une manière dite « statique » : une copie est collée à votre logiciel, ce qui augmente sa taille mais l'accélère un peu.

Pour tester empiriquement ces différentes options, vous pouvez comparer la taille et la rapidité des logiciels qui en résultent. Pour savoir plus sur les options d'optimisation, consultez le manuel en ligne de g++ en tapant `info gcc Invoking Optimize` dans une fenêtre xterm.