Une courte introduction à C++

Karl Tombre
École des Mines de Nancy

Version 1.0
Octobre 1999

1 Un peu d'histoire

Le langage C++ a deux grands ancêtres :

- Simula, dont la première version a été conçue en 1967. C'est le premier langage qui introduit les principaux concepts de la programmation objet. Probablement parce qu'il était en avance sur son temps, il n'a pas connu à l'époque le succès qu'il aurait mérité, mais il a eu cependant une influence considérable sur l'évolution de la programmation objet.

  Développé par une équipe de chercheurs norvégiens, Simula-67 est le successeur de Simula I, lui-même inspiré d'Algol 60. Conçu d'abord à des fins de modélisation de systèmes physiques, en recherche nucléaire notamment, Simula I est devenu un langage spécialisé pour traiter des problèmes de simulation. Ses concepteurs faisaient aussi partie du groupe de travail IFIP [1] qui poursuivait les travaux ayant donné naissance à Algol 60. Simula-67 est avec Pascal et Algol 68 un des trois langages issus des différentes voies explorées au sein de ce groupe. Son nom fut changé en Simula en 1986.

  Comme son prédécesseur Simula I, Simula permet de traiter les problèmes de simulation. En particulier, un objet est considéré comme un programme actif autonome, pouvant communiquer et se synchroniser avec d'autres objets. C'est aussi un langage de programmation général, reprenant les constructions de la programmation modulaire introduites par Algol 60. Il y ajoute les notions de *classe*, d'*héritage* et autorise le *masquage* des méthodes, ce qui en fait un véritable langage à objets.
- Le langage C a été conçu en 1972 aux laboratoires *Bell Labs*. C'est un langage structuré et modulaire, dans la philosophie générale de la famille Algol. Mais c'est aussi un langage proche du système, qui a notamment permis l'écriture et le portage du système Unix. Par conséquent, la programmation orientée système s'effectue de manière particulièrement aisée en C, et on peut en particulier accéder directement aux fonctionnalités du noyau Unix.

  C possède un jeu très riche d'opérateurs, ce qui permet l'accès à la quasi-totalité des ressources de la machine. On peut par exemple faire de l'adressage indirect ou utiliser des opérateurs d'incrémentation ou de décalage. On peut aussi préciser qu'on souhaite implanter une variable dans un registre. En conséquence, on peut écrire des programmes presque aussi efficaces qu'en langage d'assemblage, tout en programmant de manière structurée.

Le concepteur de C++, Bjarne Stroustrup, qui travaillait également aux *Bell Labs*, désirait ajouter au langage C les classes de Simula. Après plusieurs versions préliminaires, le langage a trouvé une première forme stable en 1983, et a très rapidement connu un vif succès dans le monde industriel. Mais ce n'est qu'assez récemment que le langage a trouvé sa forme définitive, confirmée par une norme.

C++ peut être considéré comme un successeur de C. Tout en gardant les points forts de ce langage, il corrige certains points faibles et permet l'abstraction de données. De plus, il permet la programmation objet.

D'autres langages, et en particulier Java, se sont fortement inspirés de la syntaxe de C++. Celle-ci est de ce fait devenue une référence. Nous supposons en particulier que les élèves qui ont déjà appris Java ne seront pas dépaysés par ce langage. Cependant, nous voulons mettre en garde contre plusieurs fausses ressemblances : si la syntaxe est la même ou très proche, plusieurs concepts sous-jacents sont différents. Nous nous efforcerons de signaler ces pièges potentiels.

2 Types de base et constantes

En C++, les *types de base* sont :

- `bool` : booléen [2], peut valoir `true` ou `false`,
- `char` : caractère (en général 8 bits), qui peuvent aussi être déclarés explicitement signés (`signed char`) ou non signés (`unsigned char`),
- `int` : entier (16 ou 32 bits, suivant les machines), qui possède les variantes `short [int]` et `long [int]`, tous trois pouvant également être declarés non signés (`unsigned`),
- `float` : réel (1 mot machine),

1. *International Federation for Information Processing*.

2. La présence d'un type booléen explicite est assez récente ; auparavant, les entiers étaient interprétés comme des booléens suivant leur valeur nulle ou non-nulle, et par compatibilité C++ continue à accepter des valeurs entières à la place de valeurs booléennes.

- `double` : réel en double précision (2 mots machines), et sa variante `long double` (3 ou 4 mots machine),
- `void` qui spécifie un ensemble vide de valeurs.

Les *constantes* caractères s'écrivent entre quotes simples :

```
'a'  'G'  '3'  '*' '['
```

Certains caractères de contrôle s'écrivent par des séquences prédéfinies ou par leur code octal ou hexadécimal, comme par exemple :

```
\n  \t  \r  \135  \'  \x0FF
```

Les constantes entières peuvent s'écrire en notations décimale, hexadécimale (précédées de `0x`[3]) ou octale (précédées de `0`[4]). Pour forcer la constante à être de type entier long, il faut ajouter un `L` à la fin, de même le suffixe `u` indique une constante non signée :

```
12  -43  85  18642  54L 255u 38ul
0xabfb  0x25D3a  0x3a
0321  07215  01526
```

Les constantes réelles s'écrivent avec point décimal et éventuellement en notation exponentielle :

```
532.652  -286.34  12.73
52e+4  42.63E-12  -28.15e4
```

Les constantes de type chaînes de caractères (voir plus loin) s'écrivent entre double-quotes :

```
"Home sweet home"
"Français, je vous ai compris."
```

3 Opérateurs et expressions

C++ offre un jeu très étendu d'opérateurs, ce qui permet l'écriture d'une grande variété d'expressions. Un principe général est que *toute expression retourne une valeur*. On peut donc utiliser le résultat de l'évaluation d'une expression comme partie d'une autre expression. De plus, le parenthésage permet de forcer l'ordre d'évaluation.

Les opérateurs disponibles sont les suivants :

3.1 Opérateurs arithmétiques

`+` addition
`-` soustraction
`*` multiplication
`/` division (entière ou réelle)
`%` modulo (sur les entiers)

3.2 Opérateurs relationnels

`> >= <= <` comparaisons
`== !=` égalité et inégalité
`!` négation (opérateur unaire)
`&&` ET relationnel
`||` OU relationnel

3. zéro-X.
4. zéro.

3.3 L'affectation

`=` affectation

Il faut bien noter que le signe `=` est l'opérateur d'affectation, et non de comparaison ; cela prête parfois à confusion, et entraîne des erreurs difficiles à discerner. À noter aussi que l'affectation est une expression comme une autre, c'est-à-dire qu'elle retourne une valeur. Il est donc possible d'écrire :

```
a = b = c+2;
```

ceci revenant à affecter à `b` le résultat de l'évaluation de `c+2`, puis à `a` le résultat de l'affectation `b = c+2`, c'est-à-dire la valeur qu'on a donnée à `b`. Remarquez l'ordre d'évaluation de la droite vers la gauche.

3.4 Opérateurs d'incrémentation et de décrémentation

`++` incrémentation
`--` décrémentation

Ces opérateurs, qui ne peuvent être appliqués que sur les types scalaires, peuvent s'employer de deux manières : en principe, s'ils préfixent une variable, celle-ci sera incrémentée (ou décrémentée) avant utilisation dans le reste de l'expression ; s'ils la postfixent, elle ne sera modifiée qu'après utilisation. Ainsi :

```
a = 5; b = 6;
c = ++a - b;
```

donnera à `c` la valeur 0, alors que

```
a = 5; b = 6;
c = a++ - b;
```

lui donnera la valeur -1.

Faites cependant attention dans les expressions un peu complexes où on réutilise la même variable plusieurs fois : l'ordre d'évaluation n'est pas garanti, et l'expression peut donc avoir des résultats différents suivant la machine utilisée. Par exemple, le résultat de l'expression suivante est indéfini :

```
t[++a] = a;
```

3.5 Opérateurs logiques

Ce sont les opérateurs permettant d'effectuer des opérations au niveau des bits (masquages).

`&` AND. Exemple : `a & 0x000F` extrait les 4 bits de poids faible de `a`.
`|` OR. Ainsi, `b = b | 0x100` met à 1 le 9ème bit de `b`.
`^` XOR.
`<<` SHIFT à gauche. `a = b << 2` met dans `a` la valeur de `b` où tous les bits ont été décalés de 2 positions vers la gauche.
`>>` SHIFT à droite.
`~` complément à 1 (opérateur unaire).

3.6 Modifier la valeur d'une variable

Nous avons déjà vu l'affectation, l'incrémentation et la décrémentation. Il arrive très souvent qu'on calcule la nouvelle valeur d'une variable en fonction de son ancienne valeur. C++ fournit pour cela un jeu d'opérateurs combinés, de la forme

<*variable*> <*op*>= <*expr*>

où <*op*> est un opérateur. Une telle expression est équivalente à l'expression :

<*variable*> = <*variable*> <*op*> <*expr*>

`+=` `a += b` équivaut à `a = a + b;` — À noter : `a++` $\Longleftrightarrow$ `a += 1` $\Longleftrightarrow$ `a = a + 1`
`-=` *idem*, de même que `*=`, `/=`, `%=`, «=, »=, `&=`, `|=` et `^=`.

3.7 Expressions conditionnelles

`expr1? expr2 : expr3`
est évaluée de la manière suivante :

si `expr1` *alors* `expr2`
sinon `expr3`
fsi

Cela est pratique par exemple pour calculer le maximum de 2 nombres sans passer par une fonction :

```
z = (a > b) ? a : b;
```

Cette construction pourrait bien sûr s'exprimer avec une structure conditionnelle de la forme *si–alors–sinon*, mais l'écriture sous forme d'expression conditionnelle est plus compacte.

3.8 Conversions de types

On désire souvent changer le type du résultat retourné par une expression. Pour cela existe le mécanisme de *cast*[5] :

(<*nom de type*>) `expression`
retourne une valeur dont le type est celui qui est indiqué dans la première parenthèse, et qui est obtenue en convertissant le résultat de l'expression dans le type spécifié.

Pour finir ce paragraphe, notons aussi que l'appel à une fonction est une expression comme une autre. Enfin, une expression peut dans certains cas être une suite de plusieurs expressions indépendantes séparées par des virgules ; voir à cet égard ce qui sera dit par la suite sur la structure itérative par exemple (cf. § 4.3).

Nous donnons ci-dessous un tableau récapitulatif des opérateurs de C++, classés dans l'ordre décroissant des priorités. Certains de ces opérateurs n'ont pas été mentionnés ci-dessus, mais sont décrits dans la suite du polycopié.

1	Fonction/Sélection/Portée	`() [] . -> ::`
2	Unaire	`* & - ! ~ ++ -- typeid sizeof` *casts* `new delete`
3	Multiplicatif	`* / %`
4	Additif	`+ -`
5	Décalages	`<< >>`
6	Relationnels	`< > <= >=`
7	Inégalité/Egalité	`== !=`
8	ET logique	`&`
9	XOR logique	`^`
10	OU logique	`|`
11	ET relationnel	`&&`
12	OU relationnel	`||`
13	Affectation	`=` <*op*>`=`
14	Conditionnel	`? :`
15	Exceptions	`throw`
16	Virgule	`,`

5. Dans ce polycopié, nous expliquons l'ancien système de conversion de types, qui est encore largement en vigueur. Il faut néanmoins savoir qu'un nouveau mécanisme de *cast*, utilisant les opérateurs `static_cast`, `dynamic_cast`, `const_cast` et `reinterpret_cast`, a officiellement remplacé l'ancien système dans la norme définitive de C++. Cet ancien système reste néanmoins utilisable, et la plupart des programmes existants l'emploient encore largement.

4 Structures d'un programme C++

Contrairement à Java, toutes les fonctions ne sont pas incluses dans une classe en C++. En ce sens, C++ hérite de son prédécesseur une structure modulaire, et on peut très bien concevoir un programme C++ composé d'un grand nombre de *modules*, éventuellement compilés séparément. Chaque module est alors composé de *fonctions*, et éventuellement de déclarations de variables *globales*. Dans l'ensemble des modules, une fonction particulière, ayant pour nom `main()`, doit obligatoirement exister, et de manière unique. On l'appelle souvent le *programme principal*, par abus de langage. Il serait sûrement plus correct de dire que c'est le point d'entrée à l'exécution du programme.

Ceci étant dit, il est fortement conseillé de ne pas multiplier les fonctions hors classe ; dans bien des cas, seule la fonction `main`, et éventuellement quelques fonctions annexes à des fins utilitaires, ont vocation à être définies hors d'une structuration en classes. De même, nous déconseillons *fortement* l'emploi de variables globales ; comme en Java, il est beaucoup plus judicieux, lorsque cela est nécessaire, d'utiliser des variables de classe regroupées dans une classe *ad hoc*.

Chaque fonction a la syntaxe suivante :

```
typeRetour nomDeLaFonction(spécification des paramètres formels)

{
   suite de déclarations de variables locales et d'instructions
}
```

Les paramètres formels doivent être séparés par des virgules, et sont typés.

Précisons ces notions en voyant une petite fonction :

```
int moyenne(int a, int b)
{
  int c = (a+b)/2;
  return c;
}
```

Remarque : comme en Java, on peut passer à la fonction `main` des paramètres correspondant aux paramètres d'appel du programme.

4.1 Instructions et blocs

Chaque instruction est terminée par un point-virgule. À noter que le point-virgule est une *terminaison* d'instruction et non un séparateur d'instruction. En particulier, pour qu'une expression soit considérée comme une instruction, elle doit être terminée par un `;` même si elle est la dernière d'un bloc.

Un *bloc* est une suite d'instructions délimitées par une accolade ouvrante `{` et une accolade fermante `}`. À l'intérieur de tout bloc, on peut aussi définir des variables *locales* à ce bloc :

```
if (n > 0) {
   int cumul = 0;
   for (int i=0 ; i < n ; i++) ....
   ....
   }
```

Attention à l'instruction vide — `;` — qui est source potentielle d'erreurs difficiles à détecter, comme dans :

```
/* Exemple d'une instruction vide involontaire */
 for ( ... ) ;   // Ici le point-virgule indique une instruction vide
                 // à exécuter à chaque itération ; ce n'était pas
                 // forcément le souhait du programmeur
```

Vous avez peut-être remarqué que j'ai lâchement profité de l'occasion pour introduire les deux types de commentaires valides en C++. Les portions de code comprises entre `/*` et `*/` sont des commentaires, de même que celles comprises entre `//` et la fin de la ligne.

4.2 Structures conditionnelles

La *condition* s'exprime de la manière suivante :

```
  if (<expression>) <instruction-1>
 [else <instruction-2>  ]
```

où l'exécution de la branche *alors* ou de la branche *sinon* va dépendre de l'évaluation de <*expression*> : si le résultat est vrai, on exécutera <*instruction-1*>, sinon on effectuera <*instruction-2*>. De manière tout à fait classique, s'il y a plusieurs instructions dans la partie *alors* ou la partie *sinon*, on mettra un bloc.

Quand il y a plusieurs conditions imbriquées et qu'il y a ambiguïté sur un `else`, on le rattache au `if` le plus proche.

Une autre instruction conditionnelle se comporte comme un branchement calculé. Par conséquent, il ne faut *surtout pas oublier* de mettre les `break` aux endroits nécessaires :

```
switch (<expression>) {
       case <constante-1> : <suite d'instructions> break;
       case <constante-2> : <suite d'instructions> break;
       ...
       case <constante-n> : <suite d'instructions> break;
       default : <suite d'instructions>
       }
```

Si on ne met pas de `break`, l'exécution va continuer à la suite au lieu de sortir du `switch`, puisque les différentes constantes correspondent seulement à des étiquettes de branchement. Il y a parfois des cas où c'est l'effet souhaité ; mais il faut être prudent !

4.3 Structures itératives

Plusieurs structures itératives existent en C++. Voici la première :

```
while (<expression>)
      <instruction>
```

la partie <*instruction*> pouvant bien sûr être un bloc. C'est la structure *tant-que* classique.

Une autre structure itérative est la suivante :

```
for (<expr1> ; <expr2> ; <expr3>)
    <instruction>
```

où <*expr1*>, <*expr2*> et <*expr3*> sont des expressions.

Souvenez-vous qu'une expression peut aussi être une suite d'expressions séparées par des virgules. C'est dans cette structure que cela est le plus utilisé. Cette construction est équivalente à :

```
<expr1>;
while (<expr2>) {
      <instruction>;
      <expr3>;
}
```

Résumons en disant que <*expr1*> indique l'initialisation avant entrée dans la boucle, <*expr2*> est la condition de poursuite de l'itération, et <*expr3*> est la partie qu'on effectue à la fin de chaque itération.

Une ou plusieurs de ces expressions peuvent être vides ; en particulier :

```
   for ( ; ; )
```

est une boucle infinie !

Une dernière variante de la structure itérative est :

```
do
   <instruction>
while (<expression>);
```

qui permet d'effectuer l'instruction (ou le bloc) une première fois avant le premier test sur la condition d'arrêt.

Nous avons déjà vu l'emploi de `break` dans les structures conditionnelles. En fait, `break` permet plus généralement de sortir prématurément et proprement d'une structure de contrôle. Ainsi, on peut l'utiliser également dans une itération pour sortir sans passer par la condition d'arrêt. Donnons en exemple une boucle qui lit un caractère en entrée (par une fonction `getchar()`) et qui s'arrête sur la lecture du caractère `'&'` :

```
for ( ; ; ) if ((c = getchar()) == '&') break;
```

Cette fonction peut bien sûr s'écrire plus simplement :

```
while ((c = getchar()) != '&') ; // le point-virgule ici est
                                 // l'instruction vide !
```

Une autre instruction particulière qui peut être utile dans les itérations est `continue`, qui permet de se rebrancher prématurément en début d'itération.

Enfin, signalons que C++ permet aussi de faire `goto` ; mais comme nous sommes des informaticiens bien élevés qui ne disent jamais de gros mots, nous n'en parlerons pas...

5 Fonctions et variables

Théoriquement, *toute fonction retourne une valeur*, qui peut être utilisée ou non. Toutefois, un type particulier, `void`, permet d'indiquer qu'une fonction ne retourne pas de valeur, ou plutôt que la valeur retournée ne doit pas être prise en compte.

Le passage de paramètres peut se faire par valeur ou par référence. Le passage d'une référence se note par le caractère &. En voici un exemple avec une fonction qui échange les valeurs de deux variables :

```
void swap(int& a, int& b)
{
    int tmp = a; a = b; b = tmp;
}
...
int x, y;
...
swap(x, y);
```

Une référence peut également être déclarée constante, par exemple pour passer la référence d'un objet de grande taille, tout en interdisant l'accès en écriture dans la fonction. Avec un passage par valeur, l'objet serait dupliqué dans la pile d'exécution. En supposant l'existence d'un type `Matrice` décrivant une matrice, on peut par exemple écrire :

```
void print(const Matrice& m)
{
    // le compilateur interdit toute tentative
    // de modification de la variable m dans
    // le corps de la fonction print
}
```

Une fonction peut être déclarée *en ligne*, comme dans l'exemple suivant :

```
inline int max(int x, int y) { return (x > y ? x : y); }
```

La qualification `inline` indique au compilateur qu'il est préférable de remplacer chaque appel à la fonction par le code correspondant. Cette qualification n'est qu'indicative, et n'est en particulier pas prise en compte si elle est irréalisable, c'est-à-dire si le compilateur a besoin de connaître l'adresse de la fonction.

Une fonction peut également être *surchargée* ; la discrimination est alors faite sur le nombre et le type des paramètres effectifs.

Les *variables* d'un programme C++ peuvent avoir plusieurs classes de stockage :

automatiques : c'est l'option par défaut pour toute variable interne d'une fonction. L'allocation se fait dans la pile d'exécution.

externes ou globaux : ce sont les variables définies à l'extérieur de toute fonction, et qui sont donc globales. Si on fait référence dans une fonction à une variable définie dans un autre module (en compilation séparée), on précisera qu'elle est externe par le mot-clé `extern`.
NB : Nous déconseillons fortement l'utilisation de variables externes.

statiques : une variable globale statique (mot-clé `static`) est une variable dont le nom n'est pas exporté à l'édition de liens, et qui reste donc invisible hors du module où elle est définie.
Une variable interne à une fonction qui est déclarée statique est une variable *rémanente* : sa portée de visibilité est réduite à la fonction, mais elle n'est initialisée que la première fois où la fonction qui la déclare est appelée ; ensuite, sa valeur persiste d'un appel de la fonction à l'autre.
Le mot-clé `static` permet également de définir les variables et méthodes de classe (cf. § 7.6).

registres : on peut demander qu'une variable de type entier, caractère ou pointeur soit implantée dans un registre, ce qui est souvent utile quand on veut aller vite. Les indices dans les tableaux et les pointeurs en mémoire sont souvent de bons candidats pour être déclarés comme registres.
Attention : seule une variable automatique peut être de type registre. De plus, le mot-clé `register`, à employer dans ce cas, ne donne qu'une indication au compilateur ; on ne garantit pas que la variable sera bien en registre, le compilateur n'ayant à sa disposition qu'un nombre limité de registres. À vous de donner les indications les plus intelligentes...

Les déclarations de variables peuvent en plus être agrémentées de l'un des deux mots clés suivants :

`const` : la variable désigne en fait une constante ; aucune modification n'est autorisée dans le programme.

`volatile` : un objet déclaré *volatile* peut être modifié par un événement extérieur à ce qui est contrôlé par le compilateur (exemple : variable mise à jour par l'horloge système). Cette indication donnée au compilateur lui signale que toute optimisation sur l'emploi de cette variable serait hasardeuse.

6 Pointeurs

Les *pointeurs* sont des variables contenant des adresses. Ils permettent donc de faire de l'adressage indirect. Ainsi :

```
int* px;
```

déclare une variable `px` qui est un pointeur sur un entier. La variable pointée par `px` est notée `*px`. Inversement, pour une variable

```
int x;
```

on peut accéder à l'adresse de `x` par la notation `&x`. Ainsi, je peux écrire :

```
px = &x;
```

ou

```
x = *px;
```

Voici une autre manière d'écrire la fonction `swap()` qui échange deux entiers, cette fois-ci en passant par des pointeurs :

```
swap(int* px, int* py)
{
   int temp;    // variable temporaire

   temp = *px;
   *px = *py;
   *py = temp;
}
```

et pour échanger deux paramètres on appellera :

```
int a,b;

swap(&a,&b);
```

Attention : un des pièges les plus classiques en C++ est celui du pointeur non initialisé. Le fait d'avoir déclaré une variable de type pointeur ne suffit pas pour pouvoir déréférencer ce pointeur. Encore faut-il qu'il pointe sur une « case » mémoire valide. Pour reprendre l'exemple précédent, si j'écris

```
int* px;
*px = 3;
```

j'ai de très fortes chances d'avoir une erreur à l'exécution, puisque `px` ne désigne pas une adresse mémoire dans laquelle j'ai le droit d'écrire. Ce n'est qu'après avoir écrit par exemple `px = &x;` comme dans l'exemple ci-dessus que l'instruction `*px = 3;` devient valide.

6.1 Les tableaux

On déclare un tableau de la manière suivante :

```
int a[10];
```

Il y a une très forte relation entre un pointeur et un tableau. Dans l'exemple précédent, `a` est en fait une constante de type adresse ; en effet, `a` est l'adresse du début du tableau. Par conséquent, on peut écrire les choses suivantes :

```
int* pa, a[10];

pa = &a[0];
```

ou

```
pa = a;
```

Mais attention, il y a des différences dues au fait que `a` est une adresse constante alors que `pa` est une variable. Ainsi, on peut écrire

```
pa = a;
```

mais il n'est pas valide d'écrire

```
a = pa;
```

Quand on veut passer un tableau en paramètre formel d'une fonction, il est équivalent d'écrire :

```
void funct(int tab[])
```

ou

```
void funct(int* tab)
```

car on passe dans les deux cas une adresse.

Remarque : comme en Java, les indices, qui correspondent à des déplacements, commencent toujours à 0.

Voyons maintenant comment on peut utiliser cette équivalence entre pointeurs et tableaux pour parcourir un tableau sans recalculer systématiquement l'adresse du point courant. Le problème est de calculer la moyenne d'une matrice 200×200 d'entiers.

```
int tab[200][200];
long int moyenne=0;
register int* p = tab;

for (register int i=0  ; i < 200 ; i++)
     for (register int j=0 ; j < 200 ; j++ , p++)
         moyenne += *p;
moyenne /= 40000;
```

Remarque : on peut écrire cela de manière encore plus efficace en profitant du fait qu'on utilise `p` pour l'incrémenter en même temps. Par ailleurs, une seule boucle suffit, et il est inutile d'utiliser des compteurs :

```
int tab[200][200];
long int moyenne=0;
register int* p = tab;
register int* stop = p + 200 * 200;
for ( ; p < stop ; ) /*on ne fait plus p++ ici*/
         moyenne += *p++; /*on accède à la valeur pointée
                           par p, puis on l'incrémente*/
moyenne /= 40000;
```

Mais attention : le programme devient ainsi à peu près illisible, et je déconseille d'abuser de telles pratiques, qui ne sont justifiées que dans des cas extrêmes, où l'optimisation du code est un impératif.

Notez aussi qu'il est exclu de réaliser des « affectations globales » sur les tableaux, autrement que par le mécanisme des pointeurs (pas de recopie globale).

6.2 Allocation dynamique de mémoire

L'allocation et la libération dynamique de mémoire sont réalisées par les opérateurs `new` et `delete`. Une expression comprenant l'opération `new` retourne un pointeur sur l'objet alloué. On écrira donc par exemple :

```
int* pi = new int;
```

Pour allouer un tableau dynamique, on indique la taille souhaitée comme suit :

```
int* tab = new int[20];
```

Contrairement à Java, C++ n'a pas de mécanisme de ramasse-miettes ; c'est donc à vous de libérer la mémoire dynamique dont vous n'avez plus besoin (voir aussi la notion de destructeur pour les classes — § 7.1) :

```
delete pi;
delete [] tab;
```

6.3 Arithmétique sur les pointeurs

Comme le montre l'exemple du § 6.1, un certain nombre d'opérations arithmétiques sont possibles sur les pointeurs, en particulier l'incrémentation.

Tout d'abord, on peut leur ajouter ou leur soustraire un entier n. Cela revient à ajouter à l'adresse courante n fois la taille d'un objet du type pointé. Ainsi, dans un tableau, comme nous l'avons vu, l'instruction `p++` (qui est la même chose que `p = p+1`) fait pointer `p` sur la *case suivante* dans le tableau, c'est-à-dire que l'adresse est incrémentée de la taille (en octets) du type pointé.

On peut comparer deux pointeurs avec les opérateurs relationnels. Evidemment, cela n'a de sens que s'ils pointent dans une même zone (tableau par exemple).

Enfin, on peut soustraire deux pointeurs. Le résultat est un entier indiquant le nombre de « cases » de la taille du type pointé entre les deux pointeurs. Là encore, cela n'a de signification que si les deux pointeurs pointent dans la même zone contiguë.

6.4 Compléments sur les pointeurs

On pourrait encore dire beaucoup sur les pointeurs. Nous nous contentons ici de signaler quelques points que le lecteur intéressé par la poétique de C++ pourra approfondir dans la littérature appropriée :

– C++ propose deux manières de représenter les *chaînes de caractères* : celle héritée de C et le type `string` de la bibliothèque standard C++. Nous vous conseillons bien entendu d'utiliser ce dernier.

Mais comme vous risquez d'être parfois confrontés à des chaînes de caractères « à l'ancienne » (c'est-à-dire à la mode C), sachez que ce sont des tableaux de caractères terminés par le caractère nul (de code 0, et noté comme l'entier 0 ou le caractère `\0`).

– On peut bien sûr utiliser des *tableaux de pointeurs*, des *pointeurs de pointeurs*, des pointeurs de pointeurs de pointeurs, etc. Bref, vous voyez ce que je veux dire...
– On peut même manipuler des *tableaux de fonctions*, des *pointeurs de fonctions*, ce qui permet d'appeler plusieurs fonctions différentes en se servant du même pointeur.

7 Classes et instances

De manière classique, la classe regroupe des variables d'instance et des méthodes, ainsi que d'éventuelles variables et méthodes de classe. Contrairement à Java, on distingue en C++ la définition de la classe de sa mise en œuvre. La première regroupe la déclaration des variables et les signatures des méthodes ; elle se met dans un fichier *header*, qui est inclus quand on veut faire référence à l'interface de cette classe dans une autre classe ou dans un programme. Dans ce fichier *header*, on ne met *a priori* pas les corps des méthodes, sauf celles qui sont *inline*.

Illustrons cela en déclarant une classe d'objets postaux, ayant quatre variables d'instance : `poids`, `valeur`, `recommande` et `tarif` :

```
class ObjetPostal {
protected:
    int poids;
    int valeur;
    bool recommande;
public:
    // Variable d'instance publique -- je sais, ce n'est pas bien !
    int tarif;
    // Constructeur
    ObjetPostal(int p = 20);
    // Méthodes inline
    bool aValeurDeclaree() { return (valeur > 0); }
    int poidsObjet() { return poids; }
    void recommander() { recommande = true; }
};
```

Comme en Java, les variables d'instance et les méthodes peuvent être *privées*, *protégées* ou *publiques*. La différence entre données protégées et données privées est que seules les premières restent accessibles dans les sous-classes de la classe. Les trois variables `poids`, `valeur` et `recommande` sont protégées : elles ne sont accessibles que par les méthodes définies dans la classe `ObjetPostal` et dans celles de ses sous-classes éventuelles. La variable `tarif` est publique : elle est accessible par n'importe quelle instance de n'importe quelle classe[6].

Les méthodes dont la définition est incluse dans la déclaration de la classe, comme `aValeurDeclaree`, `recommander` et `poidsObjet`, sont implantées par des fonctions `inline` pour un gain d'efficacité à l'exécution. Le fait qu'elles soient définies à l'intérieur de la déclaration de classe suffit à les rendre *inline*, sans nécessité de mot clé particulier.

La fonction `ObjetPostal(int)`, de même nom que la classe, est un *constructeur* de la classe. Elle est simplement déclarée ici, et sera définie ailleurs. Nous y reviendrons au § 7.1.

La classe `ObjetPostal` peut être utilisée comme un nouveau type dans le programme :

```
ObjetPostal* z = new ObjetPostal(200);
...
delete z;
```

6. Le fait que j'ai choisi de rendre cette variable d'instance publique pour les besoins de la démonstration ne signifie pas que cette pratique est à recommander, loin de là. Une règle générale, qui souffre très peu d'exceptions, est de toujours cacher les détails d'implantation, donc de rendre les variables d'instance privées ou protégées.

Attention : la variable `z` est ici un pointeur sur l'instance, et non l'instance elle-même.

Dans le corps d'une méthode, les variables d'instance de la classe sont désignées simplement par leur nom. L'accès aux variables et méthodes d'autres objets se fait classiquement par la notation pointée, ou par la notation « flèche » dans le cas d'un pointeur :

```
ObjetPostal op;
...
op.recommander();
...
ObjetPostal* z = new ObjetPostal(200);
...
if (z->aValeurDeclaree()) {
   ...
}
```

7.1 Constructeurs et destructeurs

Toute classe peut comporter une ou plusieurs fonctions publiques particulières portant le même nom que la classe et appelées les *constructeurs*. Elles précisent comment doit être créée — ou plutôt initialisée — une instance de la classe, en donnant en particulier les valeurs initiales de certaines variables d'instance.

Revenons sur le constructeur `ObjetPostal` déclaré précédemment dans la classe de même nom. Dans le cas présent, seule cette fonction est définie hors du fichier *header*, dans le fichier de définition qui porte le nom de la classe et typiquement le suffixe `.C` ou `.cpp` :

```
#include <ObjetPostal.h>   // inclusion de la déclaration

ObjetPostal::ObjetPostal(int p)
  { poids = p; valeur = 0; recommande = false; }
```

À noter que dans la déclaration de la classe, le paramètre `p` a la valeur par défaut 20 ; l'appel du constructeur sans paramètre est donc équivalent à son appel avec la valeur 20. À noter aussi l'utilisation de l'*opérateur de résolution de portée* `::`, nécessaire dès qu'on n'est plus « dans » la définition de la classe, pour rattacher la fonction à sa classe d'appartenance.

En fait, il n'est jamais nécessaire d'appeler explicitement un constructeur pour créer une instance. C'est le compilateur qui se charge de choisir le constructeur à utiliser, en fonction des paramètres d'instanciation. Si aucun constructeur ne s'applique, un constructeur par défaut est appelé, qui initialise les variables à des valeurs nulles. Il est cependant fortement recommandé de toujours prévoir un constructeur, en tout cas dès que la classe n'est pas triviale.

Conformément à ce qui vient d'être dit, la déclaration :

```
ObjetPostal x;
```

dans une méthode ou un programme, crée un objet postal dont le poids est de 20 (valeur par défaut). En revanche, la déclaration

```
ObjetPostal x(140);
```

crée une instance de la classe `ObjetPostal` de poids 140 grammes.

En fait, un constructeur comme ce dernier, avec un seul paramètre, tient lieu de fonction de conversion implicite de type. Par exemple, la déclaration suivante est valide :

```
ObjetPostal x = 30;
```

Elle est traduite par l'application de la fonction de conversion d'un entier en objet postal, équivalente à la déclaration suivante :

```
ObjetPostal x(30);
```

Ce mécanisme de conversion implicite reste néanmoins limité aux constructeurs ayant un seul argument, ou pour lesquels les autres arguments ont tous des valeurs par défaut.

La place mémoire occupée par une instance locale est automatiquement restituée quand la variable qui la désigne cesse d'exister, c'est-à-dire à la sortie du bloc de programme dans lequel la variable est définie.

Cependant, il arrive qu'un constructeur effectue une allocation dynamique de mémoire, typiquement pour une des variables d'instance. Pour restituer la place ainsi allouée quand l'objet doit disparaître, il faut définir un *destructeur*, déclaré comme une fonction portant le nom de la classe précédé du caractère ~. Ce destructeur est appelé automatiquement quand l'objet cesse d'exister.

Supposons par exemple qu'un sac postal est caractérisé par une capacité maximale, un nombre d'objets contenus et un tableau d'objets postaux dont la taille est fixée dynamiquement. La place nécessaire pour ce tableau étant allouée par le constructeur, elle doit être restituée par un destructeur :

```
// Dans le fichier SacPostal.h
class SacPostal {
private:
    int nbelts;        // nombre d'objets dans le sac
    int capacite;      // capacité du sac
    ObjetPostal* sac;  // le tableau représentant le sac
public:
    SacPostal(int);    // le constructeur
    ~SacPostal();      // le destructeur
    // et les autres methodes...
};

// Dans le fichier SacPostal.cpp
SacPostal::SacPostal(int cap)
{
    capacite = cap; nbelts = 0; // sac vide
    sac = new ObjetPostal[cap]; // allocation du tableau
}

SacPostal::~SacPostal()
{
    delete [] sac;      // restitution de la place
                        // occupée par le tableau sac
}
```

La déclaration d'une variable `courrierDeLyon` de type `SacPostal` peut se faire comme suit :

```
SacPostal courrierDeLyon(250);
```

Le constructeur `SacPostal::SacPostal(int)` est automatiquement appelé et un tableau de 250 objets postaux est alloué dynamiquement. Le compilateur engendre aussi un appel automatique au destructeur `SacPostal::~SacPostal()` quand la variable `courrierDeLyon` cesse d'exister, c'est-à-dire pour l'exemple donné à la sortie du bloc dans lequel elle est définie.

Les constructeurs et destructeurs peuvent aussi être appelés explicitement, lorsqu'on fait de l'allocation dynamique de mémoire, comme dans l'exemple suivant :

```
SacPostal* ps = new SacPostal(55);  // constructeur appelé
...
delete ps;                          // destructeur appelé
```

7.2 Les amis

Avec la notion d'*amis*, C++ donne le moyen d'affiner plus finement le contrôle des droits d'accès que par les simples notions de variables publiques ou privées. Par exemple, si la classe `SacPostal` est déclarée amie de la classe `ObjetPostal`, toutes ses instances sont autorisées à accéder aux variables privées d'`ObjetPostal`:

```
class ObjetPostal {
friend class SacPostal;
    ...
```

```
};
```

Cette « amitié » peut être plus sélective et se limiter à une ou plusieurs fonctions précises. Supposons qu'en fait seule la méthode `affranchir` de la classe `SacPostal` ait besoin d'accéder aux champs privés de `ObjetPostal`. Seule cette méthode est alors déclarée amie, à la place de la classe :

```
class ObjetPostal {
friend void SacPostal::affranchir();
    ...
};
...
// Et dans la définition de la classe SacPostal
SacPostal::affranchir() {
   ObjetPostal* x;
   ...
   if (x->poids < 20)  // L'accès à poids est autorisé car la
     x->tarif = 2; ... // méthode est amie de la classe ObjetPostal
}
```

Associées aux notions de données publiques, protégées et privées, les classes et les fonctions amies permettent de contrôler de manière fine les protections et les accès aux variables d'instance.

7.3 L'héritage

C++ permet de réaliser de l'héritage multiple entre classes ; nous nous limiterons cependant dans ce polycopié à l'exposé de l'héritage simple. Une sous-classe, appelée *classe dérivée*, hérite classiquement des attributs de sa superclasse :

```
class Colis : public ObjetPostal {
protected:
    int volume;
};

class Lettre : public ObjetPostal {
protected:
    bool urgent;
};

class CourrierInterne : public Lettre {
public:
    CourrierInterne(int p) : (p)
        {
           tarif = 0;
        }
};
```

Lors de la création d'une instance d'une classe donnée, tous les constructeurs de la hiérarchie d'héritage de la classe sont automatiquement activés, du plus général au plus particulier. Ainsi, la définition du constructeur de `CourrierInterne` indique les valeurs à donner aux paramètres du constructeur de la superclasse, à savoir celui d'`ObjetPostal`. Grâce à l'expression `: (p)` qui suit la définition du constructeur `CourrierInterne(int)`, le constructeur `ObjetPostal(int)` est donc appelé avec la valeur de `p` avant la mise à zéro du champ `tarif`. Ce mécanisme est similaire à l'emploi de `super` en Java.

Dans une classe, les attributs hérités deviennent privés par défaut, même s'ils étaient publics dans la superclasse. Toutes les autres classes, y compris ses sous-classes, ne peuvent y accéder directement. Cependant les attributs publics hérités restent publics si l'héritage est dit public grâce au mot clé `public`, comme dans les exemples précédents. En pratique, l'héritage est public dans la grande majorité des cas, et ce n'est que lorsqu'on souhaite hériter de l'implantation tout en masquant l'interface de la classe qu'on fait de l'héritage « privé ». Pensez donc à mettre le mot clé `public` !

7.4 Liaison dynamique

En C++, la liaison dynamique n'est pas systématique, contrairement à Java. Pour assurer cette liaison dynamique quand elle est souhaitée, on utilise le mécanisme des fonctions virtuelles. Ainsi, pour que la méthode `affranchir` de la classe `ObjetPostal` puisse être redéfinie dans les sous-classes et invoquée uniformément et dynamiquement sur une collection d'objets postaux divers, instances de ces différentes sous-classes, elle doit être déclarée comme virtuelle (mot-clé `virtual`) dans la classe `ObjetPostal`:

```
class ObjetPostal {
friend void SacPostal::affranchir();
protected:
    int poids;
    int valeur;
    bool recommande;
public:
    int tarif;
    // Constructeur
    ObjetPostal(int p = 20);
    // Destructeur virtuel -- voir ci-après
    virtual ~ObjetPostal() {} // rien de spécial à faire ici
    // Méthodes inline
    bool aValeurDeclaree() { return (valeur > 0); }
    int poidsObjet() { return poids; }
    void recommander() { recommande = true; }
    // Méthode affranchir, implantation par défaut
    // Sera redéfinie dans les sous-classes
    virtual void affranchir() { tarif = 0; }
    ...
};
```

Alternativement, on peut décider de ne pas donner d'implantation par défaut à la méthode `affranchir`, en écrivant :

```
class ObjetPostal {
   ...
   virtual void affranchir() = 0;
```

Cela fait de la classe `ObjetPostal` une *classe abstraite*, qui ne peut être instanciée. Pour être instanciables, ses sous-classes doivent obligatoirement définir une méthode `affranchir`.

Il est utile de savoir qu'un destructeur peut aussi être déclaré virtuel. Supposons que les classes `Colis`, `Lettre` et `CourrierInterne` soient munies de constructeurs et de destructeurs spécifiques. Si une instance de `SacPostal` peut contenir des objets postaux de toutes sortes, le destructeur de la classe `SacPostal` doit appeler un destructeur spécifique pour chaque objet contenu dans le sac. La façon de faire la plus élégante consiste à déclarer virtuel le destructeur de la classe `ObjetPostal` dans la définition de la classe, comme dans l'exemple ci-dessus. La fonction `SacPostal::~SacPostal()` s'écrit alors :

```
SacPostal::~SacPostal()
{
    delete [] sac;
}
```

ce qui a pour effet d'appeler successivement le destructeur de chaque élément du sac. Le destructeur de la classe `ObjetPostal` étant virtuel, c'est bien le destructeur spécifique à chaque objet du sac qui est appelé par l'instruction `delete`.

7.5 L'accès à la superméthode

L'accès à une méthode masquée peut se faire en C++ par un appel direct à cette méthode, grâce à l'opérateur de résolution de portée. Si par exemple l'affranchissement d'un courrier par avion est le même que celui

d'une lettre, augmenté de quelques opérations spécifiques, on peut écrire :

```
class Lettre : public ObjetPostal {
protected:
    bool urgent;
public:
    void affranchir() { tarif = 2 + (urgent ? 1 : 0); }
};

class ParAvion : public Lettre {
public:
    void affranchir();
};

void ParAvion::affranchir()
{
    // affranchissement ordinaire
    Lettre::affranchir();
    // 7F de supplement pour courrier aerien
    tarif += 7;
}
```

7.6 Variables de classe

Les variables de classe sont déclarées avec le mot-clé `static`. Par exemple, la classe `Lettre` peut être munie de la variable de classe `tarifLettre`, indiquant le tarif d'affranchissement par défaut :

```
class Lettre : public ObjetPostal {
protected:
    bool urgent;
public:
    static int tarifLettre;
};
...
// Dans la définition de la classe Lettre (Lettre.cpp par exemple)
// Déclatation et initialisation de la variable de classe
int Lettre::tarifLettre = 3;
```

8 La surcharge d'opérateurs

C++ autorise la surcharge des opérateurs. Par exemple, définissons un opérateur `+` permettant d'ajouter le contenu de deux sacs postaux dans un nouveau sac. Comme cet opérateur doit accéder aux champs privés de ses opérandes, il est déclaré ami de la classe `SacPostal`[7] :

```
class SacPostal {
private:
    int nbelts;
    int capacite;
    ObjetPostal* sac;
public:
    SacPostal(int);
    ~SacPostal();
    friend SacPostal operator+(SacPostal&, SacPostal&);
};
```

7. On aurait aussi pu définir l'opérateur `+` dans la classe `SacPostal`.

```
...

SacPostal operator+(SacPostal& sac1, SacPostal& sac2) {
    // création d'un gros sac de capacité ad hoc
    SacPostal grosSac(sac1.capacite + sac2.capacite);
    // nombre d'éléments de ce gros sac
    grosSac.nbelts = sac1.nbelts + sac2.nbelts;
    // mettre les éléments de sac1 dans grosSac
    int i = 0;
    for ( ; i < sac1.nbelts ; i++)
        grosSac.sac[i] = sac1.sac[i];
    // puis mettre les éléments de sac2 dans grosSac
    for (register int j = 0 ; j < sac2.nbelts ; i++, j++)
        grosSac.sac[i] = sac2.sac[j];
    return grosSac;
}
```

Le nouvel opérateur s'emploie ensuite de manière transparente sur les instances de la classe `SacPostal`:

```
SacPostal sacSeichamps = 200,
          sacVandoeuvre = 1500;
...
SacPostal sacNancy = sacSeichamps + sacVandoeuvre;
...
```

9 La bibliothèque d'entrées/sorties

Nous ne prétendons pas couvrir dans ce polycopié les très nombreuses fonctionnalités couvertes par la bibliothèque standard C++. Cependant, il nous semble utile de donner quelques indications sur les entrées/sorties.

Pour utiliser la bibliothèque, il faut inclure son fichier de déclarations :

```
#include <iostream>
```

Les opérations standards d'entrée et de sortie sont fournies par trois flots (*streams*), désignés par les variables suivantes :

- `cin` désigne le flot d'entrée standard (typiquement, votre clavier),
- `cout` désigne le flot de sortie standard (typiquement, la fenêtre d'exécution sur votre écran),
- `cerr` désigne le flot standard des messages d'erreur.

Les opérateurs `<<` et `>>` sont redéfinis pour permettre des écritures et lectures aisées :

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

cout << "Bonjour, comment vous appelez-vous ? ";
string nom;
cin >> nom;
if (nom.string_empty()) {
   cerr << "erreur : nom vide" << endl;
}
else {
   cout << nom << ", donnez-moi maintenant votre âge : ";
   int age;
   cin >> age;
   cout << "Ouah, vous n'êtes plus tout jeune !" << endl;
}
```

On notera au passage l'emploi de `string`, la bibliothèque de manipulation de chaînes de caractères C++, et de la constante `endl`, qui indique le passage à la ligne.

Bien entendu, la bibliothèque `iostream` fournit de nombreuses autres fonctionnalités d'entrée/sortie, et la bibliothèque `fstream` fournit les fonctionnalités de manipulation de fichiers. Nous nous sommes contentés ici de donner quelques rudiments vous permettant d'écrire vos tout premiers programmes...

10 Pour en savoir plus

Ce polycopié ne fournit qu'une introduction très succincte au langage C++. Nous avons volontairement passé sous silence un grand nombre de caractéristiques, dont notamment l'héritage multiple, la gestion des exceptions, les *namespaces*, les nouveaux mécanismes de conversion, l'identification dynamique de type, et la riche bibliothèque standard de C++, avec notamment les chaînes de caractères. Les notions de *templates* et d'algorithmes génériques font l'objet d'un cours séparé.

Pour approfondir vos connaissances, nous vous conseillons la lecture d'un des bons ouvrages consacrés à C++, et nous recommandons tout particulièrement les deux livres suivants :

- Bjarne Stroustrup. *The C++ programming Language, 3rd Edition*. Addison-Wesley, 1997. *La* référence de base, par l'auteur du langage.
- Stanley B. Lippman, Josée Lajoie. *C++ Primer, 3rd Edition*. Addison-Wesley, 1998. Très complet (1200 pages), un des meilleurs livres pour apprendre C++, à mon avis. Une traduction française existe, mais pour l'instant, à ma connaissance, elle n'est disponible que pour la 2^{e} édition, moins complète et pas à jour par rapport à la norme.