Une courte introduction à C++

Karl Tombre

ECOLE DES MINES DE NANCY

Département informatique
Options ISI et CSSEA
2e année, 2e semestre

Version 2.3
Avril 2006

Ce cours, qui fait partie du module *Programmation de systèmes ambiants* commun aux options *Ingénierie des systèmes informatiques* et *Conception sûre de systèmes embarqués et ambiants*, a pour objectif de donner une introduction à C++ et à la programmation générique. Nous supposerons que vous avez déjà une connaissance minimale de Java et de la programmation objet ; si ce n'est pas le cas, je vous conseille l'étude préalable de mon polycopié de première année d'introduction à la programmation [10].

Ce polycopié ne demande qu'à s'enrichir — je suis donc preneur de toute remarque ou critique constructive...

 Document édité avec XEmacs et formatté avec LaTeX. Achevé d'imprimer le 9 avril 2006. Un grand merci à Philippe Dosch, qui m'a autorisé à reproduire ici plusieurs parties de son polycopié [1], notamment tout ce qui concerne la généricité. Merci aussi à Luigi Liquori, qui m'avait aidé à rédiger une première version de la partie sur la compilation (§ 5.1). Merci enfin à tous ceux, élèves, collègues ou personnes extérieures, qui m'ont fait parvenir leurs commentaires, remarques et critiques pour faire évoluer ce cours, qui reste éminemment imparfait...

Table des matières

Chapitre 1

Introduction

1.1 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 l'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 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 que plus tard 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.

[1] *International Federation for Information Processing.*

D'autres langages, et en particulier Java et C#, 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.

Pour le lecteur qui voudrait en savoir plus que les informations succinctes données par ce polycopié, il existe de très nombreux ouvrages consacrés à C++. Il nous est difficile de faire un tri sérieux, la qualité de ces ouvrages étant très variable. Je conseille néanmoins les références à mon avis incontournables [4, 9], ainsi qu'un excellent ouvrage disponible gratuitement sur le Web [2]. Enfin, pour se perfectionner dans l'emploi de C++ comme langage de programmation, il existe de nombreux ouvrages, dont les incontournables contributions de Scott Meyers [5, 6, 7].

1.2 Implantation de modules en C++

Un module C++ (et en particulier une classe) est généralement implanté par deux fichiers : un fichier d'interface, qui spécifie les services offerts par le module (définition de types, interfaces dec classes, déclaration de fonctions et de procédures) et un fichier d'implantation dans lequel sont mises en œuvre les fonctionnalités déclarées dans l'interface. Les extensions conventionnellement utilisées en C++ sont :

- `.h, .hh, .H` pour les fichiers d'interface C++ (les *headers*).
  Remarque : la norme du langage C++ prévoit que les fichiers d'interface perdent leur extension. C'est la raison pour laquelle les interfaces spécifiques et standard de C++ sont désormais désignées sans leur extension (comme `iostream` par exemple).
- `.cc, .cpp, .C` pour les fichiers d'implantation C++.

Il y a ici une première différence avec Java, où une classe est complètement définie par un seul et même fichier de suffixe `.java`. En C++, une classe que nous appellerons `Toto` est habituellement décrite dans deux fichiers séparés :

- `Toto.H` ou `Toto.hh` ou `Toto`. Ce fichier *header* décrit l'*interface de la classe*, et contient :
  1. des directives pour le préprocesseur (cf. § 5.2),
  2. l'inclusion d'autres fichiers *headers* pour des classes employées dans le fichier courant,
  3. une définition formelle de la classe, avec ses constructeurs, ses méthodes et ses variables.
- `Toto.cpp`. Ce fichier définit la *mise en œuvre* de la classe, et contient :
  1. l'inclusion des *headers* nécessaires (l'inclusion de `Toto.hh` – ou autre nom donné au *header* de la classe – est obligatoire),
  2. la mise en œuvre des constructeurs et des méthodes,
  3. l'implantation et l'initialisation des variables de classe.

Il faut bien noter que pour utiliser un type – une classe – dans un module, il faut obligatoirement inclure son *header*, sinon la compilation ne réussira pas, les interfaces des classes utilisées n'étant pas trouvées (elles ne sont pas recherchées et chargées automatiquement, comme en Java).

Nous reviendrons au § 5.1 sur les spécificités de la compilation en C++.

Chapitre 2

Les structures de base

2.1 Types de base et constantes

En C++, les *types de base* sont :
- `bool` : booléen[1], 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),
- `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`[2]) ou octale (précédées de `0`[3]). 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 guillemets :

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

[1] 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, un certain nombre de compilateurs C++ continuent à accepter des valeurs entières à la place de valeurs booléennes.

[2] zéro-X.

[3] zéro.

2.2 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 :

2.2.1 Opérateurs arithmétiques

`+` addition

`-` soustraction

`*` multiplication

`/` division (entière ou réelle)

`%` modulo (sur les entiers)

2.2.2 Opérateurs relationnels

`> >= <= <` comparaisons

`== !=` égalité et inégalité

`!` négation (opérateur unaire)

`&&` ET relationnel

`||` OU relationnel

2.2.3 L'affectation

`=` affectation

Il faut bien noter que, comme en Java, 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.

2.2.4 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 l'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;
```

2.2.5 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` force 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).

2.2.6 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 `^=`.

2.2.7 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 ; les « vrais » programmeurs C++ sont même convaincus qu'elle est plus lisible !

2.2.8 Conversions de types

On désire souvent changer le type du résultat retourné par une expression. Pour cela existe le mécanisme de *cast*. Celui-ci a été profondément modifié par la norme définitive de C++ ; comme beaucoup de *casts* « ancien régime » existent encore, nous expliquons ici les deux, en insistant sur la norme officielle. Nous vous recommandons d'ailleurs d'utiliser les constructions de cette dernière. Certaines explications font référence à des notions qui ne sont expliquées que dans les chapitres suivants...

L'ancien système

`(`*<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.

La norme officielle

Les opérateurs de conversion de types proposés par la norme sont particulièrement utiles dans des contextes tels que le polymorphisme, afin de convertir un objet d'une classe de base vers une classe dérivée. En effet, jusqu'à l'introduction de ces opérateurs, ce type de conversion délicate était entièrement à la charge du programmeur, qui devait vérifier la validité de la conversion avant de la réaliser. Du coup, cela pouvait engendrer quelques problèmes de sécurités, que ces nouveaux opérateurs sont censés résoudre.

Certains nouveaux *casts* se basent sur une fonctionnalité ajoutée il y a quelques années au langage C++ : la *Run-Time Type Identification (RTTI)*[4]. L'objectif de ces nouveaux opérateurs est de disposer d'une syntaxe améliorée, plus claire, d'une sémantique moins ambiguë et de réaliser des conversions de types en toute sécurité. Les opérateurs de *cast* sont au nombre de 4 :

1. L'opérateur `static_cast<T> (expr)` : cet opérateur est utilisé pour effectuer des conversions qui sont résolues à la compilation. Il peut être utilisé pour convertir un pointeur (ou une référence) sur une classe de base vers un pointeur (ou une référence) sur une classe dérivée. L'opérateur n'effectue aucune vérification au cours de l'exécution (comme son nom l'indique) et doit donc être utilisé pour des conversions non-ambiguës. Mal utilisé, il renvoie un résultat indéfini. Il doit surtout être utilisé pour effectuer des conversions arithmétiques. Il est assez proche de la conversion de l'ancien système, mais permet de supprimer des trous de sécurité qui existaient.
2. L'opérateur `const_cast<T> (expr)` : cet opérateur permet de supprimer la constance d'un objet. Ce n'est pas très naturel, mais utile dans certaines situations ; il doit être ainsi utilisé avec parcimonie. Un exemple d'utilisation se trouve ci-dessous :

   ```
   void f(Article &i)
   {
   }

   void g(const Article &j)
   {
     f(j);  // Erreur : j est constant et f n'attend pas un const
     f(const_cast<Article&> (j));  // Ok
   }
   ```

   De même, il peut être utilisé à l'intérieur d'une méthode de classe constante : en l'appliquant sur le pointeur `this`, on peut modifier par la suite l'objet courant (!)
3. L'opérateur `dynamic_cast<T> (expr)` : c'est certainement l'un des nouveaux opérateurs les plus intéressants. Il peut uniquement être utilisé sur des pointeurs ou des références pour naviguer dans une hiérarchie de classes. Il peut être utilisé pour convertir un objet d'une classe dérivée vers un objet d'une classe de base ou inversement. Dans le premier cas, c'est une classique conversion statique qui est effectuée tandis que dans le second cas c'est une

[4] Identification des types au cours de l'exécution.

conversion dynamique qui est réalisée, en se basant sur le système RTTI. Dans ce cas, si la conversion est possible, l'opérateur de conversion renvoie un pointeur valide, ou un pointeur nul sinon. Cette fonctionnalité est très puissante comme le montre l'exemple suivant :

```
#define MAXELTS 1000

int main()
{
  Article* lesArticle[MAXELTS];

  // Initialisation du tableau avec des articles hétérogènes
  // Deux cas différenciés : Alcools ou autres articles

  for (int i = 0; i < MAXELTS; i++) {
      BoissonAlcoolisee* ba;

      ba = dynamic_cast<BoissonAlcoolisee*> (lesArticle[i]);

      // Si l'article est un alcool, affichage du nom et du
      // degré d'alcool. Affiche du nom uniquement sinon.

      if (ba != NULL)
        cout << ba->nom() << " (" << ba->degre() << ")" << endl;
      else
        cout << lesArticle[i]->nom() << endl;
  }
}
```

4. L'opérateur `reinterpret_cast<T> (expr)` : cet opérateur peut être utilisé pour convertir des objets dont les types ne sont pas en relation. Le résultat de la conversion est dépendante de l'implantation, et n'est ainsi pas portable. Il peut être utilisé dans certains contextes particuliers de conversion entre types de pointeurs de fonctions.

Conclusion : parmi les nouveaux opérateurs introduits, `static_cast` et le `dynamic_cast` sont à utiliser en priorité.

2.2.9 Récapitulatif

Pour finir ce long 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. § 2.3.3).

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	`,`

2.3 Structuration 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 C une structure modulaire, et on peut très bien concevoir un programme C++ composé d'un grand nombre de *modules* 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.

C++ doit donc être considéré plutôt comme un langage « multi-paradigmes », permettant d'allier programmation object, programmation procédurale et programmation générique, et non comme un langage à objets « pur et dur ».

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.

2.3.1 Instructions et blocs

Chaque instruction est terminée par un point-virgule. 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++) ....
   ....
   }
```

Il est conseillé de déclarer les variables locales le plus tard possible, seulement au moment où on en a effectivement besoin.

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. Ceci étant dit, nous vous conseillons fortement de vous en tenir aux commentaires compris entre `//` et la fin de la ligne.

2.3.2 Structures conditionnelles

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

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

où l'exécution de l'une ou de l'autres des branches *alors* ou *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 très prudent et le documenter explicitement, le cas échéant !

2.3.3 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 exécutée 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 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...

2.4 Fonctions

Théoriquement, *toute fonction retourne une valeur*, qui peut être utilisée ou non. Toutefois, un mot clé particulier, `void`, permet d'indiquer qu'une fonction ne retourne pas de valeur (ce qui en fait *stricto sensu* une procédure et non une fonction !).

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 procédure 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);
```

Il est conseillé de passer le plus systématiquement possible les objets (par opposition aux variables de type simple) par référence, et non par valeur.

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 ou la procédure. Avec un passage par valeur, l'objet serait dupliqué dans la pile d'exécution. Cette fonctionnalité très intéressante apporte un degré de contrôle supérieur à ce que permet Java sur les opérations permises sur l'objet qui est passé à une fonction. Nous vous conseillons de l'utiliser le plus possible.

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 procédure print
}
```

Une fonction peut être déclarée *inline*, 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, en particulier parce que le compilateur aurait besoin de connaître l'adresse de la fonction.

Comme en Java, une fonction peut être *surchargée* ; la discrimination est alors faite sur le nombre et le type des paramètres effectifs.

Notons aussi qu'il est possible de définir des *valeurs par défaut* pour certains paramètres de fonctions. Certaines fonctions sont appelées avec des paramètres qui changent rarement. Considérons par exemple une fonction `ecranInit` qui est chargée d'initialiser un écran d'ordinateur (en mode caractères). Dans 90% des cas, l'écran a les dimensions 24 lignes $\times$ 80 caractères et doit être initialisé dans 99% des cas avec le caractère ' ', qui provoque l'effacement de l'écran. Plutôt que de contraindre le programmeur à énumérer des paramètres qui sont *généralement* invariants, C++ offre la possibilité de donner des valeurs par défaut à certains paramètres lors de la déclaration de la fonction, comme ci-dessous :

```
void ecranInit(Ecran ecran, int lig = 24, int col = 80, char fond = ' ');

void ecranInit(Ecran ecran, int lig, int col, char fond)
{
  ...
```

```
}

int main()
{
  Ecran ec;

  ecranInit(ec);              // Éq. à : ecranInit(ec, 24, 80, ' ');
  ecranInit(ec, 26);          // Éq. à : ecranInit(ec, 26, 80, ' ');
  ecranInit(ec, 26, 92);      // Éq. à : ecranInit(ec, 26, 92, ' ');
  ecranInit(ec, 26, 92, '+');
}
```

Quelques remarques sur cette fonctionnalité :

1. Une fonction peut définir des valeurs par défaut pour tous ses paramètres ou seulement pour une partie. Les paramètres acceptant des valeurs par défaut doivent se trouver *après* les paramètres sans valeur par défaut dans la liste des paramètres acceptés par une fonction.
2. Les valeurs par défaut de chaque paramètre ne peuvent être mentionnées qu'une seule fois. Ainsi, par convention, ces valeurs sont généralement mentionnées dans la *déclaration* de la fonction et pas dans sa *définition* (donc dans le *header* et pas dans le fichier de suffixe `.cpp` ou `.C`).
3. L'ordre de déclaration des paramètres est important : dans l'exemple ci-dessus il est en effet impossible de donner une valeur à `col` sans en donner une auparavant à `lig`. D'une façon générale, il faut donc positionner parmi les paramètres ayant des valeurs par défaut en premier ceux qui ont le plus de chances d'être modifiés.

2.5 Variables

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 globles : 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, 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. § 3.8).

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. Sauf cas très particuliers, comme en programmation système ou en micro-optimisation de code dans des boucles particulières, nous vous déconseillons de recourir à l'emploi de ce mot-clé.

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.

2.6 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 :

```
void 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.

2.6.1 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 à partir du début du tableau, 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;
int* p = tab;

for (int i=0  ; i < 200 ; i++)
     for (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;
int* p = tab;
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).

2.6.2 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 — § 3.2) :

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

L'exemple ci-dessous reprend et illustre l'utilisation de `new` et de `delete` pour des variables et des tableaux :

```
// Allocation d'une variable et d'un tableau en C++

int main()
{
  int* pi = new int;
  int* tab = new int[10];

  if ((pi != NULL) && (tab != NULL)) {
    ...
    delete pi;
    delete [] tab;
  }
}
```

2.6.3 Arithmétique sur les pointeurs

Comme le montre l'exemple du § 2.6.1, 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. Évidemment, 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.

2.6.4 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 (cf. § 4.5.3).
  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.

2.7 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;
   if (age > 35) {
       cout << "Ouah, vous n'êtes plus tout jeune !" << endl;
   }
   else {
       cout << "Blanc bec !" << endl;
   }
}
```

On notera au passage l'emploi de `string`, la bibliothèque de manipulation de chaînes de caractères C++ (cf. § 4.5.3) 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... Vous trouverez en annexe A quelques indications supplémentaires sur les entrées-sorties.

2.8 Les *namespaces*

Les *namespaces* (espaces de noms) ont été introduits dans la norme définitive de C++. Dans des projets conséquents, il n'est en effet pas rare d'utiliser plusieurs bibliothèques C++ qui peuvent parfois définir les mêmes identificateurs, ce qui génère des conflits. Avec les espaces de noms, il ne doit plus y avoir de conflits de noms : les déclarations restent cachées dans un *namespace* jusqu'à ce qu'on fasse explicitement appel à lui. On rejoint à plusieurs égards la notion de *package* en Java...

Pour définir un *namespace*, il faut utiliser le mot-clé `namespace` comme cela est présenté :

```
namespace MonNameSpace
{
  // Toutes les déclarations sont regroupées
  // dans ce bloc

  int f();

  // D'autres déclarations...

  // Fin du namespace
};
```

La déclaration d'un même *namespace* peut être réalisée dans plusieurs fichiers d'interface, le *namespace* complet résultant alors de l'union des déclarations. Dans l'exemple présenté, le nom complet de la fonction `f` devient `MonNameSpace::f`, selon une syntaxe qui est similaire à celle des méthodes membres de classe. Cependant, afin de ne pas être contraint de désigner la fonction `f` par rapport à son *namespace* lorsqu'il n'y a pas de risque de conflit, des facilités d'utilisation sont disponibles grâce à la définition d'alias ou à l'utilisation du mot-clé `using` :

```
// Si on souhaite définir un alias sur le nom d'un
// namespace

namespace mon = MonNameSpace;

mon::f();   // Appelle MonNameSpace::f()

// Si la fonction f est la seule à être présente,
// on peut déclarer

using MonNameSpace::f;

f();   // Appelle MonNameSpace::f()

// Si on souhaite bénéficier de toutes les déclarations
// d'un namespace sans avoir à les préfixer du nom
// du namespace

using namespace MonNameSpace;
```

L'utilisation de ces directives `using` est cependant à bannir des fichiers d'interface, pour éviter des conflits qui pourraient apparaître dans les modules incluant ces interfaces.

Du coup, la plupart des déclarations de la bibliothèque standard C++ ont été regroupées dans un *namespace*, appelé `std`. Ainsi, un programme utilisant des fonctionnalités de la bibliothèque standard devra comporter la directive

```
using namespace std;
```

pour éviter d'avoir à écrire explicitement `std::cout`, par exemple.

Chapitre 3

Programmation objet

3.1 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*, qu'on inclut quand on veut accéder 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;
    double tarif;
public:
    // Constructeur
    ObjetPostal(int p = 20);
    // Méthodes inline
    bool aValeurDeclaree() const { return (valeur > 0); }
    int getPoids() const { 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 quatre variables `poids`, `valeur`, `recommande` et `tarif` sont protégées : elles ne sont accessibles que par les méthodes définies dans la classe `ObjetPostal` et dans celles de ses sous-classes éventuelles.

Les méthodes dont la définition est incluse dans la déclaration de la classe, comme `aValeurDeclaree`, `recommander` et `getPoids`, 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 § 3.2.

À noter aussi l'utilisation de `const` dans la déclaration des méthodes `aValeurDeclaree` et `getPoids`. Cette qualification *garantit* que ces méthodes ne modifient pas l'état interne de l'objet. Elle est le complément naturel du passage par référence constante d'un objet à une fonction (cf. § 2.4) ; en effet, sur un objet passé en référence constante à une fonction, on ne pourra appliquer que des méthodes elles-mêmes déclarées constantes. Nous vous encourageons fortement à prendre

l'habitude dès le départ d'utiliser de manière systématique ces déclarations `const` partout où cela a un sens. Ce sera une garantie contre beaucoup d'effets de bord indésirables, dans la mesure où la correction du point de vue de la constance des objets est vérifiée dès la compilation.

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

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

Attention : la variable `z` est ici un pointeur sur l'instance, et non l'instance elle-même. Nous revenons au paragraphe suivant sur l'instanciation.

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()) {
   ...
}
```

3.2 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 que l'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. En particulier, nous verrons que pour être correctement utilisée par les containers et les algorithmes de la STL, une classe doit être munie d'un constructeur par défaut et d'un constructeur par copie (cf. § 4.3.1).

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. Nous conseillons de s'en tenir à la forme explicite `ObjetPostal x(30) ;`

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[1] :

```
// 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 méthodes...
};
```

```
// 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
}
// etc.
```

[1] Plus généralement, quand on écrit quelque part `new` (resp. `new T[]`), on doit s'assurer qu'au cours de l'exécution on passera à un moment par `delete` (resp. `delete []`), sous peine d'avoir des « fuites » de mémoire.

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
```

3.3 Les amis

Avec la notion d'*amis*, C++ donne d'affiner le contrôle des droits d'accès mieux 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 SacPostal;

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 SacPostal;

class ObjetPostal {
friend void SacPostal::affranchir();
    ...
};
...
// Et dans la définition de la classe SacPostal
void SacPostal::affranchir() {
   ObjetPostal* x;
   ...
   if (x->poids < 20)  // L'accès à poids est autorisé car la
     x->tarif = 0.53;  // 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.

3.4 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;
public:
    Colis(int p, int v) : ObjetPostal(p), volume(v) {}
};
```

```
class Lettre : public ObjetPostal {
protected:
    bool urgent;
public:
    Lettre(int p) : ObjetPostal(p), urgent(false) {}
};
```

```
class CourrierInterne : public Lettre {
public:
    CourrierInterne(int p) : Lettre(p) {
           tarif = 0.0;  // pas d'affranchissement pour le courrier interne
    }
};
```

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`, dans la grande majorité des cas !

À noter que le constructeur d'une classe appelle le(s) constructeur(s) de sa (ses) superclasse(s). Si on ne mentionne rien, c'est le constructeur par défaut de superclasse qui est appelé (le constructeur sans paramètres). Mais dans la plupart des cas (je conseille de le faire systématiquement) on indiquera explicitement comment « construire » la ou les partie(s) héritée(s). Ainsi, la définition du constructeur de `CourrierInterne` indique quel constructeur de la superclasse `Lettre` appeler, et avec quels paramètres, grâce à l'expression : `Lettre (p)` qui suit la définition du constructeur `CourrierInterne(int)`, et qui indique qu'il faut appeler le constructeur `Lettre(int)` avec la valeur de `p`, avant la mise à zéro du champ `tarif`. Le constructeur de `Lettre` va à son tour appeler le constructeur de `ObjetPostal`. Ce mécanisme est similaire à l'emploi de `super` en Java.

Il faut préférer l'initialisation des variables d'instance à leur affectation, surtout quand ce sont des objets. Notez bien que les initialisations doivent être faites dans l'ordre de la déclaration des variables d'instance correspondantes.

3.5 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;
```

```
    double tarif;
public:
    // Constructeur
    ObjetPostal(int p = 20);
    // Destructeur virtuel -- voir ci-après
    virtual ~ObjetPostal() {} // rien de spécial à faire ici
    // Méthodes inline
    bool aValeurDeclaree() const { return (valeur > 0); }
    int getPoids() const { return poids; }
    void recommander() { recommande = true; }
    // Méthode affranchir, implantation par défaut
    // Sera redéfinie dans les sous-classes
    virtual void affranchir() { tarif = 0.0; }
    ...
};
```

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`. Notons aussi qu'il est conseillé dans la plupart des cas de rendre abstraites toutes les classes qui ne sont pas aux feuilles de l'arbre d'héritage.

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. Pour cela, il faut 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`. Ceci est d'autant plus important que l'utilisation de hiérarchies de classes liées par la relation d'héritage est habituellement liée à l'emploi de containers dans lesquels on souhaite pouvoir regrouper des instances de différentes sous-classes de la même classe, ce qui est le cas typique où le destructeur de la classe mère de ces sous-classes doit être déclaré comme virtuel pour mettre en place la liaison dynamique.

3.6 Le mot-clé `this`

On a parfois besoin de désigner dans une fonction membre l'objet qui est manipulé par la méthode. Comment le désigner alors qu'il n'existe aucune variable le représentant dans la fonction membre ? Les fonctions membres travaillent en effet directement sur les attributs définis par la classe. C++ résoud ce problème à l'aide du mot-clé `this`, qui permet à tout moment dans une fonction membre d'accéder à un *pointeur*[2] sur l'objet manipulé. Voici un exemple d'application :

[2] Et non à une référence comme en Java !

```
#include <iostream>
#include "Article.H"

// Fonction présente pour les besoins du test

void testAffichage(Article* unArt)
{
  cout << "Article : " << unArt->nom() << endl;
}

// Méthode de la classe Article

void Article::methodeQuelconque()
{
  // Comment appeler la fonction 'testAffichage' ?
  // Avec le mot-clé 'this' !

  testAffichage(this);
}
```

3.7 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 = 0.53 + (urgent ? 0.2 : 0.0); }
};
```

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

void ParAvion::affranchir()
{
    // affranchissement ordinaire
    Lettre::affranchir();
    // 1,20 EUR de supplement pour courrier aerien
    tarif += 1.2;
}
```

3.8 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 double tarifLettre;
};
...
// Dans la définition de la classe Lettre (Lettre.cpp par exemple)
// Déclaration et initialisation de la variable de classe
double Lettre::tarifLettre = 0.53;
```

3.9 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`[3]. Notons aussi que nous devons faire évoluer notre classe `SacPostal` du fait de l'héritage. En effet, si le tableau `sac` restait un tableau d'objets de type `ObjetPostal`, nous ferions une conversion implicite vers le type `ObjetPostal` chaque fois que nous mettrions une lettre ou un colis dans le sac, et nous perdrions ainsi les caractéristiques propres à la lettre ou au colis... Nous choisissons donc de faire de `sac` un tableau dynamique de pointeurs sur des objets de type `ObjetPostal`[4].

```
// SacPostal.h
   class SacPostal {
   private:
       int nbelts;
       int capacite;
       ObjetPostal** sac;  // tableau de pointeurs
   public:
       SacPostal(int);
       ~SacPostal();
       friend SacPostal& operator+(const SacPostal&, const SacPostal&);
   };
```

```
// 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
                           // je choisis de ne pas détruire les objets postaux
                           // pointés eux-mêmes...
   }

   SacPostal& operator+(const SacPostal& sac1, const SacPostal& sac2) {
       // création d'un gros sac de capacité ad hoc
       SacPostal* grosSac = new SacPostal(sac1.capacite + sac2.capacite);
       // nombre d'éléments de ce gros sac
       grosSac->nbelts = sac1.nbelts + sac2.nbelts;
       // mettre les éléments de sac1 dans grosSac
```

[3] On aurait aussi pu définir l'opérateur + dans la classe `SacPostal`.

[4] Bien entendu, la lecture de la suite de ce polycopié vous montrera des solutions bien plus appropriées, en recourant à des containers comme par exemple un vecteur.

```
        int i = 0;
        for ( ; i < sac1.nbelts ; i++)
            grosSac->sac[i] = sac1.sac[i];
        // puis mettre les éléments de sac2 dans grosSac
        for (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);
    SacPostal sacVandoeuvre(1500);
    ...
    SacPostal sacNancy = sacSeichamps + sacVandoeuvre;
    ...
```

À noter au passage le type retournée par l'opérateur : une référence à un objet de type `SacPostal`, ce qui explique l'instruction `return *grosSac ;` à la fin de la fonction opérateur.

NB : il existe un opérateur qu'il est conseillé de définir de manière systématique, au même titre que le constructeur de copie (cf. § 4.3.1), à savoir l'opérateur de copie :

```
class SacPostal {
  // ...
  SacPostal& operator=(SacPostal const&);
  // ...
};
```

Celui-ci permet d'affecter un objet à un autre objet, en écrivant par exemple :

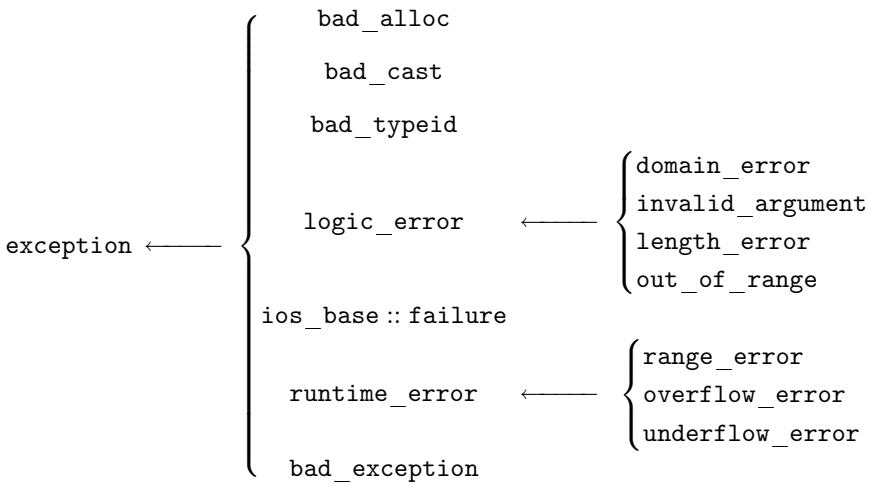
```
  sacVandoeuvre = sacSeichamps;
```

Il s'écrira de la manière suivante (noter l'expression `return *this`, pour permettre l'enchaînement des affectations `a = b = c ;`) :

```
SacPostal& SacPostal::operator=(SacPostal const& unSac) {
  // libérer le tableau alloué jusqu'ici
  delete [] sac;
  // nouvelle capacité
  capacite = unSac.capacite;
  // créer un nouveau tableau
  sac = new ObjetPostal*[capacite];
  // le remplir
  nbelts = unSac.nbelts;
  for (int i = 0 ; i < nbelts ; i++) {
    sac[i] = unSac.sac[i];
  }
  return *this;
}
```

3.10 Les exceptions

Un mécanisme normalisé de gestion d'exceptions a été ajouté à C++ il y a quelques années. Il est analogue à bien des égards à celui de Java. Toute exception est instance d'une classe qui dérive de la classe de base `exception`. La hiérarchie des exceptions standard est la suivante :



Comme en Java, une exception est provoquée par l'instruction `throw` et traitée par la construction `try` / `catch`.

Chapitre 4

La généricité

La notion de *généricité* permet de définir des modules paramétrés par le type qu'ils manipulent. Un module générique n'est alors pas directement utilisable : c'est plutôt un *patron* de module qui sera « instancié » par les types paramètres qu'il accepte.

4.1 Les fonctions génériques

Supposons que l'on souhaite écrire une fonction `min` qui accepte deux paramètres et qui renvoie la plus petite des deux valeurs qui lui est fournie. On désire bénéficier de cette fonction pour certains types simples disponibles en C++ (`int, char, float, double`). La première solution pour atteindre ce but est d'utiliser la surcharge et de définir 4 fonctions `min`, une pour chacun des types considérés :

```
int min (int a, int b) {
  return ((a < b)? a : b);
}

float min (float a, float b) {
  return ((a < b)? a : b);
}

double min (double a, double b) {
  return ((a < b)? a : b);
}

char min (char a, char b) {
  return ((a < b)? a : b);
}
```

Lors d'un appel à la fonction `min`, le type des paramètres est alors considéré et l'implantation correspondante est finalement appelée. Ceci présente cependant quelques inconvénients :

- La définition des 4 fonctions mène à des instructions identiques, qui ne sont différenciées que par le type des variables qu'elles manipulent. On s'aperçoit ici que plus qu'une fonction, on souhaiterait exprimer une méthode, valable pour n'importe quel type manipulé : la fonction `min` est la fonction qui renvoie le plus petit des paramètres qui lui est fourni. Cet élément est déterminé grâce à l'opérateur `<` qui établit une relation d'ordre sur le type d'élément considéré.
- Si on souhaite étendre la définition de cette fonction à de nouveaux types, il faut définir une nouvelle implantation de la fonction `min` par type considéré.

Une autre solution est de définir une fonction générique, dite aussi en C++ fonction *template*. Cette définition définit en fait un patron de fonction, qui est *instancié* par un type de données (ici le type `T`) pour produire une fonction par type manipulé :

```
template <class T>
T min (T a, T b)
{
  return ((a < b)? a : b);
}

int main()
{
  int a = min(10, 20);          // int min(int, int)
  float b = min(10.0, 25.0);    // float min(float, float)
  char c = min('a', 'W');       // char min(char, char)
}
```

Il n'est donc plus nécessaire de définir une implantation par type de données. De plus, la fonction `min` est valide avec tous les types de données dotés de l'opérateur `<`. On définit donc bien plus qu'une fonction, on définit une méthode permettant d'obtenir une certaine abstraction en s'affranchissant des problèmes de type. Quelques remarques :

- Il est possible de définir des fonctions *template* acceptant plusieurs types de données en paramètre. Chaque paramètre désignant une classe est alors précédé du mot-clé `class`, comme dans l'exemple : `template <class T, class U> ....`
- Chaque type de données paramètre d'une fonction *template* doit être utilisé dans la définition de cette fonction.
- Pour que cette fonctionnalité soit disponible, les fonctions génériques doivent être définies dans des fichiers d'interface (*headers*)[1]. Les fonctions *template* sont en effet expansées elles aussi. Ainsi, chaque appel fait à ce genre de fonctions est remplacé, à la précompilation, par le code source correspondant à la fonction.

4.2 Les classes génériques

Il est également possible de définir des classes génériques ou *template*, c'est-à-dire paramétrées par un type de données. Cette technique évite ainsi de définir plusieurs classes similaires pour décrire un même concept appliqué à plusieurs type de données différents. Elle est largement utilisée pour définir tous les types de containers (comme les listes, les tables, les piles, etc.), mais aussi des algorithmes génériques par exemple.

La syntaxe permettant de définir une classe générique est similaire à celle qui permet de définir des fonctions génériques. Ainsi, la classe `Point` est un exemple de classe générique, portant sur des points dont la précision de représentation (à partir d'entiers, de réels, etc.) est le type paramètre de la classe :

```
template <class T>
class Point {
protected:
  T _x;      // Abcisse
  T _y;      // Ordonnée

public :
  // Constructeur par défaut
  Point() : _x(0), _y(0) {}

  // Constructeur
  Point(T x, T y) : _x(x), _y(y) {}

  // Accès à x
  const T x() const {return _x;}
```

[1] Les fonctions *inline* et *template* sont ainsi les **seules** fonctions à être définies dans les interfaces. Toutes les autres sont définies dans les fichiers d'implantation (`.cpp`) et sont seulement déclarées dans les interfaces.

```
  // Accès à y
  const T y() const {return _y;}

  // Translation
  void translation(T x, T y);
};

template <class T> void
Point<T>::translation(T x, T y)
{
  _x += x;
  _y += y;
}
```

On peut ensuite utiliser cette classe en instanciant le type générique :

```
#include "Point.H"

int main()
{
  Point<int> pointEntier(2, 3);
  Point<float> pointReel(3.14, 2.27);

  pointReel.translation(pointEntier.x(), pointEntier.y());
}
```

Quelques remarques :

- Comme dans le cas des fonctions génériques, tout le code source correspondant à des classes génériques (y compris la définition de leurs méthodes) doit se trouver dans l'interface de la classe correspondante.
- Une classe générique permet de définir des attributs, des paramètres ou des valeurs de retour de méthodes génériques. De façon réciproque, pour pouvoir définir des entités génériques à l'intérieur d'une classe, la classe doit elle-même être générique.
- Attention à la syntaxe des méthodes génériques définies en dehors du corps de la classe. La définition se fait de la manière suivante :
  `template <class T> typeRetour nomClasse<T>::nomMéthode(liste_paramètres)`
  Voir par exemple ci-dessus la définition de la méthode `translation` de la classe générique `Point`.

4.3 La bibliothèque STL

La bibliothèque STL (*Standard Template Library*[2]) est certainement l'un des atouts de C++. Cette bibliothèque fournit un ensemble de composants C++ bien structurés qui marchent de façon cohérente et peuvent aussi être adaptés facilement. En effet, il est possible d'utiliser les structures de données proposées par STL avec des algorithmes personnels, les algorithmes de la bibliothèque avec des structures de données personnelles, ou d'utiliser toutes les composantes STL ! Lors de sa conception, l'accent a été mis sur l'efficacité et sur l'optimisation des composants, ce qui en fait un outil très puissant.

Il faut bien noter que la STL est fondée sur la séparation entre données et opérations. De ce point de vue, on peut considérer qu'elle contredit conceptuellement l'un des grands axiomes de la programmation objet, à savoir l'encapsulation dans une même entité des données et des opérations qui les manipulent ! Il faut en fait voir la généricité comme une approche « orthogonale » à l'approche objet classique.

Ce chapitre présente les généralités liées à STL. Pour en tirer pleinement partie, une bonne documentation s'avère nécessaire. Un des meilleurs ouvrages de référence de cette bibliothèque est celui de Musser & Saini [8]. On consultera aussi de manière utile d'autres références comme le livre

[2] Bibliothèque standard générique.

de Josuttis [3] qui traite de l'ensemble de la bibliothèque C++, ou le livre de Meyers dédié à la STL [7].

La STL contient cinq types de composants : des containers, des itérateurs, des algorithmes, des objets-fonctions et des adaptateurs. Nous nous intéressons dans ce chapitre aux trois premiers composants.

4.3.1 Les containers

Les containers sont des objets qui permettent de stocker d'autres objets. Ils sont décrits par des classes génériques représentant les structures de données logiques les plus couramment utilisées : les listes, les tableaux, les ensembles... Ces classes sont dotées de méthodes permettant de créer, de copier, de détruire ces containers, d'y insérer, de rechercher ou de supprimer des éléments. La gestion de la mémoire, c'est-à-dire l'allocation et la libération de la mémoire, est contrôlée directement par les containers, ce qui facilite leur utilisation. L'exemple suivant présente une application où les valeurs entières saisies par un utilisateur sont stockées dans une liste et dans un tableau :

```
#include <iostream>
#include <vector>
#include <list>

int main()
{
  vector<int> tableauEntiers;  // Crée un tableau d'entiers vide
  list<int> listeEntiers;      // Crée une liste d'entiers vide
  int unEntier;

  // Saisie des entiers
  cout << "Saisir le prochain entier (-1 pour finir) : ";
  cin >> unEntier;

  while (unEntier != -1) {
    tableauEntiers.push_back(unEntier);
    listeEntiers.push_back(unEntier);

    cout << "Saisir le prochain entier (-1 pour finir) : ";
    cin >> unEntier;
  }

  // Nombre d'éléments des containers
  cout << "Il a y " << tableauEntiers.size()
       << " éléments dans le tableau" << endl;

  cout << "Il a y " << listeEntiers.size()
       << " éléments dans la liste" << endl;

  // Accès à des éléments
  cout << "Premier élément du tableau : "
       << tableauEntiers.front() << endl;

  cout << "Premier élément de la liste : "
       << listeEntiers.front() << endl;

  int milieu = tableauEntiers.size() / 2;

  cout << "Élément de milieu de tableau : "
       << tableauEntiers[milieu] << endl;
}
```

Voici un exemple d'exécution de ce programme :

```
Saisir le prochain entier (-1 pour finir) : 4
Saisir le prochain entier (-1 pour finir) : 5
Saisir le prochain entier (-1 pour finir) : 3
Saisir le prochain entier (-1 pour finir) : 7
Saisir le prochain entier (-1 pour finir) : 6
Saisir le prochain entier (-1 pour finir) : 3
Saisir le prochain entier (-1 pour finir) : -1
Il a y 6 éléments dans le tableau
Il a y 6 éléments dans la liste
Premier élément du tableau : 4
Premier élément de la liste : 4
Élément de milieu de tableau : 7
```

Quelques remarques sur les containers et sur l'exemple présenté :

- Un certain nombre de méthodes sont disponibles sur tous les types de containers, ce qui permet d'homogénéiser leur utilisation. C'est le cas par exemple de la méthode `push_back` qui insère un nouvel élément à la fin d'un container.
- D'autres méthodes ou opérateurs sont disponibles en fonction du type de container utilisé. L'opérateur `[]` est disponible sur les objets de type `vector`, mais pas sur ceux de type `list` : il permet d'accéder directement à un élément. Plus de précisions sur ces méthodes sont disponibles au § B.1.
- L'utilisateur n'a pas à se soucier de l'allocation ou de la libération de la mémoire. C'est vrai lors de l'insertion d'éléments et aussi à la fin du programme : aucune instruction particulière n'est nécessaire pour restituer la mémoire occupée par les containers. À la sortie du bloc dans lequel ils sont définis, leur destructeur se charge de libérer toutes les ressources occupées.
- Les containers peuvent manipuler n'importe quel type de données, à partir du moment où la classe correspondante est dotée d'un certain nombre de méthodes nécessaires à STL (pour une classe `X`) :
  - `X()` : un constructeur par défaut,
  - `X(const X&)` : un constructeur par copie,
  - `operator=(const X&)` : l'opérateur d'affectation,
  - `operator==(const X&)` : l'opérateur d'égalité,
  - `operator<(const X&)` : l'opérateur inférieur (utile uniquement pour les tris).

  Ainsi, un type bien défini pour être manipulable dans un container comprendra au minimum les éléments suivants :

```
class MonType {
    // Mettre ici données privées et protégées
public:
    // Constructeurs indispensables
    MonType();
    MonType(MonType const&);
    // + autres constructeurs éventuels

    // Le destructeur -- ne pas oublier virtual
    // si MonType peut avoir des sous-classes
    virtual ~MonType();

    MonType& operator=(MonType const&);
    bool operator==(MonType const&) const;
    bool operator<(MonType const&) const;
    // autres méthodes et opérateurs ...
};
```

Les différentes sortes de containers disponibles sont :

- `vector` : container implantant les tableaux, qui autorise les accès directs sur ses éléments. Les opérations de mise à jour (insertion, suppression) sont réalisées en un temps constant à la fin du container, et en un temps linéaire (dépendant du nombre d'éléments) aux autres

endroits.

- `list` : container implantant les listes doublement chaînées, dédié à la représentation séquentielle de données. Les opérations de mise à jour sont effectuées en un temps constant à n'importe quel endroit du container.
- `deque` : container similaire au `vector`, effectuant de plus les opérations de mise à jour en début de container en un temps constant.
- `set` : container implantant les ensembles où les éléments ne peuvent être présents au plus qu'en un seul exemplaire.
- `multiset` : container implantant les ensembles où les éléments peuvent être présents en plusieurs exemplaires.
- `map` : container implantant des ensembles où un type de données appelé *clé* est associé aux éléments à stocker. On ne peut associer qu'une seule valeur à une clé unique. On appelle aussi ce type de container *tableau associatif*.
- `multimap` : container similaire au `map` supportant l'association de plusieurs valeurs à une clé unique.
- `stack` : adaptateur permettant de donner à un container le comportement d'une pile, c'est-à-dire que le premier élément qui entre est le dernier qui sort.
- `queue` : adaptateur permettant de donner à un container le comportement d'une file, c'est-à-dire que le premier élément qui entre est aussi le premier qui sort.
- `priority_queue` : adaptateur permettant de donner à un container le comportement d'une file avec priorités, une variante de la file où certains éléments peuvent avoir des priorités supplémentaires à d'autres pour sortir de la file.

Une carte de référence sur les containers existants et sur les méthodes dont ils sont dotés est disponible à l'annexee B.

4.3.2 Les itérateurs

Les itérateurs sont une généralisation des pointeurs, ce qui permet au programmeur de travailler avec des containers différents de façon uniforme. Ils permettent de spécifier une position à l'intérieur d'un container, peuvent être incrémentés ou déréférencés (à la manière des pointeurs utilisés avec l'opérateur de déréférencement `'*'`) et deux itérateurs peuvent être comparés. Tous les containers sont dotés d'une méthode `begin` qui renvoie un itérateur sur le premier de leurs éléments, et d'une méthode `end` qui renvoie un itérateur sur une place se trouvant *juste après* le dernier de leurs éléments. On ne peut ainsi pas déréférencer l'itérateur renvoyé par la méthode `end`. Voici un exemple d'utilisation des itérateurs :

```
#include <iostream>
#include <list>

int main()
{
  list<int> lesEntiers;

  // Ici, des instructions pour initialiser la
  // liste des entiers

  ...

  // Affichage des éléments contenus dans la liste

  list<int>::iterator it;

  for (it = lesEntiers.begin(); it != lesEntiers.end(); it++)
    cout << *it << endl;
}
```

Ces itérateurs sont dotées de méthodes permettant de les manipuler (cf. § B.2). Il existe une hiérarchie d'itérateurs, qui n'est pas liée à un quelconque héritage :

- Les itérateurs d'entrée (*input iterators*) : ils permettent d'accéder séquentiellement à des sources de données. Cette source peut-être un container, un flot.
- Les itérateurs de sortie (*output iterators*) : ils permettent de préciser la localisation d'une destination permettant de stocker des données. Cette source peut-être un container, un flot.
- Les itérateurs à sens-unique (*forward iterators*) : ils sont dotés de toutes les méthodes des itérateurs d'entrée et de sortie. Ils sont utilisés pour parcourir séquentiellement une séquence de données dans un sens. Ils ne peuvent pas être utilisés pour effectuer des retours en arrière.
- Les itérateurs à double-sens (*bidirectional iterators*) : ils sont dotés de toutes les méthodes des itérateurs à sens-unique. Ils sont également utilisés pour effectuer des parcours séquentiels de données, qu'ils peuvent effectuer dans les deux sens.
- Les itérateurs à accès direct (*random-access iterators*) : ils sont dotés de toutes les méthodes des itérateurs à double-sens. Ils permettent d'accéder directement à des valeurs contenues dans un container, sans être obligé d'y accéder séquentiellement.

Tous les containers disponibles sous STL fournissent au moins des itérateurs à double-sens, et certains fournissent des itérateurs à accès direct (cf. § B.1) pour les itérateurs par défaut renvoyés par chaque type de container.

4.3.3 Les algorithmes

Les algorithmes sont des fonctions C++ génériques qui permettent d'effectuer des opérations sur les containers. Afin de pouvoir s'appliquer à plusieurs types de containers, les algorithmes ne prennent pas de containers en arguments, mais des itérateurs qui permettent de désigner une partie ou tout un container. De ce fait, il est même possible d'utiliser ces algorithmes sur des objets qui ne sont pas des containers. On peut par exemple utiliser un `istream_iterator` (cf. § B.2) comme paramètre d'un algorithme, qui va alors s'appliquer à l'entrée standard. Certains algorithmes ne nécessitent que des itérateurs de base (d'entrée ou de sortie), et d'autres nécessitent des itérateurs plus évolués, comme la fonction `sort`[3] (effectuant un tri) qui nécessite un itérateur à accès direct.

Les algorithmes disponibles sont décrits en annexe B.3. Pour les utiliser, il suffit d'inclure l'en-tête `algorithm`. Un exemple d'utilisation des algorithmes est présenté ci-dessous :

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
  vector<int> tableauEntiers;  // Crée un tableau d'entiers vide
  int unEntier;

  // Saisie des entiers

  cout << "Saisir le prochain entier (-1 pour finir) : ";
  cin >> unEntier;

  while (unEntier != -1) {
    tableauEntiers.push_back(unEntier);

    cout << "Saisir le prochain entier (-1 pour finir) : ";
    cin >> unEntier;
  }

  // Tri du tableau

  sort(tableauEntiers.begin(), tableauEntiers.end());

  // Affichage des éléments triés
```

[3] Du coup, cet algorithme n'est pas applicable sur les listes qui ne fournissent que des itérateurs à double sens. C'est pour cette raison que la classe `list` est dotée d'une méthode `sort`.

```
  vector<int>::iterator it;

  for (it = tableauEntiers.begin(); it != tableauEntiers.end(); it++)
    cout << *it << " ";

  cout << endl;
}
```

Voici le résultat produit à partir de cet exemple :

```
Saisir le prochain entier (-1 pour finir) : 5
Saisir le prochain entier (-1 pour finir) : 3
Saisir le prochain entier (-1 pour finir) : 8
Saisir le prochain entier (-1 pour finir) : 10
Saisir le prochain entier (-1 pour finir) : 3
Saisir le prochain entier (-1 pour finir) : 6
Saisir le prochain entier (-1 pour finir) : 9
Saisir le prochain entier (-1 pour finir) : -1
3 3 5 6 8 9 10
```

4.4 Programmer avec la STL

Le bon usage de la STL nécessite une remise en cause de beaucoup d'habitudes prises en programmation plus traditionnelle. On se reportera avec profit au livre de Meyers [7], par exemple, pour apprendre à tirer le maximum de profit de la puissance de cette approche générique. Voici très simplement quelques conseils succincts tirés de ce livre, auquel on se référera pour les détails :

- Pour tester si un container est vide, utiliser `empty()` au lieu de regarder la taille du container avec la méthode `size()`.
- Préférer les fonctions membres travaillant sur un intervalle aux fonctions travaillant élément par élément.
- Préférer `vector` et `string` aux tableaux alloués dynamiquement.
- Utiliser `reserve` pour éviter les réallocations inutiles.
- Les méthodes de comparaison doivent toujours retourner `false` pour des valeurs égales.
- Faire suivre les algorithmes ou méthodes `remove_*` par `erase` si on veut vraiment effacer les objets et pas seulement les déplacer.
- Préférer les appels à des algorithmes aux boucles manuelles du genre `for` ou `while`.
- Préférer les fonctions membres aux algorithmes de même nom, quand on a le choix.

4.5 Quelques utilitaires de la bibliothèque standard

Mis à part la STL à proprement parler, la bibliothèque standard de C++ fournit un grand nombre de fonctionnalités plus générales. Nous en citons quelques-unes seulement ; le lecteur qui voudra une vision complète des ressources de la bibliothèque pourra se référer à un livre de référence tel que celui de Josuttis [3], ou à de l'information en ligne comme par exemple à l'adresse `http://www.cplusplus.com/`.

4.5.1 Les paires

La classe `pair` fournit un moyen générique de traiter une paire de valeurs comme une entité unique. Les classes de containers `map` et `multimap` utilisent en particulier des paires pour gérer les couples clé/valeur.

La fonction générique `make_pair` permet de créer une paire à partir de deux valeurs ; ainsi, on pourra écrire :

```
map<int, string> codesPostaux;

// ...

codesPostaux.insert(make_pair(54000, "Nancy"));
```

On accède respectivement à la première et à la deuxième valeur d'une paire par les variables `first` et `second`. Ainsi, en gardant l'exemple courant, on peut parcourir la *map* `codePostaux` par un itérateur et accéder comme suit aux valeurs associées :

```
map<int, string>::iterator it = codesPostaux.find(54280);

if (it != codesPostaux.end())
    cout << (*it).first << " - " << (*it).second;
```

À noter au passage la notation pour accéder aux éléments ou méthodes d'un objet « pointé » par l'itérateur. En fait, les compilateurs les plus récents permettent d'utiliser la notation plus naturelle (pour un programmeur C ou C++) `it->first`, mais les compilateurs plus anciens ne l'autorisent pas.

4.5.2 Opérateurs de comparaison supplémentaires

Nous avons vu que pour manipuler un type avec des containers, celui-ci doit comporter la définition des opérateurs < et ==. Le fichier *header* `utility` définit l'espace de noms `rel_ops` avec les opérateurs génériques suivants :

```
namespace std {
  namespace rel_ops {
    template <class T>
    inline bool operator!= (const T& x, const T& y) {
      return !(x == y);
    }

    template <class T>
    inline bool operator> (const T& x, const T& y) {
      return y < x;
    }

    template <class T>
    inline bool operator<= (const T& x, const T& y) {
      return !(y < x);
    }

    template <class T>
    inline bool operator>= (const T& x, const T& y) {
      return !(x < y);
    }
  }
}
```

Ainsi, en utilisant le *namespace* `std::rel_ops` on disposera pour tout type bien défini de l'ensemble des opérateurs de comparaison classiques.

4.5.3 La classe `string`

La bibliothèque standard de C++ définit le type générique `basic_string` ainsi que ses instanciations standard `string` et `wstring`. Ce dernier permet d'utiliser des jeux de caractères plus

étendus que ce qu'on peut faire avec le type standard `char`, par exemple Unicode ou des jeux de caractères asiatiques.

Nous nous contentons de donner dans la figure 4.1 un aperçu des opérations disponibles les plus courantes sur une chaîne de caractères.

Opération	**Effet**
Constructeurs	
`string s`	Crée une chaîne vide
`string s(str)`	Crée une copie de `str`
`string s(beg,end)`	Crée une chaîne initialisée par les caractères que l'on trouve entre les itérateurs `beg` et `end`
etc.	*il y a beaucoup d'autres constructeurs que nous ne détaillons pas ici*
`=`, `assign()`	Affectation
`+=`, `append()`, `push_back()`	Concaténation de caractères
`+`	Concaténation de chaînes
`==`, `!=`, `<`, etc.	Comparaisons de chaînes
`clear()`	Efface tous les caractères (la chaîne devient vide)
`empty()`	Teste si la chaîne est vide
`size()`, `length()`	Taille de la chaîne (nombre de caractères)
`[]`, `at()`	Accède à un caractère donné par son indice dans la chaîne
`data()`	Donne un tableau de caractères construit à partir de la chaîne
`substr()`	Extraction de sous-chaînes
`begin()`, `end()`	Fournit un accès de type itérateurs sur une chaîne ; de ce fait, on peut parcourir une chaîne comme on parcourt un container, et utiliser les algorithmes génériques de la `Stl` sur les chaînes
Fonctions de recherche	
`find()`	Trouve la première occurrence d'un caractère
`rfind()`	Trouve la dernière occurrence d'un caractère
`find_first_of()`	Trouve la première occurrence d'un des caractères de la chaîne donnée en paramètre
etc.	

FIG. 4.1 – Opérations les plus courantes de la classe `string`.

4.6 Autres bibliothèques

La bibliothèque standard n'a pas vocation à contenir l'ensemble des fonctionnalités dont vous pouvez avoir besoin. Elle représente uniquement les fonctionnalités de base sur lesquelles le comité de normalisation de C++ est parvenu à un accord. Il existe de très nombreuses bibliothèques complémentaires, offrant des services dans un ensemble de domaines généraux ou spécifiques.

Une mention particulière peut être donnée à BOOST[4] qui est un ensemble de bibliothèques répondant à des besoins généraux (gestion de graphes, par exemple) mais qui n'ont pas fait l'objet d'un consensus assez général pour être incluses dans la biblothèque standard. Ces bibliothèques n'en représentent pas moins des ensembles stables et robustes de fonctionnalités et je ne peux que vous encourager à aller voir si vous y trouvez les fonctions et classes dont vous avez besoin, avant de décider de les implanter vous-mêmes.

[4] `http ://www.boost.org/`.

Chapitre 5

Environnement de développement C++

5.1 Compilation

5.1.1 Compilation d'une classe C++

Le *compilateur* traduit le code C++ en code objet (ou code « machine »). Pendant ce cours, nous utiliserons le compilateur Gnu C++, appelé `g++`. Ce compilateur a beaucoup d'options, et nous vous incitons à vous reporter aux pages de manuel pour plus d'information. Toutefois, voici les options les plus utiles :

`-c` — Compiler les fichiers sources, mais sans édition de liens. Le compilateur produit un fichier *objet* `.o` pour chaque fichier source.

`-o` ***fichRes*** — Donner le nom *fichRes* au fichier exécutable produit par la compilation et l'édition de liens. Quand cette option n'est pas employée, le nom d'exécutable par défaut est `a.out`.

`-g` — engendrer de l'information pour le débogage, à utiliser avec un débogueur comme `gdb` (ou son interface utilisateur `ddd`).

Pour compiler une classe C++ `Toto` (que nous supposerons correcte), la commande à lancer est :

```
g++ -c Toto.cpp
```

Si la compilation s'achève sans erreur, vous récupérez dans votre répertoire un fichier objet appelé `Toto.o`.

5.1.2 Compilation d'un programme C++

Un programme C++ est un fichier qui

1. inclut les *headers* de *toutes* les classes et bibliothèques employées,
2. comporte une fonction `main` : `int main() ...corps du programme...`.

Pour compiler un programme C++, appelons-le `monprog.cpp`, voici la marche à suivre :

```
g++ monprog.cpp C_1.o C_2.o ... C_n.o -o monprog
```

où `C_1.o C_2.o ... C_n.o` sont les fichiers objets des classes compilées au préalable, et utilisées par le programme `monprog.cpp`. L'option `-o` indique que le fichier produit par la compilation doit s'appeler `monprog`. C'est un *exécutable* pour l'architecture et le système sur lesquels vous avez compilé[1] le programme. Le compilateur a besoin du nom des fichiers objets pour les passer à *l'édition de liens*, qui « assemble » les différents morceaux compilés pour en faire un fichier exécutable unique.

[1] Le compilateur C++ produisant directement du code machine, et non du *bytecode* comme en Java, un programme compilé sous un système et sur une architecture n'a aucune raison de s'exécuter sur d'autres architectures et/ou systèmes !

5.1.3 Construire des bibliothèques

Si vous créez un certain nombre de classes, vous ressentirez rapidement le besoin de réunir les versions compilées de ces classes dans une *bibliothèque*. En Unix ou Linux, pour réunir des fichiers objets, on utilisera typiquement la commande `ar`, qui permet de créer des fichiers « archives » :

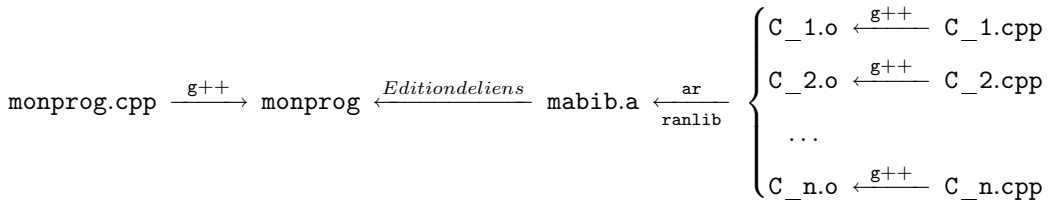
```
ar r mabib.a C_1.o C_2.o ... C_n.o
```

L'option `r` indique qu'il faut *remplacer* les versions existantes éventuelles des fichiers `C_i.o` par les nouvelles versions fournies ; s'il n'y a pas de version existante, le fichier est *ajouté* à l'archive.

Pour faire d'un fichier archive une bibliothèque utilisable, il faut construire un index de toutes les fonctions et variables définies par les différents fichiers objets qu'elle contient. Ceci est fait par la commande `ranlib`, qu'on appelle souvent systématiquement après `ar` :

```
ranlib mabib.a
```

La figure suivante résume les différentes étapes que nous avons décrites :



5.2 Le préprocesseur

Les fichiers *headers* sont ajoutés au programme à compiler par les instructions d'inclusion du préprocesseur. Le préprocesseur traite le fichier source *avant* la compilation proprement dite. Il effectue surtout des remplacements lexicographiques sur le texte à compiler.

Les instructions du préprocesseur commencent toutes par le caractère `#` ; nous présentons ci-après les plus utiles d'entre elles.

5.2.1 L'instruction `#include`

L'instruction `#include` a la syntaxe suivante :

```
#include <some_lib>
#include "my_file.hh"
```

La première ligne indique qu'on inclut `some_lib`, qui est le fichier *header* d'une bibliothèque, et qu'on trouvera dans un chemin appartenant à une liste connue[2]. La seconde ligne indique l'inclusion d'un fichier *header* fourni par l'utilisateur, avec un chemin relatif à l'emplacement du fichier source qui contient l'instruction d'inclusion.

Prenons comme exemple que nous voulons construire une classe `ManuscritAncien`, définie par un fichier `ManuscritAncien.hh` contenant l'interface de la classe, et un fichier `ManuscritAncien.cpp` contenant la mise en œuvre de la classe. Le fichier `ManuscritAncien.hh` pourrait être de la forme :

```
#include "Ouvrage.hh"
#include "Livre.hh"

class ManuscritAncien : public Livre  {
    ...
    corps de ManuscritAncien.hh
    ...
};
```

[2]Grâce à l'option `-I` du compilateur, vous pouvez ajouter vos propres chemins aux chemins prédéfinis.

et le fichier `ManuscritAncien.cpp` :

```
#include <iostream>
#include "ManuscritAncien.hh"
....
body of ManuscritAncien.cpp
....
```

S'il manque l'inclusion (éventuellement en cascade) des interfaces de *toutes* les bibliothèques et classes utilisées, la compilation échouera...

NB : il faut minimiser les dépendances entre modules en n'incluant que les directives `#include` strictement nécessaires à la compilation d'un module donné.

5.2.2 Inclusion conditionnelle

L'instruction `#if(n)def-define-endif` sert (entre autres) à éviter les inclusions multiples du même fichier :

```
#ifndef __POLYCOPIE_HH_INCLUDED__
#define __POLYCOPIE_HH_INCLUDED__
// définition de Polycopie.hh
#endif /* __POLYCOPIE_HH_INCLUDED__ */
```

teste si `__POLYCOPIE_HH_INCLUDED__` a déjà été défini C'est une *constante* du préprocesseur (à écrire en majuscules). Si `__POLYCOPIE_HH_INCLUDED__` n'a pas encore été défini, `// définition de Polycopie.hh` est inclus et traité, sinon il n'est pas inclus.

Nous ajoutons `#define __POLYCOPIE_HH_INCLUDED__` juste après l'instruction `#ifndef` pour garantir que la constante `__POLYCOPIE_HH_INCLUDED__` sera définie la première fois que le fichier *header* est mentionné dans une directive d'inclusion.

La directive `#ifdef` est souvent utilisée pour faire de la compilation conditionnelle de morceaux de code, par exemple à des fins de débogage :

```
int main () {
#ifdef DEBUG
  cout << "begin of main()"<< endl;
#endif
 ....
 // body of main()
 ....
#ifdef DEBUG
  cout << "end of main()"<< endl;
#endif
}
```

5.3 Où trouver un compilateur C++ ?

Il existe un vaste choix de compilateurs C++ commerciaux. Des solutions libres et gratuites existent également, quelques pointeurs sont donnés ici :

- **Sous Windows** :
  - Cygwin (`http://www.cygwin.com/`). Un environnement qui permet d'émuler Linux sous Windows, et donc en particulier de disposer de l'excellent compilateur G++/Gcc.
  - Dev-C++ (`http://www.bloodshed.net/devcpp.html`). C'est un système complet de développement C/C++, tournant directement sous Windows directement. Il comprend un éditeur multi-fichiers, un compilateur, un gestionnaire de projet et un débogueur. Le compilateur de base est Gcc.

Pour les amateurs d'Eclipse, il faut noter que le *plug-in* Cdt offre un environnement de développement pour C++. Associé à Cygwin ou à MinGW[3], un environnement minimal Gnu sous Windows, pour accéder au compilateur G++, il permet de travailler dans un cadre connu, pour ceux qui ont pris l'habitude d'Eclipse pour développer en Java.

– **Sous Linux** : Gcc/G++. Les sources sont téléchargeables à partir de `http://gcc.gnu.org/`, mais l'installation est plus délicate. Il est livré en standard avec certains systèmes Unix (comme Linux), ce qui permet d'éviter l'étape d'installation. Les débogueurs du monde Unix livrés en standard ne sont généralement pas très conviviaux, mais il est possible de récupérer un débogueur graphique (Ddd) à l'adresse `http://www.gnu.org/software/ddd/`.
Les personnes travaillant sous Linux utiliseront généralement leur éditeur de texte favori (comme Emacs ou XEmacs) pour compléter cet environnement. Il existe cependant différents environnements de développement intégré libres sous Linux, comme KDevelop, dont l'adresse d'accueil est `http://www.kdevelop.org/`.

[3] `http ://sourceforge.net/projects/mingw/`.

Annexe A

Format des entrées-sorties

Ce chapitre, dû à Philippe Dosch [1] que je remercie une fois de plus, donne quelques précisions supplémentaires sur les entrées-sorties en C++.

A.1 La classe `ios`

La classe `ios` est la classe de base des classes représentant les flots. Toutes les classes de flot présentées ci-dessous héritent donc de cette classe et des méthodes qu'elle définit (non exhaustif) :

- `int ios::good()` : retourne une valeur nulle s'il y a eu un échec lors de la dernière opération d'entrée-sortie, et une valeur non nulle sinon.
- `int ios::fail()` : retourne le contraire de la méthode précédente.
- `int ios::eof()` : retourne une valeur non nulle si la fin de fichier est atteinte, et la valeur nulle sinon.

La classe `ios` définit aussi un certain nombre de méthodes relatives au format des informations manipulées (non exhaustif). Voici un exemple d'utilisation :

```
#include <iostream>

int main()
{
  cout << "Largeur standard : "
       << cout.width() << endl;     // Affiche : 0

  cout.width(10);
  cout.fill('#');

  cout << 654 << endl;              // Affiche : #######654
}
```

- `int ios::width(int n)` : positionne la largeur du champ (le nombre de caractères) de sortie.
- `int ios::width()` : retourne la largeur du champ de sortie.
- `char ios::fill(char c)` : positionne le caractère de remplissage (utilisé pour remplir toute la largeur des champs).
- `char ios::fill()` : retourne le caractère de remplissage.
- `int ios::precision(int p)` : positionne la précision, c'est-à-dire le nombre de caractères (y compris le point) qu'occupe un réel.
- `int ios::precision()` : retourne la précision.

Enfin, cette classe définit un certain nombre d'opérations qui peuvent être utilisées sans les préfixer du flot sur lequel on travaille, et autorise ainsi l'écriture de lignes du type :

```
cout << "La valeur octale de 154 est : " << oct << 154 << endl ;
```

Parmi ces opérations, on trouve :

- `endl` : écrit un `'\n'` et vide le tampon du flot.

- `ends` : écrit un `'\0'` et vide le tampon du flot.
- `flush` : vide le tampon du flot.
- `dec` : la prochaine opération d'entrée-sortie se fera en décimal.
- `hex` : la prochaine opération d'entrée-sortie se fera en hexadécimal.
- `oct` : la prochaine opération d'entrée-sortie se fera en octal.
- `ws` : saute les espaces lors d'une lecture sur un flot d'entrée.
- `setfill(int c)` : positionne le caractère de remplissage pour la prochaine opération d'entrée-sortie.
- `setprecision(int p)` : positionne la précision de la prochaine opération d'entrée-sortie à `p` chiffres.
- `setw(int n)` : positionne la largeur de la prochaine entrée-sortie à `n` caractères.

A.2 La classe `ostream`

Cette classe est dédiée aux sorties formatées ou non. Un objet de type `ostream` est défini par défaut dans tout programme C++ : c'est la variable `cout`. La classe `ostream` surcharge l'opérateur `<<` pour tous les types prédéfinis du C++. Il faut, si besoin est, le surcharger pour les nouveaux types introduits (cf. § 3.9). En plus de l'opérateur `<<`, la classe `ostream` est dotée des méthodes suivantes (non exhaustif) :

- `ostream &ostream::put(char c)` : insère un caractère dans le flot.
  Exemple : `cout.put('\n')`.
- `ostream &ostream::write(const char* s, int n)` : insère `n` caractères dans le flot.
- `ostream &ostream::flush()` : vide le tampon du flot.

A.3 La classe `istream`

Cette classe est dédiée aux entrées formatées ou non. Un objet de type `istream` est défini par défaut dans tout programme C++ : c'est la variable `cin`. La classe `istream` surcharge l'opérateur `>>` pour tous les types prédéfinis du C++. Il faut, si besoin est, le surcharger pour les nouveaux types introduits (cf. § 3.9). En plus de l'opérateur `>>`, la classe `istream` est dotée des méthodes suivantes (non exhaustif) :

- `int istream::get()` : retourne la valeur du caractère lu (EOF si la fin du flot est atteinte).
- `istream &istream::get(char &c)` : extrait le premier caractère du flot (même si c'est un espace) et le place dans `c`.
- `int &istream::peek()` : lit le prochain caractère du flot sans l'enlever du flot (renvoie EOF si la fin du flot est atteinte).
- `istream &istream::get(char* ch, int n, char delim = '\n')` : extrait `(n - 1)` caractères du flot et les place à l'adresse `ch` (le tampon). La lecture s'arrête éventuellement après le délimiteur s'il est rencontré.
- `istream &istream::getline(char* ch, int n, char delim = '\n')` : identique à la méthode précédente sans placer le délimiteur dans le tampon.
- `istream &istream::read(char* ch, int n)` : extrait au plus `n` caractères du flot et les place à l'adresse `ch`. Le nombre de caractères effectivement lus peut être obtenu grâce à la méthode `gcount`.
- `int istream::gcount()` : retourne le nombre de caractères extraits lors de la dernière lecture.
- `istream &istream::flush()` : vide le tampon du flot.

A.4 Les fichiers

Les entrées-sorties sur les fichiers sont également réalisées avec des flots en C++. Ce type d'opérations nécessite l'inclusion de l'en-tête `fstream` en plus de l'en-tête `iostream`. Les deux grandes classes permettant de réaliser ces opérations sont :

– La classe `ofstream` : cette classe est dédiée aux écritures réalisées dans des fichiers. La classe `ofstream` est dérivée de la classe `ostream` et bénéficie donc de toutes les méthodes définies dans cette classe. Voici un exemple d'utilisation :

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

int main()
{
  // Deux modes d'ouverture sont possibles :
  //   - ios::out -> création, fichier écrasé si existant
  //   - ios::app -> ajout en fin de fichier

  ofstream fichierSortie("donnees.txt", ios::out);

  // Test d'ouverture du fichier

  if (!fichierSortie) {
    cerr << "Problème d'ouverture de fichier" << endl;
    exit(1);
  }

  fichierSortie << "J'écris des caractères dans le fichier "
                << "et des nombres : " << 10 << " " << 20
                << endl;

  // Fermeture du fichier

  fichierSortie.close();
}
```

– La classe `ifstream` : cette classe est dédiée aux lectures réalisées dans des fichiers. La classe `ifstream` est dérivée de la classe `istream` et bénéficie donc de toutes le méthodes définies dans cette classe. Voici un exemple d'utilisation :

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

int main()
{
  // Ouverture du fichier

  ifstream fichierEntree("donnees.txt", ios::in);

  // Test d'ouverture du fichier

  if (!fichierEntree) {
    cerr << "Problème d'ouverture de fichier" << endl;
    exit(1);
  }

  char buf[1024];

  // Tant qu'il a y des lignes dans le fichier, on les
  // lit et on les affiche à l'écran

  while (!fichierEntree.eof()) {
    fichierEntree.getline(buf, 1024);
    cout << buf << endl;
  }

  // Fermeture du fichier
```

```
  fichierEntree.close();
}
```

Annexe B

Carte de référence STL

B.1 Les containers

Tous les containers sont dotés des caractéristiques suivantes :
- Certains types prédéfinis (`typedefs`) :
  - `size_type`
  - `pointer`
  - `const_pointer`
  - `reference`
  - `const_reference`
- Des types prédéfinis pour la création d'itérateurs :
  - `iterator`
  - `const_iterator`
  - `reverse_iterator`
  - `const_reverse_iterator`
- `constructor()` : pour la création de containers vides
- `copy-constructor()`
- `destructor()`
- `bool empty() const`
- `size_type size() const` : nombre d'éléments du container
- `size_type max_size() const` : capacité (mémoire occupée) maximum du container
- `container-ref operator=(const-container-ref)` : remplacement de tout le contenu
- `void swap(container-ref)` : inverse tout le contenu
- `bool operator==(const-container-ref)` : teste l'égalité sur tous les éléments
- `bool operator<(const-container-ref)`
- `begin()` et `end()` : méthodes pour accéder au contenu
- `insert()`

Propriétés des containers

Container	Iterateur par déf.	Constructeurs	Accesseurs	Méthodes
array	-	-	op[]	-
vector	random-acc	copy	front(), back(), op[], at()	push_back(), pop_back()
bit_vector	random-acc	copy	front(), back(), op[]	push_front(), pop_back(), flip(), assign()
list	bidirectional	copy	front(), back()	push_front(), push_back(), pop_front(), pop_back, sort(), splice(), remove(), reverse(), unique(), merge()
deque	random-acc	copy	front(), back(), op[], at()	push_front(), push_back(), pop_front(), pop_back()
Associative				
set	bidirectional	copy	find(), lower_bound(), upper_bound(), equal_range()	count()
multiset	bidirectional	copy	find(), lower_bound(), upper_bound(), equal_range()	count()
map	bidirectional	copy	find(), lower_bound(), upper_bound(), equal_range()	count(), op[]
multimap	bidirectional	copy	find(), lower_bound(), upper_bound(), equal_range()	count()
Adaptor				
stack	n/a	copy	top()	push(), pop()
queue	n/a	copy	front(), back()	push(), pop()
priority_queue	n/a	copy	top()	push(), pop()
Special				
bitset	n/a	copy	front(), back(), op[]	push_front(), pop_back(), test(), any(), none(), op&=, op|=, opˆ=, op<<, op>>, set(), reset(), to_ulong(), to_string(), count(), flip()

B.2 Les itérateurs

Propriété des itérateurs

Iterateur	Constructeurs	Accesseurs	Déplacement	Comparaison
all	copy		op++(), op++(int)	
output	"	op*() *write only*	"	
input	"	op*() *read only*	"	op=, =()
forward	"	op*() *read write*	", operator=()	"
bidirectional	"	"	", op–(), op–(int)	"
random-access	"	"	", op+=, op+, op-=, op-, op[]	", op<()
C style pointer	"	"	"	"

Itérateurs spéciaux

istream_iterator	lit un flot d'entrée C++
ostream_iterator	écrit dans un flot de sortie C++
back_insert_iterator	insère à la fin d'un container
front_insert_iterator	insère au début d'un container
insert_iterator	insère dans un container à n'importe quelle position
raw_storage_iterator	itérateur sur mémoire allouée non initialisée
reverse_bidirectional_iterator	itérateur dans le sens inverse

B.3 Les algorithmes

Dans le tableau suivant, les arguments des fonctions et des valeurs de retour sont codés de la manière suivante :

- r random access iterator
- b bidirectional iterator
- f forward iterator
- i input iterator
- o output iterator
- V value
- R reference
- P pair
- B bool
- p unary predicate
- p2 binary predicate
- c compare function
- F unary function
- F2 binary function
- n count
- [...] optional args

Algorithmes en STL

Nom	Retourne	Arguments	Description
Finding			
adjacent_find	i	i,i[,p2]	find sequence of equal elements
binary_search	B	f,f,V[,c]	find a value in a sorted range
count	void	i,i,V,R	count matching elements
count_if	void	i,i,p,R	count elements which satisfy *p*
find	i	i,i,V	locate an equal element
find_if	i	i,i,p	locate an element which satisfies *p*
search	f	f,f,f,f[,p2]	locate a subrange within a range
search	f	f,f,n,V[,p2]	locate a subrange within a range
find_end	f	f,f,f,f[,p2]	find the last subrange which satisfies; like *search* but from the end
lower_bound	f	f,f,V[,c]	returns the *first* possible insert location into a sorted collection
upper_bound	f	f,f,V[,c]	returns the *last* possible insert location into a sorted collection
equal_range	P	f,f,V[,c]	returns the *range of possible insert locations* into a sorted collection
min_element	i	i,i[,c]	find the *smallest*
max_element	i	i,i[,c]	find the *largest*
Applying			
for_each	F	f,f,F	apply a function to a range
transform	o	i,i,o,F or i,i,i,o,F2	apply an operation against a range
replace	v	f,f,V,V	replace all matching elements with a new one
replace_if	v	f,f,p,V	replace all matching elements with a new one
replace_copy	o	i,i,o,V,V	replace during copy, all matching elements with a new one
replace_copy_if	o	i,i,o,p,V	replace during copy, all matching elements with a new one
Filling			
fill	v	f,f,V	fill with a value
fill_n	v	f,n,V	fill with a single value
generate	v	f,f,unary_op	fill with generated values
generate_n	v	f,n,unary_op	fill with generated values
Enumerating			
count	v	i,i,V,R	count the number of matches
count_if	v	i,i,p2,R	count the number of matches, using pred
mismatch	P	i,i,i[,p2]	returns the first subrange than does not match
equal	B	i,i,i[,p2]	*true* if the ranges match
lexicographical_compare	B	i,i,i,[,c]	*true* if the ranges match

Nom	Retourne	Arguments	Description
Copying			
copy	o	i,i,o	copy one range to another
copy_backward	b	b,b,b	reverse copy one range to another
swap_ranges	f	f,f,f	swap one range with another
Ordering			
remove	f	f,f,V	move unwanted entries to the end of the range
remove_if	f	f,f,p	move unwanted entries to the end of the range
remove_copy	o	i,i,o,V	copy and remove unwanted entries
remove_copy_if	o	i,i,o,p	copy and remove unwanted entries
unique	f	f,f[,p2]	collapse the range so that multiple copies of *equal* elements are removed
unique_copy	o	i.i,o[,p2]	copy the range skipping multiple copies of *equal* elements
reverse	v	b,b	reverse the order of a range
reverse_copy	o	b,b,o	reverse the order of a range
rotate	v	f,f,f	rotate a range, given first, middle and last
rotate_copy	o	f,f,f,o	rotate and copy, given first, middle and last
random_shuffle	v	r,r[,rand_gen]	shuffle the order of a range
Sorting			
partition	b	b,b,p	swaps to make all the pred-successes precede the pred-failures
stable_partition	b	b,b,p	swaps to make all the pred-successes precede the pred-failures; preserves relative order
sort	v	r,r[,c]	sorts the elements in the range
stable_sort	v	r,r[,c]	sorts the range; preserve relative order on the "equal" ones
partial_sort	v	r,r,r[,c]	sorts the range into the subrange
partial_sort_copy	r	i,i,r,r[,c]	sorts the range into the subrange at a new location
nth_element	v	r,r,r[,c]	sorts the range so that one specific one is in the right place
next_permutation	B	b,b[,c]	transforms range to next permutation
prev_permutation	B	b,b[,c]	tranforms range to previous permutation

Nom	Retourne	Arguments	Description
Merging			
merge	o	i,i,i,i,o[,c]	merges two input ranges
inplace_merge	v	b,b,b[,c]	merges two input ranges
Set Support			
includes	b	i,i,i,i[,c]	tests for the elementf of one range present in another
set_union	o	i,i,i,i,o[,c]	builds the sorted union
set_intersection	o	i,i,i,i,o[,c]	intersection
set_difference	o	i,i,i,i,o[,c]	difference
set_symmetric_difference	o	i,i,i,i,o[,c]	symmetric_difference
Heap Support			
push_heap	v	r,r[,c]	adds the last element of a range to a heap
pop_heap	v	r,r[,c]	changes a heap into a smaller heap *plus* a last element
make_heap	v	r,r[,c]	changes a range into a heap
sort_heap	v	r,r[,c]	sorts a heap

Bibliographie

[1] Philippe Dosch. « Introduction à la conception objet et à C++ ». Polycopié de cours, IUT Informatique, Université de Nancy 2, January 2001. Un grand merci à Philippe qui m'a autorisé à reprendre certaines parties de son polycopié, entre autres tout ce qui concerne la programmation générique. Son polycopié est plutôt destiné à des personnes connaissant déjà C, mais pas la programmation objet.

[2] B. Eckel. « Thinking in C++ ». `http://mindview.net/Books/TICPP/ThinkingInCPP2e.html`. Excellent ouvrage sur C++, téléchargeable gratuitement sur Internet. Une version française existe également, bien que je n'aie pas eu le temps de juger de la qualité de la traduction.

[3] Nicolai M. Josuttis. *The C++ Standard Library – A Tutorial and Reference.* Addison-Wesley, 1999. Un livre très complet sur la bibliothèque standard de C++.

[4] Stanley B. Lippman and Josée Lajoie. *C++ Primer.* Addison-Wesley, third edition, 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.

[5] Scott Meyers. *More Effective C++ : 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley, 1996. Un complément naturel des autres livres de Meyers.

[6] Scott Meyers. *Effective C++ : 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, 2 edition, 1998. Si vous voulez écrire du C++ de manière professionnelle, ce livre est un *must*.

[7] Scott Meyers. *Effective STL : 50 specific ways to improve your use if the standard template library.* Addison-Wesley, 2001. Un livre à lire absolument pour qui veut utiliser sérieusement la STL dans des applications professionnelles.

[8] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide : C++ Programming with the Standard Template Library.* Addison-Wesley, 1996. Excellent ouvrage de référence sur la STL.

[9] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, 1997. *La* référence de base sur C++, par l'auteur du langage.

[10] Karl Tombre. « Introduction à la programmation ». Cours de tronc commun, 1re année, version 1.3, August 2003. Téléchargeable à l'adresse `http ://www.mines.inpl-nancy.fr/~tombre/polycopies.html`.