Claude Delannoy

Programmer en C++ moderne

De C++11 à C++20

Éditions EYROLLES

Acquérir une parfaite maîtrise du C++ et de la programmation objet

Les versions C++11, C++14 et C++17 ont apporté au langage C++ plus que de nouvelles fonctionnalités : une nouvelle façon de programmer. Dès lors, une refonte complète du classique *Programmer en langage C++* de Claude Delannoy s'imposait. C'est à cette tâche que s'est attelé l'auteur à l'occasion de cette 10e édition de l'ouvrage, en réécrivant les exemples de code et en préconisant de bonnes pratiques de programmation dans l'esprit de ce C++ moderne.

L'ouvrage ainsi remanié commence par une présentation détaillée de la syntaxe de base du langage, s'appuyant dès que possible sur les structures de données de la bibliothèque standard (types *string* et *vector*) et sur la déclaration automatique (C++11). Puis il expose les techniques de gestion dynamique basées sur les « pointeurs intelligents » introduits par C++11 et C++14.

L'auteur insiste ensuite sur la bonne compréhension des concepts objet et de la programmation générique à l'aide des « patrons ». Un chapitre est consacré à la « sémantique de déplacement » introduite par C++11. Plusieurs chapitres sont dédiés aux conteneurs et aux algorithmes de la STL (*Standard Template LIbrary*). Les nouveautés de C++20 (concepts et contraintes, modules, coroutines…) sont présentées en annexe.

Chaque notion nouvelle et chaque fonction du langage est illustrée de programmes complets écrits en C++ moderne, dont le code source est fourni sur le site www.editions-eyrolles.com. Un équivalent en C++03 est proposé quand nécessaire pour les lecteurs amenés à exploiter d'anciens programmes.

À qui s'adresse ce livre ?

- **Aux étudiants de cursus universitaires (DUT, licence, master), ainsi qu'aux élèves des écoles d'ingénieurs.**
- **À tout programmeur ayant déjà une expérience de la programmation (C, C#, Java, Python, PHP…) et souhaitant s'initier au langage C++.**

Au sommaire

Ingénieur informaticien au CNRS, **Claude Delannoy** possède une grande pratique de la formation continue et de l'enseignement supérieur. Réputés pour la qualité de leur démarche pédagogique, ses ouvrages sur les langages et la programmation totalisent plus de 500 000 exemplaires vendus.

www.editions-eyrolles.com
Éditions Eyrolles | Diffusion Geodif

Conception de couverture :
Studio Eyrolles © Éditions Eyrolles

Code éditeur : G67895
ISBN : 978-2-212-67895-2

Programmer en C++ moderne

CHEZ LE MÊME ÉDITEUR

Du même auteur

C. Delannoy. – **Exercices en langage C++** - *178 exercices corrigés*.
N°67663, 4e édition, 2018, 396 pages.

C. Delannoy. – **Programmer en Java** (*couvre Java 9*).
N°67536, 10e édition, 2017, 920 pages.

C. Delannoy. – **Exercices en Java** (*couvre Java 8*).
N°67385, 4e édition, 2017, 346 pages.

C. Delannoy. – **Programmer en langage C** *Cours et exercices corrigés*
N°11825, 5e édition, 2016, 268 pages.

C. Delannoy. – **Le guide complet du langage C**.
N°14012, 2014, 844 pages.

C. Delannoy. – **S'initier à la programmation et à l'orienté objet**
Avec des exemples en C, C++, C#, Python, Java et PHP
N°11826, 2e édition, 2016, 360 pages.

Dans la même collection

A. Tasso. – **Le livre de Java premier langage** *Avec 109 exercices corrigés*.
N°67840, 13e édition, 2019, 600 pages.

H. Bersini, P. Alexis, G. Degols. – **Apprendre la programmation web avec Python et Django**.
N°67515, 2018, 396 pages.

H. Bersini, I. Wellesz. – **La programmation orientée objet**.
Cours et exercices en UML 2 avec Java, C#, C++, Python, PHP et LinQ.
N°67399, 7e édition, 2017, 696 pages.

J. Engels. – **PHP 7**.
N°67360, 2017, 585 pages.

P. Roques. – **UML 2.5 par la pratique**. *Études de cas et exercices corrigés*.
N°67565, 8e édition, 2018, 408 pages.

C. Soutou. – **Programmer avec MySQL**.
N°67379, 5e édition, 2017, 522 pages.

G. Swinnen. – **Apprendre à programmer avec Python 3**.
N°13434, 3e édition, 2012, 435 pages.

Claude Delannoy

Programmer en C++ moderne

De C++11 à C++20

Éditions EYROLLES

Éditions Eyrolles
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

À l'occasion de cette 10e édition, l'ouvrage *Programmer en langage C++* de Claude Delannoy change de titre pour s'appeler désormais *Programmer en C++ moderne*.

ISBN : 978-2-212-67895-2

Avant-propos

Depuis sa conception dans les années 1980 par B. Stroustrup, le C++ a tout naturellement évolué à travers diverses normalisations. Mais, avec la version C++11, le langage a été suffisamment modifié pour qu'on se mette à parler de « C++ moderne ». Le présent ouvrage constitue un remaniement profond des éditions précédentes ; non seulement, il intègre les nouvelles possiblités offertes par les versions C++11, C++14, C++17 et C++20 mais, de surcroît, il présente les nouvelles pratiques de progammation induites par ce changement radical. Les exemples de codes ont tous été réécrits dans ce sens.

1 À qui s'adresse l'ouvrage

Destiné à tous ceux qui souhaitent maîtriser l'ensemble des facettes de la programmation en C++, cet ouvrage s'adresse aussi bien aux étudiants qu'aux développeurs ou aux enseignants en informatique.

Il ne requiert aucune connaissance en programmation orientée objet dont les principes seront présentés en détail au fil de l'étude. Il en va de même pour le langage C dans la mesure où les spécificités de C++ issues du C seront exposées avec le même niveau de détail que les autres.

En revanche, il reste préférable que le lecteur possède déjà une petite expérience de la programmation procédurale (ou structurée), c'est-à-dire dire qu'il soit familiarisé avec les notions de variables, de types, d'affectation, de structures de contrôle, de fonctions, etc., communes à la plupart des langages en usage aujourd'hui (C/C++, Python, JavaScript, Java, PHP...).

2 Structure et contenu de l'ouvrage

L'ouvrage commence par un chapitre très général présentant de façon globale les différentes caractéristiques du langage, montrant ainsi comment y sont intégrés les différents concepts intervenant en programmation procédurale ou en programmation orientée objet (P.O.O.) et en quoi C++ se démarque des autres langages.

La programmation procédurale (chapitres 2 à 11)

Après un chapitre présentant de façon informelle les rudiments les plus indispensables à l'écriture de programmes complets, on aborde l'ensemble des possibilités de programmation procédurale : types de variables, opérateurs et expressions, instructions de contrôle, fonctions (y compris les expressions lambdas).

La déclaration automatique (*auto*) est présentée à ce niveau de façon à être exploitée dans la suite de l'ouvrage lorsqu'elle offre un intérêt.

On s'appuie très tôt sur le type *string* dont on présente les principales propriétés, et ceci alors même que la notion de classe n'a pas encore été exposée.

On trouvera ensuite la présentation des pointeurs natifs et celle de la gestion dynamique dans laquelle on introduit les pointeurs intelligents (en se limitant à ce niveau aux types de base).

Cette partie s'achève sur :

- l'étude en parallèle des tableaux natifs et des principales propriétés des vecteurs (alors même que, là encore, les proprités des patrons et des conteneurs ne seront étudiées en détail qu'ultérieurement) ;
- l'étude succinte des chaînes de style C dont la connaissance reste nécessaire dans certaines situations.

Le chapitre 5 propose une première approche des entrées/sorties conversationnelles, de façon à pouvoir les employer dans un code, avant même que n'aient été étudiées leurs fonctionnalités détaillées.

La programmation orientée objet (chapitres 12 à 25)

Les aspects orientés objet sont ensuite abordés de façon progressive, mais sans pour autant nuire à l'exhaustivité de l'ouvrage. Nous y traitons, non seulement les purs concepts de P.O.O. (classe, constructeur, destructeur, héritage, redéfinition, polymorphisme, programmation générique), mais aussi les aspects très spécifiques au langage (surdéfinition d'opérateurs, fonctions amies). Nous pensons ainsi permettre au lecteur de devenir parfaitement opérationnel dans la conception, le développement et la mise au point de ses propres classes. C'est ainsi, par exemple, que nous avons largement insisté sur le rôle du constructeur de recopie, ainsi que sur la redéfinition de l'opérateur d'affectation, éléments qui conduisent à la notion de « classe canonique ». Toujours dans le même esprit, nous avons pris soin de bien développer les notions avancées mais indispensables que sont la ligature dynamique et les classes

abstraites, lesquelles débouchent sur la notion la plus puissante du langage (et de la P.O.O.) qu'est le polymorphisme.

Tout ce qui est lié à la sémantique de déplacement introduite par C++11 (références à des *rvalue*, construction et affectation par déplacement, références universelles, fonction *forward*...) a été regroupé dans un chapitre spécifique (23), de façon à permettre au lecteur d'en aborder l'étude que lorsqu'il le juge nécessaire.

Cette partie s'achève par deux chapitres (24 et 25) plus techniques consacrés aux flots et à la gestion des exceptions.

Les structures de données et les algorithmes (chapitres 26 à 31)

On étudie ensuite en détail les principaux éléments de la S.T.L. (*Standard Template Library*) après avoir pris soin d'exposer préalablement d'une part les notions de classes et de fonctions génériques (patrons), d'autre part celles de conteneur, d'itérateur et d'algorithmes qui conditionnent la bonne utilisation de la plupart de ses composants.

C'est naturellement là qu'on trouvera l'étude détaillée des classes *string* et *vector* présentées succinctement auparavant.

Les chapitres finaux et les annexes

Les derniers chapitres sont consacrés :

- au préprocesseur ;
- aux énumérations et aux structures de bas niveau héritées du C (champs de bits et unions) ;
- à une introduction aux threads.

Enfin quelques annexes fournissent des récapitulatifs ou des compléments. On y trouvera notamment :

- un récapitulatif des règles de recherche d'une fonction surdéfinie ;
- la liste complète des algorithmes standards.

Le C++ moderne : sa place dans le livre

Les nouvelles fonctionnalités du C++ moderne sont tout naturellement intégrées au fil du texte. Par exemple :

- les déclarations automatiques (*auto*) et la nouvelle syntaxe d'initialisation (entre accolades) sont présentées dans le chapitre 3 pour les types de base (leur intérêt étant alors limité) et elles seront complétées au fur et à mesure de la rencontre de nouveaux types ;
- les pointeurs intelligents sont présentés au chapitre 10 et seront ensuite utilisés à chaque fois qu'on devra recourir à l'allocation dynamique ;

- le type *initializer_list* est présenté au chapitre 7 et on le retrouvera ensuite tout au long de l'ouvrage et l'on verra les problèmes qu'il peut poser avec l'initialisation avec accolades ;
- la boucle *for* pour les séquences est introduite au chapitre 8 pour le type *string* et on la retrouvera ensuite pour les autres formes de séquences ;
- la sémantique de déplacement sera exposée dans son intégralité dans le chapitre 23.

D'une manière générale, le C++ moderne offre des avantages considérables, tant sur le plan de l'uniformisation du langage que sur celui de l'efficacité de la bibliothèque standard ou encore de l'optimisation du code. Il n'en reste pas moins que le respect strict de la compatibilité avec les versions antérieures introduit de nouvelles difficultés liées :

- à la multiplicité des rédactions possibles qui s'offre au programmeur ;
- à quelques incohérences internes ; par exemple, dans de rares cas, *auto* n'est pas utilisable ou ne fournit pas le type escompté.

De plus, l'universalité des déclarations souhaitées par les concepteurs des nouvelles normes n'est pas toujours au rendez-vous ; en particulier, elle est souvent perturbée par l'introduction du type *initializer_list*.

Fidèle à nos habitudes, ces difficultés sont clairement identifiées au fil du texte. Ainsi, le lecteur restera en mesure d'effectuer ses propres choix sur ces situations délicates.

3 Guide d'utilisation

L'ouvrage est conçu sous la forme d'un cours progressif. Celui qui aborde le C++ pour la première fois devra tenter de l'étudier de façon séquentielle. Néanmoins certaines parties peuvent ne pas être utiles à la poursuite de l'étude : elles sont alors soit placées dans une rubrique « informations complémentaires » pour les plus brèves, soit indiquées clairement par une mention en italique figurant au début du paragraphe correspondant.

Chaque notion fondamentale est illustrée d'un programme simple mais complet, accompagné d'un exemple d'exécution. Le code correspondant peut être trouvé sur le site de l'éditeur *www.editions-eyrolles.com*.

Outre son caractère didactique, l'ouvrage est conçu d'une manière très structurée, de façon à en permettre la consultation au delà de la phase d'apprentissage. C'est ainsi qu'il est doté d'une table des matières très détaillée et d'un index fourni. Les exemples complets peuvent servir à une remémoration rapide du concept qu'ils illustrent. Des encadrés permettent de retrouver rapidement la syntaxe d'une instruction ou certaines règles fondamentales.

Bien que fondé sur le C++ moderne, l'ouvrage propose différents outils permettant, le cas échéant, de bien distinguer l'apport des versions récentes :

- des pictogrammes appropriés mentionnent les apports de chacune des versions récentes (C++11, C++14, C++17 et C++20) ;

- lorsque tout un paragraphe est concerné par l'une de ces versions, outre le pictogramme évoqué, le titre en mentionnera le numéro correspondant ;
- les codes, bien qu'écrits en C++ moderne, fournissent systématiquement en commentaires une formulation pour C++98 ; en outre lorsqu'une partie de code s'appuie sur C++14 ou C++17, une formulation C++11 équivalente est proposée.

Ces outils devraient permettre au développeur d'exploiter d'anciens codes. L'enseignant qui le souhaiterait pourra se bâtir un cours simplifié fondé sur un C++ réduit plus proche du C++ historique que du C++ moderne.

Par ailleurs, le programmeur C abordant l'étude du C++ pourra tirer profit des remarques titrées « En C », lesquelles viennent signaler les différences les plus importantes existant entre C et C++.

Enfin, compte tenu de la popularité du langage Java, nous avons introduit de nombreuses remarques titrées « En Java ». Elles mettent l'accent sur les différences majeures existant entre Java et C++. Elles seront utiles non seulement au programmeur Java qui apprend ici le C++, mais également au lecteur qui, après la maîtrise du C++, souhaitera aborder l'étude de Java.

4 Ce que vous trouverez sur le site de l'éditeur

Sur le site *www.editions-eyrolles.com*, outre les codes sources des différents exemples, vous trouverez deux chapitres qui figuraient dans les précédentes éditions :

- la description des principales fonctions héritées du langage C ;
- l'étude des principaux « design patterns » et leur programmation en C++, ce chapitre étant été lui aussi remanié pour le C++ moderne.

Table des matières

7

Les fonctions

Dès qu'un programme dépasse quelques pages de texte, il est pratique de pouvoir le décomposer en des parties relativement indépendantes dont on pourra comprendre facilement le rôle, sans avoir à examiner l'ensemble du code.

La programmation procédurale permet un premier pas dans ce sens, grâce à la notion de fonction que nous allons aborder dans ce chapitre : il s'agit d'un bloc d'instructions qu'on peut utiliser à loisir dans un programme en citant son nom et, éventuellement, en lui fournissant des « arguments » (on dit aussi « paramètres »).

La P.O.O. constituera une seconde étape dans ce processus de décomposition. Chaque classe, définie de façon indépendante, associera des données et des méthodes ; nous verrons que ces méthodes seront rédigées de façon comparable à des fonctions, de sorte que nous serons alors amenés à utiliser l'essentiel de ce que nous aurons étudié ici.

On notera qu'en C++ (comme en C ou en Java), la fonction possède un rôle plus général que la « fonction mathématique ». En effet, une fonction mathématique :

- possède des arguments dont on fournit la valeur lors de l'appel (par exemple, *x* dans *sqrt(x)* ou *5.2* dans *sqrt(5.2)* ;
- fournit un résultat (scalaire) désigné simplement par son appel : *sqrt(x)* désigne le résultat fourni par la fonction ; on peut l'utiliser directement dans une expression arithmétique comme *y + 2*sqrt(x)*.

Or, en C++, si une fonction peut effectivement, comme *sqrt*, jouer le rôle d'une fonction mathématique, elle pourra aussi :

- modifier les valeurs de certains des arguments qu'on lui a transmis ;

- réaliser une action (autre qu'un simple calcul), par exemple : lire des valeurs, afficher des valeurs, ouvrir un fichier, établir une connexion...
- fournir un résultat d'un type non scalaire (structures, objets...) ;
- fournir une valeur qu'on n'utilisera pas ;
- ne pas fournir de valeur du tout.

Nous commencerons par vous présenter la notion de fonction sur un exemple, et nous donnerons quelques règles générales concernant l'écriture des fonctions, leur utilisation et leur déclaration. Nous verrons ensuite que, par défaut, les arguments sont transmis par valeur et nous apprendrons à demander explicitement une transmission par référence. Nous parlerons succinctement des variables globales, surtout pour en déconseiller l'utilisation. Nous ferons ensuite le point sur la classe d'allocation et l'initialisation des variables locales. Nous apprendrons à définir des valeurs par défaut pour certains arguments d'une fonction. Puis nous étudierons l'importante notion de surdéfinition qui permet de définir plusieurs fonctions de même nom, mais ayant des arguments différents. Nous verrons également comment réaliser des fonctions dites « à arguments variables », ce qui nous amènera au passage à parler du type *initializer_list* introduit par C++11. Nous donnerons ensuite quelques éléments concernant les possibilités de compilation séparée de C++. Enfin, nous verrons comment définir des « fonctions en ligne ».

1 Exemple de définition et d'utilisation d'une fonction

Pour vous montrer comment définir et utiliser une fonction en C++, nous commencerons par un exemple simple correspondant en fait à une fonction mathématique, c'est-à-dire recevant des arguments et fournissant une valeur.

```
//ExempleFonction
#include <iostream>
using namespace std ;
float fexple (float, int, int) ; // declaration de fonction fexple
        /***** le programme principal (fonction main) ******/
int main ()
{ float x = 1.5 ;
  int n = 3, p = 5, q = 10 ;
       /* appel de fexple avec les arguments x, n et p */
  float y = fexple (x, n, p) ;       // ou (C++11) : auto y = fexple (x, n, p) ;
  cout << "valeur de y : " << y << endl ;
       /* appel de fexple avec les arguments x+0.5, q et n-1 */
  float z = fexple (x+0.5, q, n-1) ; // ou (C++11) : auto z = fexple (x+0.5, q, n-1) ;
  cout << "valeur de z : " << z << endl ;
}
```

```
        /*************** la fonction fexple ****************/
float fexple (float x, int b, int c)
{ float val ;          // declaration d'une variable "locale" à fexple
  val = x * x + b * x + c ;
  return val ;
}
```

```
valeur de y : 11.75
valeur de z : 26
```

Exemple de définition et d'utilisation d'une fonction

Nous y trouvons tout d'abord, de façon désormais classique, un programme principal formé d'un bloc. Mais, cette fois, à sa suite, apparaît la **définition d'une fonction**. Celle-ci possède une structure voisine de la fonction *main*, à savoir un en-tête et un corps délimité par des accolades ({ et }). Mais l'en-tête est plus élaboré que celui de la fonction *main*. On y trouve le nom de la fonction (*fexple*), une liste d'arguments (nom + type), ainsi que le type de la valeur qui sera fournie par la fonction (on la nomme indifféremment « résultat », « valeur de la fonction », « valeur de retour »...) :

```
float          fexple         (float x,        int b,        int c)
  |              |               |              |             |
type de la    nom de la       premier        deuxième      troisième
"valeur       fonction        argument       argument      argument
de retour"                  (type float)    (type int)    (type int)
```

Les noms des arguments n'ont d'importance qu'au sein du corps de la fonction. Ils servent à décrire le travail que devra effectuer la fonction quand on l'appellera en lui fournissant trois valeurs.

Si on s'intéresse au corps de la fonction, on y rencontre tout d'abord une déclaration :

```
float val ;
```

Celle-ci précise que, pour effectuer son travail, notre fonction a besoin d'une variable de type *float* nommée *val*. On dit que *val* est une variable locale à la fonction *fexple*, de même que les variables telles que *n*, *p*, *y*... sont des variables locales à la fonction *main* (mais comme jusqu'ici nous avions affaire à un programme constitué d'une seule fonction, cette distinction n'était pas utile). Un peu plus loin, nous examinerons plus en détail cette notion de variable locale et celle de portée qui s'y attache.

L'instruction suivante de notre fonction *fexple* est une affectation classique (faisant toutefois intervenir les valeurs des arguments *x*, *n* et *p*).

Enfin, l'instruction *return val* précise la valeur que fournira la fonction à la fin de son travail.

En définitive, on peut dire que *fexple* est une fonction telle que *fexple (x, b, c)* fournit la valeur de l'expression $x^2 + bx + c$. Notez bien l'aspect arbitraire du nom des arguments ; on obtiendrait la même définition de fonction avec, par exemple :

```
float fexple (float z, int coef, int n)
{  float val ; // declaration d'une variable "locale" à fexple
   val = z * z + coef * z + n ;
   return val ;
}
```

Notez qu'avant la fonction *main*, on trouve une déclaration :

```
float fexple (float, int, int) ;
```

Elle sert à prévenir le compilateur que *fexple* est une fonction, et elle lui précise le type de ses arguments ainsi que celui de sa valeur de retour. Nous reviendrons plus loin en détail sur le rôle d'une telle déclaration, ainsi que sur les endroits où elle peut figurer.

Quant à l'utilisation de notre fonction *fexple* au sein de la fonction *main*, elle est classique et comparable à celle d'une fonction prédéfinie telle que *sqrt*. Ici, nous nous sommes contentés d'appeler notre fonction à deux reprises avec des arguments différents.

2 Quelques règles

2.1 Arguments muets et arguments effectifs

Les noms des arguments figurant dans l'en-tête de la fonction se nomment des « arguments muets », ou encore « arguments formels » ou « paramètres formels » (de l'anglais : *formal parameter)*. Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.

Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment des « arguments effectifs » (ou encore « paramètres effectifs »). Comme le laisse deviner l'exemple précédent, on peut utiliser n'importe quelle expression comme argument effectif ; au bout du compte, c'est la valeur de cette expression qui sera transmise à la fonction lors de son appel. Notez qu'une telle « liberté » n'aurait aucun sens dans le cas des paramètres formels : il serait impossible d'écrire un en-tête de *fexple* sous la forme *float fexple (float, a+b, ...)*, pas plus qu'en mathématiques vous ne définiriez une fonction *f* par $f(x+y)=5$!

2.2 L'instruction *return*

Voici quelques règles générales concernant cette instruction.

- L'instruction *return* peut mentionner n'importe quelle expression. Ainsi, nous aurions pu définir la fonction *fexple* précédente de cette manière :

```
float fexple (float x, int b, int c)
{  return (x * x + b * x + c) ;
}
```

- L'instruction *return* peut apparaître à plusieurs reprises dans une fonction, comme dans cet autre exemple :

```
double absom (double u, double v)
{   double s ;
    s = a + b ;
    if (s>0)    return (s) ;
         else   return (-s)
}
```

Notez bien que non seulement l'instruction *return* définit la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée (en l'occurrence, ici, la fonction *main*). Nous verrons qu'une fonction peut ne fournir aucune valeur : elle peut alors disposer de une ou plusieurs instructions *return* **sans expression**, interrompant simplement l'exécution de la fonction ; mais elle peut aussi, dans ce cas, ne comporter aucune instruction *return*, le retour étant alors mis en place automatiquement par le compilateur à la fin de la fonction.

- Si le type de l'expression figurant dans *return* est différent du type du résultat tel qu'il a été déclaré dans l'en-tête, le compilateur mettra automatiquement en place des instructions de conversion : les conversions légales sont celles qui sont autorisées par affectation (avec les mêmes risques de conversions dégradantes, par exemple de *double* en *int*).

  Il est toujours possible de ne pas utiliser le résultat d'une fonction, même si elle en produit un. Bien entendu, cela n'a d'intérêt que si la fonction fait autre chose que calculer un résultat. En revanche, il est interdit d'utiliser la valeur d'une fonction ne fournissant pas de résultat (si certains compilateurs l'acceptent, vous obtiendrez, lors de l'exécution, une valeur aléatoire !).

2.3 Cas des fonctions sans valeur de retour ou sans arguments

Quand une fonction ne renvoie pas de résultat, on le précise, à la fois dans l'en-tête et dans sa déclaration, à l'aide du mot-clé *void*. Par exemple, voici l'en-tête d'une fonction recevant un argument de type *int* et ne fournissant aucune valeur :

```
void sansval (int n)
```

et voici quelle serait sa déclaration :

```
void sansval (int) ;
```

Naturellement, la définition d'une telle fonction ne doit, en principe, contenir aucune instruction *return*. Certains compilateurs ne détecteront toutefois pas l'erreur.

Quand une fonction ne reçoit aucun argument, on se contentera de ne rien mentionner dans la liste d'arguments. Voici l'en-tête d'une fonction ne recevant aucun argument et renvoyant une valeur de type *float* (il pourrait s'agir, par exemple, d'une fonction fournissant un nombre aléatoire !) :

```
float tirage ()
```

Sa déclaration serait très voisine (elle ne diffère que par la présence du point-virgule !) :

```
float tirage () ;
```

Enfin, rien n'empêche de réaliser une fonction ne possédant ni argument ni valeur de retour. Dans ce cas, son en-tête sera de la forme :

```
void message ()
```

et sa déclaration sera :

```
void message () ;
```

Voici un exemple illustrant deux des situations évoquées. Nous y définissons une fonction *affiche_carres* qui affiche les carrés des nombres entiers compris entre deux limites fournies en arguments, et une fonction *erreur* qui se contente d'afficher un message d'erreur (il s'agit de notre premier exemple de programme source contenant plus de deux fonctions).

```
#include <iostream>
using namespace std ;
void affiche_carres (int, int) ; // declaration de affiche_carres
void erreur () ;                 // declaration de erreur
int main ()
{  int debut = 5, fin = 10 ;
      .....
   affiche_carres (debut, fin) ;
      .....
   if (...) erreur () ;
}
void affiche_carres (int d, int f)
{  int i ;
   for (i=d ; i<=f ; i++)
      cout << i << " a pour carre " << i*i << "\n" ;
}
void erreur ()
{  cout << "*** erreur ***\n" ;
}
```

Remarque

En toute rigueur, l'en-tête de la fonction *main* montre l'existence d'une valeur de retour de type *int*. Effectivement, il est possible d'introduire, dans cette fonction *main*, une ou plusieurs instructions *return* accompagnées d'une expression de type *int*. Cette valeur est susceptible d'être utilisée par l'environnement de programmation. Il est convenu que la valeur 0 indique un bon déroulement du programme. Si aucune instruction *return* ne figure dans *main*, tout se passe comme si on trouvait *return 0* à la fin de son exécution.

En C

Le langage C est beaucoup plus tolérant (à tort) que C++ dans les déclarations de fonctions ; on peut omettre le type des arguments (quels que soient leurs types) ou celui de la valeur de retour (s'il s'agit d'un *int*). Mais les règles employées par C++ restent

valides (et même conseillées) en C. Une seule incompatibilité existe dans le cas des fonctions sans argument : C utilise le mot *void*, là où C++ demande une liste vide.

3 Les fonctions et leurs déclarations

3.1 Les différentes façons de déclarer une fonction

Dans notre exemple du paragraphe 1, nous avions fourni la définition de la fonction *fexple* **après** celle de la fonction *main*. Mais nous aurions pu tout aussi bien faire l'inverse :

```
float fexple (float, int, int) ;      // declaration de la fonction fexple
float fexple (float x, int b, int c)
{
    ....
}
int main ()
{      .....
   y = fexple (x, n, p) ;
       .....
}
```

En toute rigueur, dans ce cas, la déclaration de la fonction *fexple* est facultative car, lorsqu'il traduit la fonction *main*, le compilateur connaît déjà la fonction *fexple*. Néanmoins, nous vous déconseillons d'omettre la déclaration de *fexple* dans ce cas (d'ailleurs, bon nombre de compilateurs fourniront un message d'avertissement) ; en effet, il est tout à fait possible qu'ultérieurement vous soyez amené à modifier votre programme source ou même à l'éclater en plusieurs fichiers source comme l'autorisent les possibilités de compilation séparée de C++.

La déclaration d'une fonction porte le nom de **prototype**. Il est possible, dans un prototype, de faire figurer des noms d'arguments, lesquels sont alors totalement arbitraires ; cette possibilité a pour seul intérêt de pouvoir écrire des prototypes qui sont identiques à l'en-tête de la fonction (au point-virgule près), ce qui peut en faciliter la création automatique. Dans notre exemple du paragraphe 1, notre fonction *fexple* aurait pu être **déclarée** ainsi :

```
float fexple (float x, int b, int c) ;
```

3.2 Où placer la déclaration d'une fonction

Dans notre exemple du paragraphe 1, nous avions placé la déclaration de la fonction *fexple* avant la définition de la fonction (*main*) qui y faisait appel. Nous avions donc affaire à une déclaration globale dont la portée s'étendait à l'ensemble du fichier source, ce qui pourrait permettre, le cas échéant, d'utiliser *fexple* dans d'autres fonctions du même fichier. Il s'agit là de la démarche la plus utilisée, notamment dans les programmes conséquents. Mais, on peut théoriquement recourir également à des déclarations locales à une fonction. Ainsi, notre exemple du paragraphe 1 aurait pu utiliser ce canevas :

```
int main()
{ float fexple (float, int, int) ;   // declaration de fexple - portee limitée au main
  .....
}
```

Par ailleurs, nous verrons, au paragraphe 16.1 que, **dans les programmes de quelque importance, les déclarations de fonctions sont en fait placées dans des fichiers en-têtes**, dont l'inclusion se fait alors préférentiellement à un niveau global.

3.3 Contrôles et conversions induites par le prototype

La déclaration d'une fonction peut être utilisée par le compilateur, de deux façons complètement différentes.

1 Si la définition de la fonction se trouve dans le même fichier source (que ce soit avant ou après la déclaration), il s'assure que les arguments muets ont bien le type défini dans le prototype. Dans le cas contraire, il signale une erreur.

2 Lorsqu'il rencontre un appel de la fonction, il met en place d'éventuelles conversions des valeurs des arguments effectifs dans le type indiqué dans le prototype. Par exemple, avec notre fonction *fexple* du paragraphe 1, un appel tel que :

```
fexple (n+1, 2*x, p)
```

sera traduit par :

– l'évaluation de la valeur de l'expression *n+1* (en *int*) et sa conversion en *float* ;

– l'évaluation de la valeur de l'expression *2*x* (en *float*) et sa conversion en *int* (conversion dégradante).

4 Transmission des arguments par valeur

Jusqu'ici, nous nous sommes contentés de dire que les valeurs des arguments étaient transmis à la fonction au moment de son appel. Nous vous proposons de voir ici ce que cela signifie exactement, et les limitations qui en découlent. Nous examinerons tout d'abord le cas général sur un exemple, avant de nous intéresser au cas des arguments (muets et effectifs) constants.

4.1 Cas général

Voyez cet exemple :

```
//TansValeur
#include <iostream>
using namespace std ;
void echange (int a, int b) ;
```

```
int main ()
{ int n=10, p=20 ;
  cout << "-- avant appel   : " << n << " " << p << endl ;
  echange (n, p) ;
  cout << "-- apres appel   : " << n << " " << p << endl ;
}
void echange (int a, int b)
{ int c ;
  cout << "-- debut echange : "<< a << " " <<  b << endl ;
  c = a ;
  a = b ;
  b = c ;
  cout << "-- fin echange   : " << a << " " << b << endl ;
}
```

```
-- avant appel   : 10 20
-- debut echange : 10 20
-- fin echange   : 20 10
-- apres appel   : 10 20
```

Conséquences de la transmission par valeur des arguments

La fonction *echange* reçoit deux valeurs correspondant à ses deux arguments muets *a* et *b*. Elle effectue un échange de ces deux valeurs. Mais, lorsque l'on est revenu dans le programme principal, aucune trace de cet échange ne subsiste sur les arguments effectifs *n* et *p*.

En effet, lors de l'appel de *echange*, il y a eu transmission de la valeur des expressions *n* et *p*. On peut dire que ces valeurs ont été recopiées localement dans la fonction *echange* dans des emplacements nommés *a* et *b*. C'est effectivement sur ces copies qu'a travaillé la fonction *echange*, de sorte que les valeurs des variables *n* et *p* n'ont, quant à elles, pas été modifiées. C'est ce qui explique le résultat constaté.

En fait, ce mode de transmission par valeur n'est que le mode utilisé par défaut par C++. Comme nous allons le voir bientôt, le choix explicite d'une transmission par référence permettra de réaliser correctement notre fonction *echange*.

Remarques

1 C'est bien parce que la transmission des arguments se fait « par valeur » que les arguments effectifs peuvent prendre la forme d'une expression quelconque. Et, d'ailleurs, nous verrons qu'avec la transmission par référence, les arguments effectifs ne pourront plus être des expressions, mais simplement des *lvalue*.

2 La norme n'impose aucun ordre pour l'évaluation des différents arguments d'une fonction lors de son appel. En général, ceci est de peu d'importance, excepté dans une situation (fortement déconseillée !) telle que :

```
int i = 10 ;
   ...
f (i++, i) ;  // i++ peut se trouver caclulé avant i - l'appel sera : f (10, 11)
              //                          ou après i - l'appel sera : f (10, 10)
```

3 En toute rigueur, la valeur de retour (lorsqu'elle existe) est elle aussi transmise par valeur, c'est-à-dire qu'elle fait l'objet d'une recopie de la fonction appelée dans la fonction appelante. Ce point peut sembler anodin, mais nous verrons plus tard qu'il existe des circonstances où il s'avère fondamental et où, là encore, il faudra recourir à une transmission par référence.

4.2 Transmission par valeur et constance des arguments

Nous avons déjà vu la signification de *const* (et *constexpr* depuis C++11) dans le cas des déclarations de variables, lesquelles deviennent alors des *lvalue* non modifiables. Ici, nous examinons :

- les conséquences de ce type de qualificatif *const* ou *constexpr* lorsqu'il concerne un argument effectif d'une fonction ;
- comment il est également possible d'appliquer le qualificatif *const* à des arguments muets et quelle en est alors la signification.

4.2.1 Cas des arguments effectifs constants

Si l'on considère la fonction *f* définie ainsi :

```
void f (int n) { ..... }
```

on peut l'appeler avec la valeur de n'importe quelle expression, y compris d'une expression constante puisque la valeur en sera recopiée :

```
const int  p = 4 ;
f (p) ;    // OK
constexpr int NB = 12 ;
f(NB) ;    // OK
```

4.2.2 Cas des arguments muets constants

L'utilisation de *const* sur des aruments muets a une toute autre signification. Il précise que la *lvalue* corrrespondante ne doit pas être modifiée dans le corps de la fonction. Dans le cas contraire, on obtient une erreur de compilation :

```
void f (const int n)
{ .....
  n++ ;    // rejeté en compilation
}
```

Notez que l'utilisation de *constexpr* dans cette situation n'a aucune signification.

5 Transmission des arguments par référence

Nous venons de voir que, par défaut, les arguments d'une fonction sont transmis par valeur. Comme nous l'avons constaté avec la fonction *echange*, ce mode de transmission ne permet pas à une fonction de modifier la valeur d'un argument. Or, C++ dispose de la notion de **référence**, laquelle correspond à celle d'adresse : considérer la référence d'une variable revient à considérer son adresse, et non plus sa valeur. Ici, nous allons voir comment utiliser cette notion de référence pour la transmission d'arguments ; au paragraphe 8, nous verrons comment l'utiliser également pour une valeur de retour. Enfin, au paragraphe 14. nous verrons comment cette notion de référence possède en fait un caractère plus général mais peu usité.

5.1 Exemple de transmission d'argument par référence

Le programme ci-dessous montre comment utiliser une transmission par référence dans notre précédente fonction *echange* :

```
// TransRef
#include <iostream>
using namespace std ;
void echange (int &, int &) ;
int main ()
{  int n=10, p=20 ;
   cout << "-- avant appel :   " << n << " " << p << endl ;
   echange (n, p) ;        // attention, ici pas de &n, &p
   cout << "-- apres appel :   " << n << " " << p << endl ;
}
void echange (int & a, int & b)
{  int c ;
   cout << "-- debut echange : " << a << " " << b << endl ;
   c = a ; a = b ; b = c ;
   cout << "-- fin echange   : " << a << " " << b << endl ;
}
```

```
-- avant appel :   10 20
-- debut echange : 10 20
-- fin echange   : 20 10
-- apres appel   : 20 10
```

Utilisation de la transmission d'argument par référence en C++

Dans l'instruction :

```
void echange (int & a, int & b) ;
```

la notation *int & a* signifie que a est une information de type *int* transmise par référence. Notez bien que, dans la fonction *echange*, on utilise simplement le symbole *a* pour désigner cette variable dont la fonction aura reçu effectivement l'adresse.

En Java

La notion de référence existe en Java, mais elle est entièrement transparente au programmeur. Plus précisément, les variables d'un type de base sont transmises par valeur, tandis que les objets sont transmis par référence. Il reste cependant possible de créer explicitement une copie d'un objet en utilisant une méthode appropriée dite de *clonage*.

5.2 Propriétés de la transmission par référence d'un argument

La transmission par référence d'un argument évite une recopie d'information, d'où un gain de temps d'exécution qui, s'il n'est guère sensible dans le cas de variables scalaires, pourra devenir apréciable dans le cas de gros agrégats ou de gros objets. En revanche, il entraîne un certain nombre de conséquences qui n'existaient pas dans le cas de la transmission par valeur.

Ainsi, dans l'appel d'une fonction recevant un argument par référence, l'argument effectif correspondant doit obligatoirement être une *lvalue* modifiable (une expression n'a pas d'adresse, une constante n'est pas modifiable). En outre, comme il s'agit de transmettre l'adresse de cette *lvalue*, il n'est pas possible d'en prévoir une quelconque conversion (même non dégradante), puisqu'il faudrait pour cela créer une nouvelle valeur, à une nouvelle adresse :

```
void f (int &) ;   // f reçoit la référence à un entier
  .....
const int n=15 ;
int q ;
f(q) ;       // OK
f(2*q+3) ;   // erreur : 2*q+3 n'est pas une lvalue modifiable
f(3) ;       // erreur : 3 n'est pas modifiable (c'est une rvalue)
f(n) ;       // erreur : n est une lvalue non modifiable
  .....
float x ;
f(x) ;       // erreur : x n'est pas de type int
```

La transmission par référence impose donc à un argument effectif d'être une lvalue modifiable du type prévu pour l'argument muet. Il existe cependant une exception dans le cas des arguments muets constants comme nous allons le voir maintenant.

5.3 Référence à un argument muet constant

En toute rigueur, la transmission par référence, telle que nous venons de l'examiner, possède deux propriétés distinctes :

- elle évite la recopie d'informations ;
- elle permet à une fonction de travailler directement sur l'argument effectif transmis et, donc, d'en modifier la valeur.

C++ vous permet de profiter de la première propriété, tout en interdisant la modification de l'argument effectif. Il suffit pour cela de prévoir constant l'argument muet correspondant comme dans cet en-tête :

```
void f (const int & n) ;
```

Dans ce cas, la fonction *f* s'attend à recevoir l'adresse d'une constante et elle ne devra pas, dans sa définition, modifier la valeur de *n* ; dans le cas contraire, on obtiendra une erreur de compilation.

Cette fois, les appels suivants seront corrects :

```
const int c = 15 ;
   .....
f(3) ;    // correct ici
f(c) ;    // correct ici
```

L'acceptation de ces instructions se justifie par le fait que *f* a prévu de recevoir une référence à quelque chose de constant ; le risque de modification évoqué précédemment n'existe donc plus. Notez d'ailleurs que, dans l'appel *f(3)*, 3 n'est même plus une *lvalue* non modifiable, mais une *rvalue.*

A priori, C++ aurait pu se limiter à cela. Mais, en fait, **on pourra également appeler *f* en lui fournissant en argument une expression quelconque** *exp*, de type *int* ou même d'un type compatible avec *int*. Pour ce faire, il est alors prévu la création d'une variable temporaire qui recevra la valeur de l'expression (ou de sa conversion en *int*) et dont l'adresse sera fournie à *f*. Par exemple :

```
void f (const int &) ;
int n ;
f (3*n + 1) ;    // f reçoit la référence à une variable temporaire
                 //   contenant la valeur de l'expression 3*n+1
float x ;
f (x) ;          // correct : f reçoit la référence à une variable temporaire
                 //  contenant le résultat de la conversion de x en int
f (3*x + 5.2) ;  // correct : f reçoit la référence à une variable temporaire
                 //  contenant le résultat de la conversion en int de la valeur
                 //  de l'expression 3*x +5.2
```

En définitive, l'utilisation de *const* pour un argument muet transmis par référence est lourde de conséquences. Certes, comme on s'y attend, cela amène le compilateur à verifier la constance de l'argument concerné au sein de la fonction. Mais, de surcroît, on autorise la création d'une copie de l'argument effectif (précédée d'une conversion) dès lors que ce dernier n'est plus une *lvalue* ou une constante du type attendu.

Cette remarque prendra encore plus d'acuité dans le cas où l'argument en question sera un objet volumineux.

5.4 Induction de risques indirects

Le choix du mode de transmission par référence est fait au moment de l'écriture de la fonction concernée. L'utilisateur de la fonction n'a plus à s'en soucier ensuite, si ce n'est au

niveau de la déclaration du prototype de la fonction (d'ailleurs, ce prototype proviendra en général d'un fichier en-tête).

En contrepartie, l'emploi de la transmission par référence accroît les risques d'« effets de bord » non désirés. En effet, lorsqu'il appelle une fonction, l'utilisateur ne sait plus s'il transmet, au bout du compte, la valeur ou l'adresse d'un argument (la même notation pouvant désigner l'une ou l'autre des deux possibilités). Il risque donc de modifier une variable dont il pensait n'avoir transmis qu'une copie de la valeur.

Remarque

Nous verrons (paragraphe 6 du chapitre 9) qu'il est également possible de « simuler » une transmission par référence, par le biais de pointeurs. Dans ce cas, l'utilisateur de la fonction devra transmettre explicitement des adresses : les risques évoqués précédemment disparaissent, en contrepartie d'une programmation plus délicate et plus risquée de la fonction elle-même.

6 Les variables globales

Nous avons vu comment échanger des informations entre différentes fonctions grâce à la transmission d'arguments et à la récupération d'une valeur de retour.

En théorie, en C++, plusieurs fonctions (dont, bien entendu le programme principal *main*) peuvent partager des variables communes qu'on qualifie alors de **globales**. Il s'agit cependant là d'une **pratique risquée qu'il faudra éviter au maximum**. Nous vous la présenterons cependant ici car :

- vous risquez de rencontrer du code y recourant ;
- la notion de variable globale permet de mieux comprendre la différence entre classe d'allocation statique et classe d'allocation dynamique, laquelle prendra toute son importance dans un contexte objet ;
- dans une classe, les champs de données auront un comportement « global » pour les (seules) méthodes de cette classe.

6.1 Exemple d'utilisation de variables globales

Voyez l'exemple de programme ci-après :

```
// VarGlobales
#include <iostream>
using namespace std ;
int i ;                    // declaration d'une variable globale
void optimist (void) ;
```

```
int main ()
{   for (i=1 ; i<=5 ; i++) optimist() ;
}
void optimist(void)
{   cout << "il fait beau " << i << " fois\n" ;
}
```

```
il fait beau 1 fois
il fait beau 2 fois
il fait beau 3 fois
il fait beau 4 fois
il fait beau 5 fois
```

Exemple d'utilisation de variable globale

La variable *i* a été déclarée en dehors de la fonction *main*. Elle est alors connue de toutes les fonctions qui seront compilées par la suite au sein du même programme source. Ainsi, ici, le programme principal affecte à *i* des valeurs qui se trouvent utilisées par la fonction *optimist*.

Notez qu'ici la fonction *optmist* se contente d'utiliser la valeur de *i* mais rien ne l'empêche de la modifier. C'est précisément ce genre de remarque qui doit vous inciter à n'utiliser les variables globales que dans des cas limités. En effet, toute variable globale peut être modifiée insidieusement par n'importe quelle fonction. Lorsqu'on souhaite qu'une fonction modifie la valeur d'une variable, il est beaucoup plus judicieux d'en transmettre l'adresse en argument (soit par référence, comme nous avons appris à le faire, soit par pointeur, comme on le verra plus tard). Dans ce cas, l'appel de la fonction indique clairement quelles sont les seules variables susceptibles d'être modifiées.

6.2 La portée des variables globales

Les variables globales ne sont connues du compilateur que dans la partie du programme source suivant leur déclaration. On dit que leur **portée** (ou encore leur **espace de validité**) est limitée à la partie du programme source qui suit leur déclaration (pour l'instant, nous nous limitons au cas où l'ensemble du programme est compilé en une seule fois).

Ainsi, voyez, par exemple, ces instructions :

```
int main ()
{ .... }
int n ;
float x ;
void fct1 (...)
{ .... }
void fct2 (...)
{ .... }
```

Les variables *n* et *x* sont accessibles aux fonctions *fct1* et *fct2*, mais pas au programme principal. En pratique, bien qu'il soit possible effectivement de déclarer des variables globales à n'importe quel endroit du programme source qui soit extérieur aux fonctions, on procédera rarement ainsi. En pratique, s'il faut absolument recourir à des variables globales (par exem-

ple, dans des codes critiques en temps d'exécution), on s'arrangera pour privilégier la lisibilité des codes en regroupant en début de programme[1] les déclarations de toutes ces variables globales.

6.3 La classe d'allocation des variables globales

D'une manière générale, les variables globales existent pendant toute l'exécution du programme dans lequel elles apparaissent. Leurs emplacements en mémoire sont parfaitement définis lors de l'édition de liens. On traduit cela en disant qu'elles font partie de la **classe d'allocation statique**.

De plus, ces variables se voient **initialisées à zéro**[2], avant le début de l'exécution du programme, sauf, bien sûr, si vous leur attribuez explicitement une valeur initiale au moment de leur déclaration.

7 Les variables locales

À l'exception de l'exemple du paragraphe précédent, les variables que nous avions rencontrées jusqu'ici n'étaient pas des variables globales. Plus précisément, elles étaient définies au sein d'une fonction (qui pouvait être *main*). De telles variables sont dites **locales** à la fonction dans laquelle elles sont déclarées.

7.1 La portée des variables locales

Les variables locales ne sont connues du compilateur qu'à l'intérieur de la fonction où elles sont déclarées. **Leur portée est donc limitée à cette fonction.** Les variables locales n'ont aucun lien avec des variables globales de même nom ou avec d'autres variables locales à d'autres fonctions. Voyez cet exemple :

```
int n ;
int main ()
{  int p ;
   ....
}
void fct1 ()
{  int p ;
   int n ;
}
```

La variable *p* de *main* n'a aucun rapport avec la variable *p* de *fct1*. De même, la variable *n* de *fct1* n'a aucun rapport avec la variable globale *n*. En toute rigueur, si l'on souhaite utiliser

1. Ou dans un fichier en-tête séparé.

2. Cette notion de « zéro » sera précisée pour les pointeurs et pour les types structurés (tableaux, vecteurs, objets...).

dans *fct1* la variable globale *n*, on utilise l'opérateur dit « de résolution de portée » (::) en la nommant *::n*.

7.2 Les variables locales automatiques

Par défaut, les variables locales ont une durée de vie limitée à celle d'**une exécution** de la fonction dans laquelle elles figurent.

Plus précisément, leurs emplacements ne sont pas définis de manière permanente comme ceux des variables globales. Un nouvel espace mémoire leur est alloué à chaque entrée dans la fonction et libéré à chaque sortie. Il sera donc généralement différent d'un appel au suivant.

On traduit cela en disant que la **classe d'allocation** de ces variables est **automatique**. Nous aurons l'occasion de revenir plus en détail sur ce mode de gestion de la mémoire. Pour l'instant, il est important de noter que la conséquence immédiate de ce mode d'allocation est que les valeurs des variables locales ne sont pas conservées d'un appel au suivant (on dit aussi qu'elles ne sont pas « rémanentes »). Nous reviendrons un peu plus loin (paragraphe 9) sur les éventuelles initialisations de telles variables.

D'autre part, les valeurs reçues en arguments par une fonction sont traitées de la même manière que les variables locales. Leur durée de vie correspond également à celle de la fonction.

Informations complémentaires

Généralement, on utilise pour les variables automatiques une « pile » de type FIFO (*First In, First Out*) simulée dans une zone de mémoire. Lors de l'appel d'une fonction, on alloue de l'espace sur la pile à la fois pour la valeur de retour, les valeurs des arguments ou leurs références et pour les différentes variables locales à la fonction.

Lors de la sortie de la fonction, ces différents emplacements sont libérés par la fonction elle-même, hormis celui de la valeur de retour qui sera libéré par la fonction appelante, après qu'elle l'aura utilisé.

La gestion de la pile se fait à l'aide d'un pointeur désignant le premier emplacement disponible. La libération d'un emplacement se fait par une simple modification de la valeur de ce pointeur ; l'emplacement libéré garde généralement sa valeur, de sorte que si, par une erreur de programmation, on y accède avant qu'il ait été alloué à une autre variable, on peut croire, à tort, qu'une variable locale est « rémanente »...

On notera que dans bon nombre d'environnements de programmation, l'espace mémoire alloué à la pile souffre de limitations plus importantes que celui alloué au tas.

7.3 Les variables locales statiques

Il est possible de demander d'attribuer un emplacement permanent à une variable locale et de conserver ainsi sa valeur d'un appel au suivant. Il suffit pour cela de la déclarer à l'aide du mot-clé **static**. En voici un exemple :

```
// VarLocStatiques
#include <iostream>
using namespace std ;
void fct() ;
int main ()
{  for ( int n=1 ; n<=5 ; n++)
      fct() ;
}
void fct()
{  static int i ;
   i++ ;
   cout << "appel numero : " << i << endl ;
}
```

```
appel numero : 1
appel numero : 2
appel numero : 3
appel numero : 4
appel numero : 5
```

Exemple d'utilisation de variable locale statique

La variable locale *i* a été déclarée de classe « statique ». On constate bien que sa valeur progresse de *1* à chaque appel. De plus, on note qu'au premier appel sa valeur est nulle. En effet, comme pour les variables globales (lesquelles sont aussi de classe statique) : **les variables locales de classe statique sont, par défaut, initialisées à zéro**. Notez que nous aurions pu initialiser explicitement *i*, par exemple :

```
static int i = 3 ;
```

Dans ce cas, nos appels auraient porté des numéros allant de 4 à 8.

Prenez garde à ne pas confondre une variable locale de classe statique avec une variable globale. En effet, la portée d'une telle variable reste toujours limitée à la fonction dans laquelle elle est définie. Ainsi, dans notre exemple, nous pourrions définir une variable globale nommée *i* qui n'aurait alors aucun rapport avec la variable *i* de *fct*.

Remarque

Si *i* n'avait pas été déclarée avec l'attribut *static*, il se serait agit d'une variable locale usuelle, non rémanente et, de surcroît, non initialisée. Sa valeur aurait donc été aléatoire. De plus, ici, c'est toujours le même emplacement qui se serait trouvé alloué à *i* sur la pile,

de sorte qu'on afficherait bien une séquence de valeurs (dont la première est aléatoire), donnant l'illusion d'une certaine rémanence de *i*.

7.4 Variables locales à un bloc

Comme nous l'avions déjà évoqué succinctement, C++ vous permet de déclarer des variables locales à un bloc. Leur portée est alors tout naturellement limitée à ce bloc ; leur emplacement est alloué à l'entrée dans le bloc et il disparaît à la sortie. Il est permis qu'une variable locale porte le même nom qu'une variable locale d'un bloc englobant.

```
void f()
{ int n ;         // n est accessible de tout le bloc constituant f
    .....
  for (...)
  { int p ;      // p n'est connue que dans le bloc de for
    int n ;      // n masque la variable n de portée "englobante"
      .....      // attention, on ne peut pas utiliser ::n ici qui
      .....      // désignerait une variable globale (inexistante ici)
  }
    .....
  { int p ;      // p n'est connue que dans ce bloc ; elle est allouée ici
    .....        // et n'a aucun rapport avec la variable p ci-dessus
  }              // et elle sera désallouée ici
    .....
}
```

Notez qu'on peut créer artificiellement un bloc, indépendamment d'une quelconque instruction structurée comme *if*, *for*. C'est le cas du deuxième bloc interne à notre fonction *f* ci-dessus.

D'autre part, nous avons déjà vu qu'il était possible d'effectuer une déclaration dans une instruction *for*, par exemple :

```
for (int i=2, j=4 ; ... ; ...)
{ // i et j sont considérées comme deux variables locales à ce bloc
}
```

On notera que, même si l'instruction *for* ne contient aucun bloc explicite, comme dans :

```
for (int i=1, j=1 ; i<4 ; i++) cout << i+j ;
```

les variables *i* et *j* ne seront plus connues par la suite, exactement comme si l'on avait écrit :

```
for (int i=1, j=1 ; i<4 ; i++) { cout << i+j ; }
```

Informations complémentaires

En toute rigueur, il existe une classe d'allocation un peu particulière, à savoir la classe « registre » : toute variable entrant a priori dans la classe automatique peut être déclarée explicitement (jusqu'à C++17) avec le qualificatif *register*. Celui-ci demande au compilateur d'utiliser, dans la mesure du possible, un « registre » de la machine pour y ranger la variable : cela peut amener quelques gains de temps d'exécution. Bien entendu, cette

possibilité ne peut s'appliquer qu'aux variables d'un type simple. Cette fonctionnalité disparaît en C++17 mais *register* reste un mot-clé.

7.5 Le cas des fonctions récursives

C++ autorise la récursivité des appels de fonctions. Celle-ci peut prendre deux aspects :

- récursivité directe : une fonction comporte, dans sa définition, au moins un appel à elle-même ;
- récursivité croisée : l'appel d'une fonction entraîne celui d'une autre fonction qui, à son tour, appelle la fonction initiale (le cycle pouvant d'ailleurs faire intervenir plus de deux fonctions).

Voici un exemple fort classique (d'ailleurs inefficace sur le plan du temps d'exécution) d'une fonction calculant une factorielle de manière récursive :

```
long fac (int n)
{   if (n>1) return (fac(n-1)*n) ;
       else return(1) ;
}
```

Fonction récursive de calcul de factioriuelle

Il faut bien voir que chaque appel de *fac* entraîne une allocation d'espace pour les variables locales et pour son argument *n* (apparemment, *fct* ne comporte aucune variable locale ; en réalité, il lui faut prévoir un emplacement destiné à recevoir sa valeur de retour). Or chaque nouvel appel de *fac*, à l'intérieur de *fac*, provoque une telle allocation, sans que les emplacements précédents soient libérés.

Il y a donc un empilement des espaces alloués aux variables locales, parallèlement à un empilement des appels de la fonction. Ce n'est que lors de l'exécution de la première instruction *return* que l'on commencera à « dépiler » les appels et les emplacements et donc à libérer de l'espace mémoire.

8 Transmission par référence d'une valeur de retour

N.B. *L'étude de ce paragraphe peut être différée jusqu'à celle du chapitre sur la surdéfinition des opérateurs.*

8.1 Introduction

Le mécanisme que nous avons exposé pour la transmission des arguments par référence s'applique à la valeur de retour d'une fonction. Considérons :

```
int & f ()
{ .....
  return n ;   // on suppose ici n de type int
}
```

Un appel de *f* provoquera la transmission en retour non plus d'une valeur, mais de la référence (adresse) de *n*. Là encore, on évite une recopie, ce qui entraîne un gain de temps d'exécution qui deviendra intéressant avec de gros objets. Cependant, cette fois, on voit que ***n* ne peut pas être une variable locale à *f*** car, sinon, elle serait détruite à la sortie de *f* et l'on récupérerait dans la fonction appelante, une adresse à quelque chose qui n'existe plus[3]. En revanche, on pourra ainsi fournir en valeur de retour la référence à un objet créé dynamiquement comme nous apprendrons à le faire plus tard. On pourra également renvoyer une référence qu'on aura reçue en argument.

8.2 Conséquences dans la définition de la fonction

Puisqu'il s'agit d'une transmission d'adresse, la valeur de retour mentionnée dans l'instruction *return* ne peut être qu'une *lvalue* modifiable. Il ne pourra donc pas s'agir d'une expression et toute conversion s'avère impossible (même s'il s'agit d'une conversion non dégradante). La *lvalue* figurant dans l'instruction *return* doit donc être du type exact prévu dans l'en-tête.

Le cas de la valeur de retour constante fera cependant exception à ces règles comme nous le verrons au paragraphe 8.5.

8.3 Conséquences dans l'utilisation de la fonction

Dès lors qu'une fonction renvoie une référence, il devient possible d'utiliser son appel comme une *lvalue* modifiable. Voyez cet exemple :

```
int & f (int q) { ..... }
int n, p;
float x ;
   .....
f(p) = 2 * n + 5 ;  // à la référence fournie par f(p), on range la valeur
                    // de l'expression 2*n+5, de type int
```

3. Cette erreur s'apparente à celle due à la transmission en valeur de retour d'un pointeur sur une variable locale (situation que nous rencontrerons plus tard). Elle est encore plus difficile à détecter dans la mesure où le seul moment où l'on peut utiliser la référence concernée est l'appel lui-même (alors qu'un pointeur peut être utilisé à volonté...). Dans un environnement ne modifiant pas la valeur d'une variable lors de sa « destruction », aucune erreur ne se manifeste ; ce n'est que lors du portage dans un environnement ayant un comportement différent que les choses deviennent catastrophiques.

```
f(2*p+1) = x ;      // à la référence fournie par f(2*p+1), on range la valeur
                    // de x, apres conversion en int
```

On voit qu'on se trouve ainsi en présence d'expressions d'une forme inhabituelle, même en mathématiques.

Cette propriété sera largement exploitée lorsqu'il s'agira de surdéfinir un opérateur (en fait, une fonction) dont la nature imposera de fournir une *lvalue* en résultat. Ce sera précisément le cas de l'opérateur []4.

Nous verrons cependant au paragraphe 8.5 qu'il ne sera plus possible d'utiliser l'appel de la fonction comme une *lvalue* dès lors que la valeur de retour sera une référence à une constante.

En revanche, contrairement à ce qui se produisait pour les arguments transmis par référence, aucune contrainte d'exactitude de type ne pèse sur l'utilisation d'une valeur de retour fournie par référence, car il reste toujours possible de la soumettre à une conversion avant de l'utiliser :

```
inf & f () ;
float x ;
   .....
x = f() ;   // OK : on convertira en int la valeur située à la référence
            //  reçue en retour de f
```

8.4 Exemple

Voici un exemple, un peu artificiel, d'une fonction recevant deux références d'entiers et renvoyant l'une d'entre elles. Un indicateur booléen statique permet de choisir une fois la première, une fois la seconde.

```
// ValRetourRef
#include <iostream>
using namespace std ;
int & alterne (int &, int &) ;
int main ()
{ int n=1, p=3, q=5 ;
  alterne (n, p ) = 0 ; // on affecte 0 a la lvalue dont alterne fournit la ref
  cout << "n = " << n << "  p = " << p << endl ;
  alterne (p, q) = 12 ;
  cout << "p = " << p << "  q = " << q << endl ;
}
int & alterne (int & n1, int & n2)
{ static bool indic = true ;
  if (indic) { indic = false ; return n1 ; }
        else { indic = true ;  return n2 ; }
}
```

4. Toutefois, on emploiera alors pour l'appel une notation de la forme *t[i] = n*, laquelle semblera alors plus naturelle, malgré son équivalence avec la notation fonctionnelle utilisée ici.

```
n = 0  p = 3
p = 3  q = 12
```

Exemple de transmission d'une valeur de retour par référence

8.5 Valeur de retour constante

Il est possible de prévoir qu'une fonction fournisse par référence une valeur de retour constante, comme dans :

```
const int & f2(.....)
```

Dans ce cas, dans la définition de la fonction, l'instruction *return* pourra toujours mentionner une *lvalue* modifiable comme auparavant, mais cette fois, il deviendra également possible d'y faire figurer une *rvalue* (constante ou expression). La norme a en effet prévu qu'on renvoie alors la référence d'une **copie temporaire de cette expression**. Qui plus est, puisqu'on crée une copie, une éventuelle conversion redevient possible (pour peu qu'elle soit légale par affectation). Ainsi, avec une fonction *f2* définie de cette manière :

```
const int & f2 (.....)  // cette fois on renvoie la référence à un entier constant
{ int n ; float x ;
  .....
}
```

voici quelques exemples d'instructions *return* légales :

```
return 5 ;   // OK : on renvoie la référence à une copie temporaire de 5
return n+5 ; // OK : on renvoie la référence à une copie temporaire
             //       de la valeur de l'expression n+5
return n ;   // A eviter : on renvoie la référence à une variable locale
             //     (en général, avertissement à la compilation)
return x ;   // OK : on renvoie la référence à un int temporaire
             // obtenu par conversion de la valeur de x
```

En revanche, dans l'appel de la fonction, la valeur de retour (référence à une constante temporaire) ne pourra plus être utilisée comme une *lvalue* ni faire l'objet de conversion (comme c'était le cas avec une référence non constante) :

```
const int & f2 () ;
int n ;
float x ;
   .....
f2() = 2 * n + 5 ;  // erreur : f() n'est pas une lvalue
f2() = x ;          // idem
```

9 Initialisation des variables

Nous avons vu qu'il était possible d'initialiser explicitement une variable lors de sa déclaration. Ici, nous allons faire le point sur ces possibilités, lesquelles dépendent en fait de la classe d'allocation de la variable concernée.

9.1 Les variables de classe statique

Il s'agit des variables globales, ainsi que des variables locales déclarées avec l'attribut *static*. Ces variables sont permanentes. Elles sont initialisées une seule fois avant le début de l'exécution du programme. Elles peuvent être initialisées explicitement lors de leur déclaration, à l'aide de constantes ou d'**expressions constantes** (calculables par le compilateur) d'un **type compatible par affectation** avec celui de la variable. On notera bien que les conversions dégradantes sont acceptées avec l'initialisation utilisant le signe = (seule forme existant en C++98), mais peu conseillées. Avec C++11, on peut utiliser la forme avec accolades qui interdit les conversions dégradantes. Voici quelques exemples concernant ici des variables déclarées statiques dans une fonction *f* (les mêmes considérations s'appliqueraient à des variables globales) :

```
void f ()
{ const int NB = 5 ;
  static int limit = 2 * NB + 1 ;  // 2*NB+1 est une expression constante
  static short CTOT1 = 25 ;        // 25 de type int est converti en short int
  static short CTOT2 = 50000 ;     // 50000 (de type int) est converti en short int
                                   //    avec risque de résultat erroné
  // static short CTOT3 {50000} ;  // C++11 : rejeté à la compilation
                                   //   si 50000 non représentable en short int
                                   //   (donc dépendant de l'implémentation)
                                   // - mêmes remarques avec CTOT3 = {50000}
  static float XMAX1 = 5 ;         // 5 de type int est converti en float
  static float XMAX2 { 5 } ;       // 5 de type int est converti en float
                                   // - même chose avec XMAX2 = { 5 }
  static long YTOT1 = 9.7 ;        // 9.7 de type float est converti en long
                                   //   avec résultat tronqué (9)
  // static long YTOT2 { 9.7 } ;   // C++11 : rejeté à la compilation
                                   // - même chose avec YTOT2 = { 9.7 }
}
```

En l'absence d'initialisation explicite, ces variables seront initialisées à zéro[5].

9.2 Les variables de classe automatique

Il s'agit des variables locales à une fonction ou à un bloc. Ces variables ne sont **pas initialisées par défaut**. En revanche, comme les variables de classe statique, elles peuvent être initialisées explicitement lors de leur déclaration. Dans ce cas, la valeur initiale peut être fournie sous la forme d'une **expression quelconque (d'un type compatible par affectation)**, pour peu que sa valeur soit définie au moment de l'entrée dans la fonction correspondante (il peut s'agir de la fonction *main* !). N'oubliez pas que ces variables automatiques se trouvent alors initialisées à chaque appel de la fonction dans laquelle elles sont définies. En voici un cas d'école :

5. Rappelons que la notion de zéro est relative au type concerné. Avec un type simple, il s'agit bien de la valeur nulle mais avec un pointeur, il s'agira de la valeur *nullptr* tandis qu'avec le type *string*, il s'agira d'une chaîne vide...

```
// ClasseAutomatique
#include <iostream>
using namespace std ;
int n ;              // variable globale
void fct (int r) ;
int main ()
{ for (int p=1 ; p<=5 ; p++) { n = 2*p ; fct(p) ; }
}
void fct(int r)
{ int q=n, s=r*n ;   // l'initialisation utilise la globale n et l'argument r
  cout << r << " " << q << " " << s << endl ;
}
```

```
1 2 2
2 4 8
3 6 18
4 8 32
5 10 50
```

Initialisation de variables de classe automatique

10 Les arguments par défaut

10.1 Exemples

Jusqu'ici, nos appels de fonction renfermaient autant d'arguments que la fonction en attendait effectivement. C++ permet de s'affranchir en partie de cette règle, grâce à un mécanisme d'attribution de valeurs par défaut à des arguments non fournis lors de l'appel.

Exemple 1

Considérez l'exemple suivant :

```
// ArgDefaut1
#include <iostream>
using namespace std ;
void fct (int, int=12) ; // prototype avec une valeur par defaut
int main ()
{  int n=10, p=20 ;
   fct (n, p) ;             // appel "normal"
   fct (n) ;                // appel avec un seul argument
}
void fct (int a, int b)     // en-tete "habituel"
{  cout << "premier argument : " << a << endl ;
   cout << "second argument  : " << b << endl ;
}
```

```
premier argument : 10
second argument  : 20
premier argument : 10
second argument  : 12
```

Exemple de définition de valeur par défaut pour un argument

La déclaration de *fct*, ici dans la fonction *main*, est réalisée par le prototype :

```
void fct (int, int = 12) ;
```

La déclaration du second argument apparaît sous la forme :

```
int = 12
```

Celle-ci précise au compilateur que, en cas d'absence de ce second argument dans un éventuel appel de *fct*, il lui faudra « faire comme si » l'appel avait été effectué avec cette valeur.

Les deux appels de *fct* illustrent le phénomène. Notez qu'un appel tel que :

```
fct ( )
```

serait rejeté à la compilation puisque ici il n'était pas prévu de valeur par défaut pour le premier argument de *fct*.

Exemple 2

Voici un second exemple, dans lequel nous avons prévu des valeurs par défaut pour tous les arguments de *fct* :

```
// ArgDefaut2
#include <iostream>
using namespace std ;
void fct (int=0, int=12) ; // prototype avec deux valeurs par defaut
int main ()
{  int n=10, p=20 ;
   fct (n, p) ;               // appel "normal"
   fct (n) ;                  // appel avec un seul argument
   fct () ;                   // appel sans argument
}
void fct (int a, int b)     // en-tete "habituel"
{  cout << "premier argument : " << a << endl ;
   cout << "second argument  : " << b << endl ;
}
```

```
premier argument : 10
second argument  : 20
premier argument : 10
second argument  : 12
premier argument : 0
second argument  : 12
```

Exemple de définition de valeurs par défaut pour plusieurs arguments

10.2 Les propriétés des arguments par défaut

Lorsqu'une déclaration prévoit des valeurs par défaut, les arguments concernés doivent obligatoirement être les derniers de la liste.

Par exemple, une déclaration telle que :

```
float fexple (int = 5, long, int = 3) ;
```

est interdite. En fait, une telle interdiction relève du pur bon sens. En effet, si cette déclaration était acceptée, l'appel suivant :

```
fexple (10, 20) ;
```

pourrait être interprété aussi bien comme :

```
fexple (5, 10, 20) ;
```

que comme :

```
fexple (10, 20, 3) ;
```

Notez bien que le mécanisme proposé par C++ revient à **fixer les valeurs par défaut dans la déclaration de la fonction et non dans sa définition**. Autrement dit, ce n'est pas le « concepteur » de la fonction qui décide des valeurs par défaut, mais l'utilisateur. Une conséquence immédiate de cette particularité est que les arguments soumis à ce mécanisme et les valeurs correspondantes peuvent varier d'une utilisation à une autre ; en pratique toutefois, ce point ne sera guère exploité, ne serait-ce que parce que les déclarations de fonctions sont en général « figées » une fois pour toutes, dans un fichier en-tête.

Nous verrons que les arguments par défaut se révéleront particulièrement précieux lorsqu'il s'agira d'écrire ce que l'on nomme le « constructeur d'une classe ».

Remarque

Les valeurs par défaut ne sont pas nécessairement des expressions constantes. Elles ne peuvent toutefois pas faire intervenir de variables locales[6].

En Java

Les arguments par défaut n'existent pas en Java.

11 Surdéfinition de fonctions

D'une manière générale, on parle de « surdéfinition »[7] lorsqu'un même symbole possède plusieurs significations différentes, le choix de l'une des significations se faisant en fonction du contexte. C'est ainsi que la plupart des langages évolués utilisent la surdéfinition d'un certain nombre d'opérateurs. Par exemple, dans une expression telle que :

6. Ni la valeur *this* pour les fonctions membres (*this* sera étudié au chapitre 13).

7. De *overloading*, parfois traduit par « surcharge ».

```
a + b
```

la signification du + dépend du type des opérandes *a* et *b* ; suivant les cas, il pourra s'agir d'une addition d'entiers ou d'une addition de flottants. De même, le symbole * peut désigner, suivant le contexte, une multiplication d'entiers, de flottants (ou, commme nous le verrons lorsque nous étudierons les pointeurs, une indirection).

Un des grands atouts de C++ est de permettre la surdéfinition de la plupart des opérateurs (lorsqu'ils sont associés à la notion de classe). Lorsque nous étudierons cet aspect, nous verrons qu'il repose en fait sur la surdéfinition de fonctions. C'est cette dernière possibilité que nous proposons d'étudier ici pour elle-même.

Pour pouvoir employer plusieurs fonctions de même nom, il faut bien sûr un critère (autre que le nom) permettant de choisir la bonne fonction. En C++, ce choix est basé (comme pour les opérateurs cités précédemment en exemple) sur le type des arguments. Nous commencerons par vous présenter un exemple complet montrant comment mettre en œuvre la surdéfinition de fonctions. Nous examinerons ensuite différentes situations d'appel d'une fonction surdéfinie avant d'étudier les règles détaillées qui président au choix de la « bonne fonction ».

11.1 Mise en œuvre de la surdéfinition de fonctions

Nous allons définir et utiliser deux fonctions nommées *sosie*. La première possédera un argument de type *int*, la seconde un argument de type *double*, ce qui les différencie bien l'une de l'autre. Pour que l'exécution du programme montre clairement la fonction effectivement appelée, nous introduisons dans chacune une instruction d'affichage appropriée. Dans le programme d'essai, nous nous contentons d'appeler successivement la fonction surdéfinie *sosie*, une première fois avec un argument de type *int*, une seconde fois avec un argument de type *double*.

```
// Surdefinition
#include <iostream>
using namespace std ;
void sosie (int) ;              // les prototypes
void sosie (double) ;
int main ()                     // le programme de test
{  int n=5 ;
   double x=2.5 ;
   sosie (n) ;
   sosie (x) ;
}
void sosie (int a)              // la premiere fonction
{  cout << "sosie numero I   a = " << a << endl ;
}
void sosie (double a)           // la deuxieme fonction
{   cout << "sosie numero II  a = " << a << endl ;
}
```

```
sosie numero I   a = 5
sosie numero II  a = 2.5
```

Exemple de surdéfinition de la fonction sosie

Vous constatez que le compilateur a bien mis en place l'appel de la « bonne fonction » *sosie*, au vu de la liste d'arguments (ici réduite à un seul).

11.2 Exemples de choix d'une fonction surdéfinie

Notre précédent exemple était simple, dans la mesure où nous appelions toujours la fonction *sosie* avec un argument ayant **exactement** l'un des types prévus dans les prototypes (*int* ou *double*). On peut se demander ce qui se produirait si nous l'appelions par exemple avec un argument de type *char* ou *long*, ou si l'on avait affaire à des fonctions comportant plusieurs arguments...

Avant de présenter les règles de détermination d'une fonction surdéfinie, examinons tout d'abord quelques situations assez intuitives.

Exemple 1

```
void sosie (int) ;       // sosie I
void sosie (double) ;    // sosie II
char c ; float y ;
   .....
sosie(c) ;   // appelle sosie I, après conversion de c en int
sosie(y) ;   // appelle sosie II, après conversion de y en double
sosie('d') ;  // appelle sosie I, après conversion de 'd' en int
```

Exemple 2

```
void essai (int, double) ;    // essai I
void essai (double, int) ;    // essai II
int n, p ; double z ; char c ;
   .....
essai(n,z) ; // appelle essai I
essai(c,z) ; // appelle essai I, après conversion de c en int
essai(n,p) ; // erreur de compilation,
```

Compte tenu de son ambiguïté, le dernier appel conduit à une erreur de compilation. En effet, deux possibilités existent ici : convertir *p* en *double* sans modifier *n* et appeler *essai I* ou, au contraire, convertir *n* en *double* sans modifier *p* et appeler *essai II*.

Exemple 3

```
void test (int n=0, double x=0) ;     // test I
void test (double y=0, int p=0) ;     // test II
int n ; double z ;
   .....
test(n,z) ; // appelle test I
```

```
test(z,n) ; // appelle test II
test(n) ;   // appelle test I
test(z) ;   // appelle test II
test() ;    // erreur de compilation, compte tenu de l'ambiguïté.
```

Exemple 4

Avec ces déclarations :

```
void truc (int) ;         // truc I
void truc (const int) ;   // truc II
```

vous obtiendrez une erreur de compilation. En effet, C++ n'a pas prévu de distinguer *int* de *const int*. Cela se justifie par le fait que, les deux fonctions *truc* recevant une copie de l'information à traiter, il n'y a aucun risque de modifier la valeur originale. Notez bien qu'ici l'erreur tient à la seule présence des déclarations de *truc*, indépendamment d'un appel quelconque.

Exemple 5

En revanche, considérez maintenant ces déclarations :

```
void chose (int &) ;        // chose I
void chose (const int &) ;  // chose II
int n = 3 ;
const int p = 5 ;       // ou constexpr depuis C++11
   .....
chose (n) ;  // appelle chose I
chose (p) ;  // appelle chose II
```

Cette fois, la distinction entre *int &* et *const int &* est justifiée. En effet, on peut très bien imaginer que *chose I* modifie la valeur de la *lvalue* dont elle reçoit la référence, tandis que *chose II* n'en fait rien.

Exemple 6

L'exemple précédent a montré comment on pouvait distinguer deux fonctions agissant, l'une sur une référence, l'autre sur une référence à une constante. Mais l'utilisation de références possède des conséquences plus subtiles, comme le montrent ces exemples (revoyez éventuellement le paragraphe 5.3) :

```
void chose (int &) ;       // chose I
void chose (const int &)   // chose II
int n :
float x ;
   .....
chose (n) ; // appelle chose I
chose (2) ; // appelle chose II, après copie éventuelle de 2 dans un entier[8]
            // temporaire dont la référence sera transmise à chose
chose (x) ; // appelle chose II, apres conversion de la valeur de x en un
            // entier temporaire dont la référence sera transmise à chose
```

8. Comme l'autorise la norme, l'implémentation est libre de faire ou non une copie dans ce cas.

```
chose (2*n+1) ; // appelle chose II, en lui transmettant la référence de l'empla-
                //  cement temporaire contenant la valeur de l'expression 2*n+1
```

Remarque

Comme le mode de transmission (référence ou valeur) n'intervient pas dans le choix d'une fonction surdéfinie, on pourrait penser que ces déclarations sont refusées :

```
void chose (int &) ; // chose I -  argugment de type int
void chose (int) ;   // chose II - argument d'un type compatible avec int
```

En réalité, les choses sont plus subtiles que cela puisque le rejet n'a lieu qu'en cas d'appel ambigu :

```
int n ; float x ;
chose (x) ;  // OK : seule chose II convient, après conversion de x en int
chose (n) ;  // NON - rejeté car chose I et chose II conviennent
```

Les choses sont encore plus étonnantes avec ces déclarations :

```
void chose (int &) ;       // chose I -   argugment de type int
void chose (const int &) ; // chose II -  argugment de type compatible avec int
void chose (int) ;         // chose III - argument d'un type compatible avec int
```

Elles sont en effet acceptées mais **tout appel de *chose*** (avec un *int*, une constante ou une expression compatible avec *int*) **sera rejeté** compte tenu de son ambigüité.

En pratique, on se limitera aux surdéfinitions des exemples 5 et 6 (référence et référence à une constante).

En Java

La surdéfinition des fonctions existe en Java. Mais les règles de recherche de la bonne fonction sont beaucoup plus simples qu'en C++, car il existe peu de possibilités de conversions implicites.

11.3 Règles de recherche d'une fonction surdéfinie

Pour l'instant, nous vous présenterons plutôt la philosophie générale, ce qui sera suffisant pour l'étude des chapitres suivants. Au cours de cet ouvrage, nous serons amenés à vous apporter des informations complémentaires. De plus, l'ensemble de toutes ces règles sont reprises en Annexe A.

11.3.1 Cas des fonctions à un argument

Le compilateur recherche la « meilleure correspondance » possible. Bien entendu, pour pouvoir définir ce qu'est cette meilleure correspondance, il faut qu'il dispose d'un critère d'évaluation. Pour ce faire, il est prévu différents niveaux de correspondance :

1) ***Correspondance exacte*** : on distingue bien les uns des autres les différents types de base, en tenant compte de leur éventuel attribut de signe[9] ; de plus, comme on l'a vu dans les

exemples précédents, l'attribut *const* peut intervenir dans le cas de références (il en ira de même pour les pointeurs).

2) ***Correspondance avec promotions numériques***, c'est-à-dire essentiellement :

   char et *short* –> *int*

   float –> *double*

   Rappelons qu'un argument transmis par référence ne peut être soumis à aucune conversion, sauf lorsqu'il s'agit de la référence à une constante.

3) ***Conversions dites standards*** : il s'agit des conversions légales en C++, c'est-à-dire de celles qui peuvent être imposées par une affectation (sans opérateur de *cast*) ; cette fois, il peut s'agir de conversions dégradantes puisque, notamment, toute conversion d'un type numérique en un autre type numérique est acceptée.

4) ***D'autres niveaux*** sont prévus ; en particulier on pourra faire intervenir ce que l'on nomme des « conversions définies par l'utilisateur » (C.D.U.), qui ne seront étudiées qu'au chapitre 17.

   Là encore, un argument transmis par référence ne pourra être soumis à aucune conversion, sauf s'il s'agit d'une référence à une constante.

La recherche s'arrête au premier niveau ayant permis de trouver une correspondance, qui doit alors être unique. Si plusieurs fonctions conviennent au même niveau de correspondance, il y a erreur de compilation due à l'ambiguïté rencontrée. Bien entendu, si aucune fonction ne convient à aucun niveau, il y a aussi erreur de compilation.

11.3.2 Cas des fonctions à plusieurs arguments

L'idée générale est qu'il doit se dégager une fonction « meilleure » que toutes les autres. Pour ce faire, le compilateur sélectionne, **pour chaque argument**, la ou les fonctions qui réalisent la meilleure correspondance (au sens de la hiérarchie définie ci-dessus). Parmi l'ensemble des fonctions ainsi sélectionnées, il choisit celle (si elle existe et si elle est unique) qui réalise, pour chaque argument, une correspondance au moins égale à celle de toutes les autres fonctions[10].

Si plusieurs fonctions conviennent, là encore, on aura une erreur de compilation due à l'ambiguïté rencontrée. De même, si aucune fonction ne convient, il y aura erreur de compilation.

Notez que les fonctions comportant un ou plusieurs arguments par défaut sont traitées comme si plusieurs fonctions différentes avaient été définies avec un nombre croissant d'arguments.

---

9. Attention : en C++, *char* est différent de *signed char* et de *unsigned char*.

10. Ce qui revient à dire qu'il considère l'intersection des ensembles constitués des fonctions réalisant la meilleure correspondance pour chacun des arguments.

12 Les fonctions et la déclaration *auto (C++11)*

12.1 Déclaration automatique du type des valeurs de retour

Depuis C++11, on peut effectuer une déclaration automatique du type de la valeur de retour, afin que son type soit déduit automatiquement du type de l'expression fournie à l'instruction *return*, comme dans cet exemple :

```
auto f(int n) { .....
                return n * n ;    // le type renvoyé par f sera int
              }
```

Ici, le type de la valeur de retour sera déduit par le compilateur comme étant celui de l'expression *n * n*, c'est-à-dire *int*.

Avec cette autre définition :

```
auto g(int n) { .....
                return 5.25 * n ;    // le type renvoyé par f sera double
              }
```

il serait déduit comme étant *double.*

On notera bien que *auto* doit alors obligatoirement être utilisé à la fois dans l'en-tête et dans le prototype de la fonction concernée. Ici, il faudra par exemple déclarer *f* de cette manière :

```
auto f(int n) ;    // OK
```

et non :

```
int f (int n) ;    // NON
```

La démarche s'applique même lorsqu'il y a plusieurs instructions *return*, à condition que les expressions mentionnées soient du même type ; dans l'exemple suivant, le type déduit pour la valeur de retour sera *double* :

```
auto h(int n) { if(n%2==0) return 2*n+1 ;   // OK : les deux expressions 2*n+1
                       else return 2*n ;    // et 2*n sont de même type
              }
auto h (int)  { if(n%2==0) return 2*n+1 ;   // NON : les deux expressions 2*n+1
                       else return 2.5*n ;  // et 2.5*n ne sont pas de même type
              }
```

12.2 Déclaration automatique du type des arguments muets

Toujours depuis C++11, on peut effectuer une déclaration automatique du type d'un argument, lequel sera alors déduit du type de l'argument effectif fourni lors de l'appel. Voyez cet exemple :

```
void f(auto n) { ..... }
int p = 5 ; double x = 5.2 ;
f(p) ;      // l'argument muet n sera du type de p : int
f(x) ;      // l'argument muet x sera du type de x : double
f(2.5*p) ;  // l'argument muet n sera du type de l'expression 2.5*p : double
```

12.3 Combinaison des deux possibilités

On peut exploiter ces deux possibilités de déclaration automatique (valeur de retour et argument muet) pour réaliser une fonction partiellement « générique », c'est-à-dire applicable à différents types non prévus dans sa définition, comme dans cet exemple :

```
// AutoFonctionsGeneriques
#include <iostream>
using namespace std ;
auto f(auto n) { return n+5 ; }
int main()
{ int n = 3 ;
  cout << "f(3) = " << f(n) << endl ;
  double x = 2.6 ;
  cout << "f(2.6) = " << f(x) << endl ;
  char c = 'a' ;
  cout << "f('a') = " << f(c) << endl ;
}
```

```
f(3) = 8
f(2.6) = 7.6
f('a') = f
```

Exemple de fonction générique

Notez que la généricité est partielle, car *f* ne peut être appelée que pour des types disposant de l'opération d'addition.

13 Les fonctions déclarées *constexpr* (C++11)

Nous avons déjà vu comment, depuis C++11, on peut distinguer les expressions constantes (qui ne varient pas au sein de la fonction concernée) des expressions calculables à la compilation déclarées avec *constexpr*.

Cette notion de possibilité de calcul à la compilation va se généraliser aux fonctions définies par l'utilisateur.

Plus précisément, une f**onction déclarée *constexpr*** est une fonction dont le corps ne comporte qu'une ligne de la forme *return x* où *x*, après substitution des valeurs des arguments, est une expression constante, comme dans cet exemple :

```
constexpr int carre (int n)
{ return n * n ;
}
```

Voici un exemple d'utilisation de *carre* :

```
const int n = 6 ;                          // expression constante, déjà en C++98
constexpr int quantite = 12 + carre (n) ;  // expression constante avec C++11
```

Mais une fonction déclarée *constexpr* peut aussi être appelée avec des arguments non connus à la compilation. Elle se comporte alors comme une fonction ordinaire ; en voici un exemple utilisant la fonction *constexpr carre* précédente :

```
int p ; cin >> p ;
int q1 = carre (p) ;   // OK : p n'est pas constexpr ; la fonction carre
                       // sera utilisée classiquement lors de l'exécution
constexpr q2 = carre (p) ;  // sera rejeté à la compilation
                            // car q2 n'est pas une expression constante
```

À partir de C++14, les fonctions déclarées *constexpr* peuvent contenir des variables locales et plusieurs instructions exécutables (affectation, *for*, *if*, *switch*, *while*, *do... while*). En voici un exemple :

```
#include <iostream>
using namespace std ;
constexpr f (int a, int b)
{ int c = 0 ;
  if (a>0) c = a * b ;
      else c = a + b ;
   return c  ;
}
int main()
{ int n ;
  cout << "Donnez un entier : " ; cin >> n ;
  constexpr int val = f(3, 8) ;
  cout << "Resultat : " << val << endl ;
}
```

```
Donnez un entier : 4
Resultat : 24
```

Exemple de définition d'une fonction constexpr *(depuis C++14)*

14 La référence d'une manière générale

N.B. *Ce paragraphe peut être ignoré dans un premier temps.*

L'essentiel concernant la notion de référence réside dans la transmission d'arguments ou de valeur de retour. Cependant, en toute rigueur, la notion de référence peut intervenir en dehors de ces contextes comme nous allons le voir ici.

14.1 Déclaration de variables de type référence

D'une manière générale, il est possible de déclarer un identificateur comme référence d'une autre variable. Considérez, par exemple, ces instructions :

```
int n ;
int & p = n ;
```

La seconde signifie que *p* est une référence à la variable *n*. Ainsi, dans la suite, *n* et *p* désigneront le même emplacement mémoire. Par exemple, avec :

```
n = 3 ;
cout << p ;
```

nous obtiendrons la valeur 3.

Remarque

Il ne sera pas possible de définir des pointeurs sur des références, ni des tableaux de références.

14.2 Initialisation de référence

La déclaration :

```
int & p = n ;
```

est en fait une déclaration de référence (ici *p*) accompagnée d'une initialisation (à la référence de *n*). D'une façon générale, il n'est pas possible de déclarer une référence sans l'initialiser, comme dans :

```
int & p ;     // incorrect, car pas d'initialisation
```

Notez bien qu'une fois déclarée (et initialisée), une référence ne peut plus être modifiée. D'ailleurs, aucun mécanisme n'est prévu à cet effet : si, ayant déclaré *int & p=n ;* vous écrivez *p=q*, il s'agit obligatoirement de l'affectation de la valeur de *q* à l'emplacement de référence *p*, et non de la modification de la référence *q*.

On ne peut pas initialiser une référence avec une constante. La déclaration suivante est incorrecte :

```
int & n = 3 ;        // incorrecte
```

Cela est logique puisque, si cette instruction était acceptée, elle reviendrait à initialiser *n* avec une référence à la valeur (constante) 3. Dans ces conditions, l'instruction suivante conduirait à modifier la valeur de la constante 3 :

```
n = 5 ;
```

En revanche, il est possible de définir des références à des constantes qui peuvent alors être initialisées par des constantes. Ainsi la déclaration suivante est-elle correcte :

```
const int & n = 3 ;
```

Elle génère une variable temporaire (ayant une durée de vie imposée par l'emplacement de la déclaration) contenant la valeur 3 et place sa référence dans *n*. On peut dire que tout se passe comme si vous aviez écrit :

```
int temp = 3 ;
int & n = temp ;
```

avec cette différence que, dans le premier cas, vous n'avez pas explicitement accès à la variable temporaire.

Enfin, les déclarations suivantes sont encore correctes :

```
float x ;
const int & n = x ;
```

Elles conduisent à la création d'une variable temporaire contenant le résultat de la conversion de *x* en *int* et placent sa référence dans *n*. Ici encore, tout se passe comme si vous aviez écrit ceci (sans toutefois pouvoir accéder à la variable temporaire *temp*) :

```
float x ; int temp = x ;
const int & n = temp ;
```

Remarques

1 En toute rigueur, l'appel d'une fonction conduit à une « initialisation » des arguments muets. Dans le cas d'une référence, ce sont donc les règles que nous venons de décrire qui sont utilisées. Il en va de même pour une valeur de retour. On retrouve ainsi le comportement décrit aux paragraphes 5.2 et 7.5.

2 Au chapitre 23, nous verrons que C++11 a également introduit la notion de référence à une *rvalue* (variable temporaire).

15 Les fonctions à arguments variables

Grâce à la surdéfinition, nous savons réaliser des fonctions recevant des arguments de types et de nombre différents. Cependant, les possibilités d'appel en sont alors limitées par le jeu des fonctions prévues et les éventuelles conversions implicites susceptibles d'être mises en oeuvre. Ici, nous allons voir comment une même fonction peut disposer d'arguments quelconques en nombre et en type.

Avant C++11, on pouvait réaliser une fonction recevant un nombre quelconque d'arguments, éventuellement de types différents (mais qui devaient pouvoir être logiquement déduits lors de l'écriture de la fonction). On recourait pour cela à des fonctions spéciales (*va_start* et *va_arg*) dont l'usage est dorénavant déconseillé mais que nous présenterons toutefois à la fin de ce paragraphe pour vous permettre d'exploiter d'anciens codes. Depuis C++11, on peut recourir à un nouveau type *initializer_list* qui permet de manipuler une liste d'un nombre quelconque de valeurs d'un même type (ou, plus généralement, d'un type compatible avec un type donné). Nous allons l'étudier ici. Plus tard (paragraphe 7 du chapitre 18), nous verrons qu'il existe des patrons de fonctions dits « variadiques » autorisant cette fois des arguments de types différents, en nombre variable.

15.1 Le type *initializer_list* (C++11)

C++11 définit donc un type dit « générique »[11], noté *initializer_list*<*T*>, *T* étant un type quelconque. Une variable d'un tel type est formée d'une liste de valeurs **non modifiables** d'un type implicitement compatible avec *T*.

11. Plus tard, nous verrons qu'il s'agit en fait d'un patron de classes.

Par exemple, avec ces instructions :

```
int n = 10 ;
initializer_list<int> li = { 1, 3, 5, n, n+5 } ;
```

on crée une liste de 5 valeurs formée des valeurs entières 1, 3, 5, 10 et 15.

Nous verrons plus tard que ce nouveau type s'apparente aux « conteneurs » et qu'il dispose à ce titre d'un mécanisme d'itération permettant d'en parcourir les valeurs. Pour l'instant, nous nous contenterons d'utiliser la forme généralisée de l'instruction *for*, indépendante de la notion de conteneur, applicable aux séquences de valeurs (nous la retrouverons pour les vecteurs, les chaînes ou les tableaux). Elle se présente ainsi :

```
for (int i : li)     // ou encore for (auto i : li)
{  // utilisation de i, copie d'une valeur de li
}
```

Dans la boucle, *i* représente successivement une **copie** de chacune des valeurs de la liste *li*. Si on modifie *i*, cela n'affecte aucune valeur de la liste *li*.

Voici un exemple de programme utilisant ces fonctionnalités :

```
// Initialiszer_list
#include <iostream>
using namespace std;
int main()
{  int n = 10 ; float x = 5.25 ;
   initializer_list<int> li = {1, 3, 5, n, n+5, 'a'} ;  // 'a' est converti en int
   initializer_list<double> ld = {1, 3, x, x*x } ;      // 1 et 3 en double
   cout << "liste li = " ;
   for (int i : li) cout << i << " " ;
   cout << endl ;
   cout << "liste ld = " ;
   for (double d : ld) cout << d << " " ;
   cout << endl ;
   //  for (int & i :li) n++ ;    interdit ici - voir remarque
}
```

```
liste li = 1 3 5 10 15 97
liste ld = 1 3 5.25 27.5625
```

Utilisation du type initializer_list

Remarques

1 Dans le chapitre consacré au type *string*, nous verrons que la boucle *for* généralisée peut aussi travailler directement avec les références des valeurs concernées et non, comme ici, sur une copie. Cependant, cette possibilité ne s'applique pas au type *initializer_list* car ses valeurs ne sont pas modifiables.

2 En théorie, la déclaration *auto* s'applique à des listes d'initialisation. Néanmoins, on conseille dans ce cas d'utiliser alors l'initialisation copie comme dans :

```
auto l1 = {10, 20} ;
```

En effet, la forme suivante est acceptée en C++11 et rejetée en C++14 :

```
auto l2 {10, 20} ;
```

Pour une liste à un seul élément, il est indispensable d'utiliser l'initialisation copie si l'on veut obtenir une liste et non une variable d'un type simple :

```
auto l3 = {10} ;   // OK
auto l4  {10} ;    // ici, l4 serait un int et non une initializer_list<int>
```

Il va de soi que *auto* ne convient pas pour une liste vide.

15.2 Application à une fonction à nombre variable d'arguments (C++11)

On peut donc utiliser ce type *initializer_list* pour transmettre à une fonction un nombre variable d'arguments, soit de même type, soit d'un type implicitement compatible avec un type donné. Ainsi, une fonction d'en-tête :

```
void f (initializer_list<float> val)       // val est une liste de float
```

pourra être appelée avec une liste de valeur d'un type compatible avec *float* (*char*, *int*, *long*, *float, double, ...*).

Voyez cet exemple dans lequel nous utilisons la fonction *size* pour obtenir le nombre d'éléments de la liste *val* (notez qu'on utilise la notation *val.size()* et non *size(val)* car, comme nous le verrons plus tard, nous avons affaire à une fonction membre d'une classe).

```
// FonctionsArgVar1
#include <iostream>
using namespace std;
void f (initializer_list<float> val)       // val est une liste de float
{ cout << val.size() << " Valeurs : " ;
  for (float v : val) cout << v << "  " ;
  cout << endl ;
}
int main()
{ f({1.5, 2.3}) ;              // liste de 2 valeurs de type float
  f({}) ;   // liste vide (attention f {} serait considéré comme f sans arguments)
  f ({1.25, 5, 9, 3.4, 7 }) ;  // liste de 5 valeurs convertibles en float
}
```

```
2 Valeurs : 1.5  2.3
0 Valeurs :
5 Valeurs : 1.25  5  9  3.4  7
```

Utilisation d'une liste d'initialisation en argument d'une fonction

On notera qu'on peut utiliser une liste vide, à condition de bien la noter *({})* et non simplement *{}* (qui serait interprété comme absence d'arguments) ou encore mélanger plusieurs types dans un initialiseur, pour peu qu'ils soient compatibles par affectation avec le type prévu.

15.3 Les anciennes fonctions *va_start* et *va_arg*

Comme nous l'avons dit en introduction, avant C++11, la seule façon de réaliser une fonction à arguments variables consistait à utiliser les fonctions particulières *va_start* et *va_arg* (dont le prototype figure dans le fichier en-tête *cstdarg*). La seule contrainte à respecter était que la fonction devait posséder au moins un argument fixe (c'est-à-dire toujours présent). En effet, comme nous allons le voir, c'était le dernier argument fixe qui permettait, en quelque sorte, d'initialiser le parcours de la liste d'arguments.

Voici un exemple dans lequel les deux premiers arguments sont fixes, l'un étant de type *int*, l'autre de type *char*. Les arguments suivants, de type *int*, sont en nombre quelconque et l'on a supposé que le dernier d'entre eux était *-1*. Cette dernière valeur sert donc, en quelque sorte, de « sentinelle ». Par souci de simplification, nous nous sommes contentés, dans cette fonction, de lister les valeurs de ces différents arguments (fixes ou variables), à l'exception du dernier.

```
// FonctArgVar2
#include <iostream>
#include <cstdarg>       // pour va_arg et va_list
using namespace std ;
void essai (int, char, ...) ;
int main ()
{  cout << "++ Premier essai\n" ;
   essai (125, 'a', 15, 30, 40, -1) ;
   cout << "++ Deuxieme essai\n" ;
   essai (6264, 'S', -1) ;
}
void essai (int par1, char  par2, ...)
{  va_list adpar ;
   int parv ;
   cout << " - premier parametre : " << par1 << endl ;
   cout << " - second parametre  : " << par2 << endl ;
   va_start (adpar, par2) ;
   while ( (parv = va_arg (adpar, int) ) != -1)
       cout << " - argument variable : " << parv << endl ;
}
```

```
++ Premier essai
 - premier parametre : 125
 - second parametre  : a
 - argument variable : 15
 - argument variable : 30
```

```
 - argument variable : 40
++ Deuxieme essai
 - premier parametre : 6264
 - second parametre  : S*/
```

Arguments en nombre variable, délimités par une sentinelle

Vous constatez la présence, dans l'en-tête de la fonction *essai*, des deux noms des paramètres fixes *par1* et *par2*, déclarés de manière classique ; les trois points servent à spécifier au compilateur l'existence de paramètres en nombre variable.

La déclaration :

```
va_list adpar ;
```

précise que *adpar* est un identificateur de liste variable. C'est lui qui nous servira à récupérer, les uns après les autres, les différents arguments variables.

Comme à l'accoutumée, une telle déclaration n'attribue aucune valeur à *adpar*. C'est effectivement la fonction *va_start* qui va permettre de l'initialiser à l'adresse du paramètre variable. Notez bien que cette dernière est déterminée par *va_start* à partir de la connaissance du nom du dernier paramètre fixe.

Le rôle de la fonction *va_arg* est double :

- d'une part, elle fournit comme résultat la valeur trouvée à l'adresse courante fournie par *adpar* (son premier argument), suivant le type indiqué par son second argument (ici *int*) ;
- d'autre part, elle incrémente l'adresse contenue dans *adpar*, de manière que celle-ci pointe alors sur l'argument variable suivant.

Ici, une instruction *while* nous permet de récupérer les différents arguments variables, sachant que le dernier a pour valeur *-1*.

Enfin, la norme prévoit que la macro *va_end* doit être appelée avant de sortir de la fonction concernée. Si vous manquez à cette règle, vous courez le risque de voir un prochain appel à la fonction conduire à un mauvais fonctionnement du programme.

Remarques

1 Les arguments variables peuvent être de types différents, à condition toutefois que la fonction soit en mesure de les connaître, d'une façon ou d'une autre.

2 La gestion de la fin de la liste des arguments variables est laissée au bon soin de l'utilisateur ; en effet, il n'existe aucune fonction permettant de connaître le nombre effectif de ces arguments. Cette gestion peut se faire :

   – par sentinelle, comme dans notre précédent exemple ;

   – par transmission, en argument fixe, du nombre d'arguments variables.

Informations complémentaires

En toute rigueur, *va_start* et *va_arg* ne sont pas de véritables fonctions, mais des « macros » ; cette distinction n'a que peu d'incidence sur leur utilisation effective. Les macros, beaucoup moins utilisées en C++ qu'en C, seront présentées au paragraphe 2.2 du chapitre 33.

16 Conséquences de la compilation séparée

16.1 Compilation séparée et prototypes

Nous avons déjà été amenés à utiliser des fonctions prédéfinies telles que *sqrt*. Pour ce faire, nous incorporions le fichier en-tête *cmath* qui contient les déclarations des fonctions mathématiques telles que *sqrt*. Nous savons que le module objet correspondant à cette fonction a déjà été compilé, qu'il figure dans une bibliothèque et qu'il sera incorporé par l'éditeur de liens pour créer le programme exécutable.

Cette démarche, dans laquelle on réunit plusieurs modules objets compilés de façon indépendante l'une de l'autre (ici votre *main* d'une part, *sqrt* d'autre part) peut s'appliquer à des fichiers sources conçus par l'utilisateur. On parle alors de « compilation séparée ». Par exemple, vous pourriez tout à fait placer dans des fichiers sources différents les fonctions que nous avons été amenés à créer auparavant (*fexple*, *echange*, *optimist*, *fct*) et les compiler séparément. Dans ce cas, la définition de la fonction ne figure plus dans le programme l'utilisant. On n'y trouvera que sa déclaration. Si certaines des fonctions que vous développez doivent être utilisées par plusieurs programmes, vous serez probablement amené à prévoir un fichier en-tête (nommé par exemple *MesFonc*) en contenant les déclarations, de façon à éviter d'éventuelles erreurs d'écriture (ou plutôt un fichier en-tête par groupe de fonctions ayant un rapport entre elles). Généralement, un tel fichier portera l'extension *.h*. Comme pour les fichiers en-têtes prédéfinis, vous incorporerez votre fichier en-tête par une directive *#include*. Il faut cependant savoir que la syntaxe en est légèrement modifiée pour les fichiers de l'utilisateur (utilisation de "…" au lieu de <...>) :

```
include "MesFonc.h"
```

Généralement, l'inclusion d'un fichier en-tête se fait à un niveau global comme dans ce schéma :

```
#include "MesFonc.h"  // Attention : "MesFonc.h" et non <MesFonc.h>
                      //  pour un fichier en-tête utilisateur
int main ()
{ ...                 // ici, on dispose des déclarations figurant dans MesFonc.h
}
void f()
{                     // ici, également
}
```

Bien que cela soit peu utilisé, il est possible, en théorie, d'effectuer une inclusion à un niveau local, comme dans ce schéma :

```
int main ()
{ #include "MesFonc.h"
  ...                 // ici, on dispose des déclarations figurant dans MesFonc.h
}
void f()
{                     // ici, non
}
```

16.2 Fonction manquante lors de l'édition de liens

Compte tenu de ces possibilités de compilation séparée, on voit qu'il est tout à fait possible d'écrire un programme dans lequel on a omis la définition d'une fonction (pour peu qu'elle soit correctement déclarée) :

```
void f() ;      // déclaration de f
int main ()
{ .....
  f() ;         // utilisation de f
  .....
}
```

La compilation se déroulera sans problème. En revanche, si lors de l'édition de liens, la définition de *f* n'est trouvée dans aucun module objet (y compris dans ceux constituant la bibliothèque standard), on obtiendra une erreur.

Remarque

Ne confondez pas les fichiers en-tête qui ne contiennent que les déclarations de fonctions avec les modules objets qui, quant à eux, contiennent bien le code exécutable correspondant à leur définition. L'un ne remplace pas l'autre. Certes, la confusion peut être entretenue par les fonctions de la bibliothèque standard dont on a l'impression qu'il suffit de citer les fichiers en-têtes contenant leur déclaration pour en disposer. En fait, le travail de recherche de l'éditeur de liens est totalement indépendant de la compilation et n'a aucun rapport avec l'éventuelle inclusion de fichiers en-tête. Vous pourriez par exemple **utiliser *sqrt*, sans incorporer *cmath***, pour peu que vous en fournissiez la déclaration.

16.3 Le mécanisme de la surdéfinition de fonctions

Dans notre étude de la surdéfinition des fonctions du paragraphe 11, nous avions examiné la manière dont le compilateur faisait le choix de la « bonne fonction », en raisonnant sur un seul fichier source à la fois. Mais on voit maintenant qu'il est tout à fait envisageable :

- de compiler dans un premier temps un fichier source contenant les différentes définitions d'une fonction (telle que *sosie* ou *chose* dans nos précédents exemples) ; on peut même éclater ces surdéfinitions dans plusieurs fichiers sources ;

- d'utiliser ultérieurement ces fonctions dans un autre fichier source en nous contentant d'en fournir les prototypes.

Or, pour que cela soit possible, l'éditeur de liens doit être en mesure d'effectuer le lien entre le choix opéré par le compilateur et la « bonne fonction » figurant dans un autre module objet. Cette reconnaissance est fondée sur la modification, par le compilateur, des noms « externes » des fonctions ; celui-ci fabrique un nouveau nom fondé d'une part sur le nom interne de la fonction, d'autre part sur le nombre et la nature de ses arguments.

Il est très important de noter que ce mécanisme s'applique à toutes les fonctions, qu'elles soient surdéfinies ou non (il est impossible de savoir si une fonction compilée dans un fichier source sera surdéfinie dans un autre). On voit donc qu'un problème se pose, dès que l'on souhaite utiliser dans un programme C++ une fonction écrite et compilée en C (ou dans un autre langage utilisant les mêmes conventions d'appel de fonction, notamment l'assembleur ou le Fortran). En effet, une telle fonction n'aura pas son nom modifié suivant le mécanisme évoqué. Une solution existe toutefois : déclarer une telle fonction en faisant précéder son prototype de la mention *extern "C"*. Par exemple, si nous avons écrit et compilé en C une fonction d'en-tête :

```
double fct (int n, char c) ;
```

et que nous souhaitons l'utiliser dans un programme C++, il nous suffira de fournir son prototype de la façon suivante :

```
extern "C" double fct (int, char) ;
```

Remarques

1 Il existe une forme « collective » de la déclaration *extern,* qui se présente ainsi :

```
extern "C"
{ void exple (int) ;
  double chose (int, char, float) ;
  .....
} ;
```

2 Le problème évoqué pour les fonctions C (assembleur ou Fortran) se pose, a priori, pour toutes les fonctions de la bibliothèque standard C que l'on réutilise en C++. En fait, dans la plupart des environnements, cet aspect est automatiquement pris en charge au niveau des fichiers en-tête correspondants (ils contiennent des déclarations *extern* conditionnelles).

3 Il est possible d'employer, au sein d'un même programme C++, une fonction C (assembleur ou Fortran) et une ou plusieurs autres fonctions C++ de même nom (mais d'arguments différents). Par exemple, nous pouvons utiliser dans un programme C++ la fonction C *fct* précédente et deux fonctions C++ d'en-tête :

```
void fct (double x)
void fct (float y)
```

en procédant ainsi :

```
extern "C" void fct (int) ;
void fct (double) ;
void fct (float) ;
```

Suivant la nature de l'argument d'appel de *fct*, il y aura bien appel de l'une des trois fonctions *fct*. Notez qu'il n'est pas possible de mentionner plusieurs fonctions C de nom *fct*.

16.4 Compilation séparée et variables globales

N.B. Ce paragraphe a surtout un intérêt si vous devez exploiter du code utilisant cette technique déconseillée de variables globales. Il peut également servir à distinguer la notion de portée (compilation) de celle de lien (édition de liens).

16.4.1 La portée d'une variable globale – la déclaration extern

A priori, la portée d'une variable globale semble limitée au fichier source dans lequel elle a été définie. Ainsi, supposez que l'on compile séparément ces deux fichiers source :

```
source 1                    source 2
int x ;                     void fct2()  { ..... }
int main () { ..... }       void fct3()  { ..... }
void fct1() { ..... }
```

Il ne semble pas possible, dans les fonctions *fct2* et *fct3* de faire référence à la variable globale *x* déclarée dans le premier fichier source (alors qu'aucun problème ne se poserait si l'on réunissait ces deux fichiers source en un seul, du moins si l'on prenait soin de placer les instructions du second fichier à la suite de celles du premier).

En fait, C++ prévoit une déclaration permettant de spécifier qu'une variable globale a déjà été définie dans un autre fichier source. Celle-ci se fait à l'aide du mot-clé *extern*. Ainsi, en faisant précéder notre second fichier source de la déclaration :

```
extern int x ;
```

il devient possible de mentionner la variable globale *x* (déclarée dans le premier fichier source) dans les fonctions *fct2* et *fct3*.

Remarque

Cette déclaration *extern* n'effectue **pas de réservation d'emplacement de variable**. Elle ne fait que préciser que la variable globale *x* est définie par ailleurs et elle en indique le type.

16.4.2 Les variables globales et l'édition de liens

Supposons que nous ayons compilé les deux fichiers source précédents et voyons d'un peu plus près comment l'éditeur de liens est en mesure de rassembler correctement les deux modules objets ainsi obtenus. En particulier, examinons comment il peut faire correspondre au symbole *x* du second fichier source l'adresse effective de la variable *x* définie dans le premier.

D'une part, après compilation du premier fichier source, on trouve, dans le module objet correspondant, une indication associant le symbole *x* et son adresse dans le module objet. Autrement dit, contrairement à ce qui se produit pour les variables locales, pour lesquelles ne subsiste aucune trace du nom après compilation, le nom des variables globales continue à exister au niveau des modules objets. On retrouve là un mécanisme analogue à ce qui se passe pour les noms de fonctions, lesquels doivent bien subsister pour que l'éditeur de liens soit en mesure de retrouver les modules objets correspondants.

D'autre part, après compilation du second fichier source, on trouve, dans le module objet correspondant, une indication mentionnant qu'une certaine variable de nom *x* provient de l'extérieur et qu'il faudra en fournir l'adresse effective.

Ce sera effectivement le rôle de l'éditeur de liens que de retrouver dans le premier module objet l'adresse effective de la variable *x* et de la reporter dans le second module objet.

Ce mécanisme montre que s'il est possible, par mégarde, de réserver des variables globales de même nom dans deux fichiers source différents, il sera, par contre, en général, impossible d'effectuer correctement l'édition de liens des modules objets correspondants (certains éditeurs de liens peuvent ne pas détecter cette anomalie). En effet, dans un tel cas, l'éditeur de liens se trouvera en présence de deux adresses différentes pour un même identificateur, ce qui est illogique.

16.4.3 Les variables globales cachées – la déclaration static

Il est possible de « cacher » une variable globale, c'est-à-dire de la rendre inaccessible à l'extérieur du fichier source où elle a été définie (on dit aussi « rendre confidentielle » au lieu de « cacher » ; on parle alors de « variables globales confidentielles »). Il suffit pour cela d'utiliser la déclaration **static** comme dans cet exemple :

```
static int a ;
int main ()  { ..... }
void fct()   { ..... }
```

Sans la déclaration *static*, *a* serait une variable globale ordinaire. Par contre, cette déclaration demande qu'aucune trace de *a* ne subsiste dans le module objet résultant de ce fichier source. Il sera donc impossible d'y faire référence depuis une autre source par *extern*. Mieux, si une autre variable globale apparaît dans un autre fichier source, elle sera acceptée à l'édition de liens puisqu'elle ne pourra pas interférer avec celle du premier source.

17 La spécification *inline*

Comme on peut s'y attendre, le code exécutable correspondant à une fonction est généré une seule fois par le compilateur. Néanmoins, pour chaque appel de cette fonction, le compilateur doit prévoir, non seulement le branchement au code exécutable correspondant, mais également des instructions utiles pour établir la communication entre le programme appelant et la fonction, notamment :

- sauvegarde de l'état courant (valeurs de certains registres de la machine, par exemple) ;

- allocation d'espace sur la pile et copie des valeurs des arguments ;
- branchement à la fonction (dont l'adresse définitive sera en fait fournie par l'éditeur de liens) ;
- recopie de la valeur de retour ;
- restauration de l'état courant et retour dans le programme appelant.

Dans le cas de « petites fonctions », ces différentes instructions de « service » peuvent représenter un pourcentage important du temps d'exécution total de la fonction. Lorsque l'efficacité du code devient un critère important, C++ vous permet de gagner du temps dans l'appel des fonctions, au détriment de la taille du code, grâce à la spécification *inline*.

Voyez cet exemple :

```
// FonctionEnLigne
#include <cmath>         // pour sqrt
#include <iostream>
using namespace std ;
    /*******  declaration definition d'une fonction en ligne ********/
inline double norme (double vec[3])
{  double s = 0 ;
   for (int i=0 ; i<3 ; i++)  s+= vec[i] * vec[i] ;
   return sqrt(s) ;
}
    /******* exemple d'utilisation d'une fonction en ligne *********/
int main ()
{  double v1[3], v2[3] ;
   for (int i=0 ; i<3 ; i++)
   { v1[i] = i ; v2[i] = 2*i-1 ; }
   cout << "norme de v1 : " << norme(v1) << endl ;
   cout << "norme de v2 : " << norme(v2) << endl ;
}
```

```
norme de v1 : 2.23607
norme de v2 : 3.31662
```

Exemple de définition et d'utilisation d'une fonction en ligne

La fonction *norme* a pour but de calculer la norme d'un vecteur à trois composantes qu'on lui fournit en argument.

La présence du mot *inline* demande au compilateur de traiter la fonction *norme* différemment d'une fonction ordinaire. À chaque appel de *norme*, le compilateur devra incorporer au sein du programme les instructions correspondantes (en langage machine[12]). Le mécanisme habi-

12. Notez qu'il s'agit bien ici d'un travail effectué par le compilateur lui-même, alors que dans le cas d'une macro, un travail comparable était effectué par le préprocesseur.

tuel de gestion de l'appel et du retour n'existera plus (il n'y a plus besoin de sauvegardes, recopies...), ce qui permet une économie de temps. En revanche, les instructions correspondantes seront générées à chaque appel, ce qui consommera une quantité de mémoire croissant avec le nombre d'appels.

Il est très important de noter que, par sa nature même, une fonction en ligne doit être définie dans le même fichier source que celui où on l'utilise. **Elle ne peut plus être compilée séparément !** Cette absence de possibilité de compilation séparée constitue une contrepartie notable aux avantages offerts par la fonction en ligne. En effet, pour qu'une même fonction en ligne puisse être partagée par différents programmes, il faudra absolument la placer dans un fichier en-tête[13].

En revanche, cette définition d'une fonction en ligne joue bien le rôle d'une déclaration, de sorte qu'il n'est pas nécessaire d'en prévoir une déclaration supplémentaire.

Remarques

1 La déclaration *inline* constitue une demande effectuée auprès du compilateur. Ce dernier peut éventuellement (par exemple, si la fonction est volumineuse) ne pas l'introduire en ligne et en faire une fonction ordinaire. De même, si vous utilisez quelque part (au sein du fichier source concerné) l'adresse d'une fonction déclarée *inline*, le compilateur en fera une fonction ordinaire (dans le cas contraire, il serait incapable de lui attribuer une adresse et encore moins de mettre en place un éventuel appel d'une fonction située à cette adresse).

2 Notez que, si nous avions fourni la définition de *norme* après celle de *main*, il aurait été nécessaire de déclarer notre fonction, comme n'importe quelle autre, en utilisant également le mot-clé *inline* :

```
inline double norme (double [3])
```

Nous avons volontairement évité ici de procéder de cette manière car, dès lors qu'une fonction en ligne figure dans un fichier en-tête, son incorporation précédera son utilisation.

Informations complémentaires

C++ a hérité de C la possibilité de définir des « macros ». Il s'agit d'instructions fournies au préprocesseur qui effectue alors des substitutions paramétrées de texte. La macro s'appelle comme une fonction (d'ailleurs, certaines des « fonctions » de la bibliothèque standard du C sont des macros) et elle présente quelques similitudes avec l'emploi de *inline* :

13. À moins d'en écrire plusieurs fois la définition, ce qui ne serait pas « raisonnable », compte tenu des risques d'erreurs que cela comporte.

– le code correspondant est introduit à chaque appel (au niveau du préprocesseur, cette fois, et non plus au niveau du compilateur) ;

– on obtient un gain de temps d'exécution, en contrepartie d'une perte d'espace mémoire.

Mais la ressemblance s'arrête là, car l'emploi des macros présente de très grands risques (notamment d'effets de bord). C'est ce qui explique que les macros soient déconseillées en C++ (*inline* n'existe pas en C !). Nous étudierons les macros au paragraphe 2.2 du chapitre 33.

18 Terminaison d'un programme

Nous avons déjà vu qu'on peut arrêter le déroulement de la fonction *main* par une instruction *return*, accompagnée d'une valeur entière indiquant si l'exécution s'est déroulée convenablement (valeur 0) ou non (valeur non nulle). On peut placer autant d'instructions *return* dans le *main* qu'on le ferait dans une fonction usuelle.

Mais, si une instruction *return* dans le *main* met fin à l'exécution du programme, il n'en va plus de même pour une instruction *return* figurant dans une fonction autre que *main*. Or, dans certaines situations, en général en cas d'erreur importante, on aimera pouvoir, depuis une fonction, mettre fin à l'exécution du programme, sans devoir « remonter » la pile des différents appels. Il est alors possible de faire appel à la fonction standard *exit* (prototype dans *cstdlib*), à laquelle on fournit, ici encore, un entier utilisable comme compte rendu du déroulement du programme.

Informations complémentaires

En toute rigueur, la fonction *exit* ferme les fichiers ouverts, détruit convenablement les objets dynamiques, mais pas les objets automatiques. Néanmoins, on considère que l'appel de cette fonction correspond à une « fin normale » du programme. La valeur de l'argument précise s'il s'agit d'une fin sans aucun échec (0) ou d'une fin avec échec.

Il existe une autre fonction *abort* (prototype dans *cstdlib*), utilisée pour des fins anormales, qui peut être appelée par certaines fonctions de la bibliothèque en cas de difficultés majeures compromettant la bonne poursuite de l'exécution (nous en verrons un exemple dans le cadre de la gestion des exceptions). Celle-ci met fin brutalement au programme, sans aucune intervention au niveau des fichiers ou des objets.

Enfin, il est possible de demander qu'une ou plusieurs fonctions de son choix soient exécutées lors de la fin d'un programme, en utilisant la fonction *atexit*[14].

14. On trouvera plus d'informations sur ces fonctions, héritées du langage C, dans *Le guide complet du langage C*, du même auteur, chez le même éditeur.

11

Les vecteurs, les tableaux natifs et les chaînes C

En programmation, on parle de tableau pour représenter un ensemble d'éléments de même type, désigné par un nom unique, chaque élément étant repéré par un « indice » précisant sa position au sein de l'ensemble.

Comme tous les langages, C++ permet de manipuler des tableaux. Mais les outils pour le faire se sont enrichis au fil des différentes versions du langage. Avant la norme C++98, on disposait de ce que nous appelerons les « tableaux natifs » ; il s'agissait de tableaux dont la taille (nombre d'éléments) était fixe et définie à la compilation. Comme nous le verrons, ils souffraient de graves lacunes en ce qui concerne leur transmission en argument d'une fonction. Avec la norme C++98 est apparue la bibliothèque standard, laquelle dispose d'un type *vector* qui présente l'avantage d'être à la fois de taille modifiable et facilement transmissible à une fonction.

Nous commencerons par ce dernier en l'introduisant sur un exemple simple qui en montre l'intérêt. Nous étudierons ensuite les principales propriétés de ces vecteurs (le chapitre 27 en fera une présentation plus approfondie).

Puis nous aborderons les tableaux natifs à une ou plusieurs dimensions et la manière de les initialiser lors de leur déclaration. Nous verrons en quoi ils sont liés aux pointeurs natifs, à savoir qu'un nom de tableau est un pointeur constant, ce qui nous amènera, au passage, à fournir quelques compléments sur l'arithmétique des pointeurs natifs. Nous montrerons ensuite comment exploiter cette particularité pour transmettre un tableau natif en argument

d'une fonction. Nous aborderons alors la gestion de tableaux dynamiques, soit à l'aide de pointeurs natifs, soit à l'aide de pointeurs intelligents.

Nous étudierons ensuite ce que l'on nomme généralement les « chaînes de style C », à savoir une convention de représentation de chaînes, héritée du langage C, dont il subsiste encore quelques traces dans C++, malgré une apparente redondance avec le type *string*.

1 Les vecteurs

1.1 Exemple de présentation des vecteurs

Supposons que nous souhaitions déterminer, à partir de notes d'élèves, fournies en données, combien d'entre elles sont supérieures à la moyenne de la classe.

S'il ne s'agissait que de calculer simplement la moyenne de ces notes, il nous suffirait d'en calculer la somme, en les cumulant dans une variable, au fur et à mesure de leur lecture. Mais, ici, il nous faut à nouveau pouvoir consulter les notes pour déterminer combien d'entre elles sont supérieures à la moyenne ainsi obtenue. Il est donc nécessaire de pouvoir mémoriser ces notes.

Pour ce faire, il n'est pas possible de prévoir autant de variables scalaires différentes qu'il y a de notes puisque nous n'en connaissons pas le nombre. Et même si ce nombre était connu, cette démarche ne serait guère envisageable dès que le nombre de notes est important. Le vecteur va offrir une solution convenable à ce problème en donnant un nom unique à l'ensemble des notes et en repérant chaque note par un numéro nommé indice. C'est ce que montre le programme suivant (notez l'inclusion du fichier en-tête *vector*) :

```
// ExempleVector
#include <iostream>
#include <vector>                  // pour le type vector
using namespace std ;
int main()
{  unsigned int nb ;               // nombre de notes
   cout << "Donnez le nombre de notes : " ;  cin >> nb ;
   vector <float> notes  (nb) ;    // pour les nb notes
   for (unsigned int i=0 ; i<nb ; i++)      // lecture des notes
    { cout << "Donnez la note numero " << i+1 << " : " ;
      cin >> notes[i] ;
    }
   float som = 0 ;                 // calcul de la somme des notes
   for (unsigned int i=0 ; i<nb ; i++)  som += notes[i] ;
      // ou (C++11 - voir § 1.4) :  for (auto note:notes)  som += note ;
   float moy = som / nb ;
   cout << "Moyenne de la classe : " << moy << endl ;
```

```
   int nbm = 0 ;                    // calcul nb notes >= moyenne
   for (unsigned int i=0 ; i<nb ; i++ ) if (notes[i] > moy) nbm++ ;
      // ou (C++11 - voir § 1.4) :  for (auto note:notes) if (note>moy) nbm ++ ;
   cout << nbm << " eleves ont plus de cette moyenne" << endl;
}
```

```
Donnez le nombre de notes : 6
Donnez la note numero 1 : 12.5
Donnez la note numero 2 : 11
Donnez la note numero 3 : 8.25
Donnez la note numero 4 : 13.5
Donnez la note numero 5 : 15.25
Donnez la note numero 6 : 13
Moyenne de la classe : 12.25
4 eleves ont plus de cette moyenne
```

Exemple d'utilisation d'un vecteur

L'exécution de l'instruction :

```
vector <float> notes  (nb) ;
```

crée un vecteur de *nb* éléments (on dit aussi de dimension *nb*) de type *float*. Chaque élément de ce vecteur est repéré par un « indice » indiquant sa position dans le vecteur. Conventionnellement, comme pour les chaînes, le premier élément porte le numéro 0. La première note sera donc désignée par *notes[0]*, la troisième par *notes[2]*, la dernière par *notes[nb-1]*. Plus généralement, *notes[i]* désigne un élément dont la position est fournie par la valeur de *i* et représente donc une valeur de type *float*.

Notez bien que le vecteur est créé avec un nombre d'éléments égal à la valeur de *nb* au moment où l'on exécute l'instruction de déclaration. Si la valeur de *nb* évolue par la suite, cela ne modifiera nullement le nombre d'éléments du vecteur.

1.2 Les éléments et les indices d'un vecteur

1.2.1 Quelques règles

Comme on le voit, le type vecteur se définit à l'aide du mot *vector*, suivi entre crochets angulaires du type de ses éléments, lesquels sont donc tous d'un même type. La dimension (nombre d'éléments) est fournie sous forme d'une expression entière dont la valeur est calculée au moment de l'exécution de l'instruction de déclaration correspondante.

Un **élément de vecteur** est une *lvalue* modifiable. Il peut donc apparaître à gauche d'un opérateur d'affectation comme dans (*t* étant supposé de type *vector<int>*) :

```
t[2] = 5
```

Il peut aussi apparaître comme opérande d'un opérateur d'incrémentation, comme dans :

```
t[3]++              --t[i]
```

Notez toutefois, que les éléments de *t* ne seraient pas modifiables si l'on avait déclaré :

```
const vector<int> t
```

Un **indice** peut prendre la forme de n'importe quelle expression arithmétique d'un type entier quelconque (*int*, *short*, *long* avec leurs variantes non signées). Par exemple, si *n*, *p*, *k* et *j* sont de type *int* et *t* de type *vector<int>*, ces notations sont correctes :

```
t[n-3]
t[3*p-2*k+j%l]
```

Il en va de même, si *c1* et *c2* sont de type *char*, de :

```
t[c1+3]
t[c2-c1]
```

puisque les expressions correspondantes sont de type *int*.

En théorie, un indice peut également être de type caractère ; dans ce cas, sa valeur sera simplement convertie en *int* suivant les règles habituelles (attention à l'attribut de signe !). En revanche, la tentative d'utilisation d'indices de type flottant conduira à une erreur de compilation.

1.2.2 Absence de contrôle d'indice

Il est très important de noter qu'aucun contrôle n'est effectué sur la valeur de l'indice lorsque l'on cherche à accéder à un vecteur à l'aide de l'opérateur [] comme nous l'avons fait jusqu'ici. Notez, que, pour être efficace, ce contrôle d'indice devrait pouvoir se faire, non seulement dans le cas où l'indice est une constante, mais également dans tous les cas où il s'agit d'une expression quelconque. Cela nécessiterait l'incorporation, dans le programme objet, d'instructions supplémentaires assurant cette vérification lors de l'exécution, ce qui conduirait à une perte de temps. Pour comprendre les conséquences d'un tel risque de « débordement », il faut savoir que les éléments d'un vecteur sont rangés en mémoire de façon consécutive. Lorsque le compilateur rencontre une *lvalue* telle que *t[i]*, il en détermine l'adresse en ajoutant à l'adresse de début du vecteur *t*, un décalage proportionnel à la valeur de *i* (et aussi proportionnel à la taille de chaque élément du vecteur).

Voici un exemple de programme quelque peu disuasif :

```
//DebordementVector
#include <iostream>
using namespace std;
#include <vector>
int main()
{   const int N = 10 ;   // ou constexpr depuis C++11
    vector<int> t(N) ;
    t[12] = 5 ;   // on modifie un emplacement situe hors du tableau
    cout << "t[12] =  " << t[12] << endl ;
    cout << "t[100] = " << t[100] << endl ;   // t[100] est en dehors de t
    cout << "t[-3] =  " << t[-3] << endl ;    // t[-3]  est en dehors de t
}
```

```
t[12] =  5
```

```
t[100] = 0
t[-3] =  196800
```

Conséquences de l'absence de contrôle d'indice de l'opérateur []

Le vecteur *t* a été créé ici avec 10 éléments. Le programme peut donner l'illusion qu'il en possède d'avantage, mais il n'en est rien. Les éléments d'indices 12, 100 et -3 se trouvent simplement « quelque part » en dehors de la zone prévue pour le tableau.

Remarques

1 Il existe une autre manière d'accéder aux éléments d'un vecteur dont nous parlerons au chapitre 27 (utilisation de la fonction *at*), laquelle conduit bien à un contrôle des indices.

2 Voyez l'instruction suivante (dans laquelle nous avons malencontreusement remplacé les crochets par des parenthèses) :

```
vector<int> t[10] ;
```

Elle ne provoque par d'erreur de compilation car, comme nous le verrons plus loin, *t* sera alors un tableau natif de 10 éléments, chacun du type *vector<int>* et vide (0 éléments).

3 Nous avons déjà eu l'occasion de distinguer les données statiques, automatiques et dynamiques. A priori, on peut penser qu'un vecteur déclaré dans une fonction comme dans nos précédents exemples, est de classe automatique. En fait, l'emplacement nécessaire à ses éléments est alloué dynamiquement dans le tas (et libéré à la sortie de la fonction) et sur la pile, on ne trouve que deux informations : dimension du vecteur, adresse du premier élément dans le tas. Les vecteurs permettent donc de réaliser implicitement une gestion dynamique, sans avoir à la programmer explicitement. La même remarque s'appliquerait à un vecteur statique dont les éléments seraient alloués dynamiquement en début de programme et libérés à sa fin.

1.3 Initialisation d'un vecteur

Lorsque vous exécutez la déclaration :

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

le vecteur *v* voit ses éléments initialisés à 0. Il s'agit là d'une différence importante avec les variables simples qui, comme nous l'avons vu, n'étaient pas initialisées. Nous verrons plus tard que cela tient à ce qu'un vecteur est un objet, lequel est initialisé par appel de ce que l'on nomme un constructeur.

Il existe d'autres façons de créer un vecteur que nous examinons maintenant.

- On peut créer un **vecteur de taille donnée**, en initialisant tous ses éléments avec une valeur donnée (compatible avec le type déclaré pour les éléments), comme dans (*n* étant de type *int*, *x* une valeur de type *double* ou compatible) :

```
vector<double> v (n, 0.5) ;      // n éléments de type double initialisés à 0.5
vector<string> vx (3, "xxx") ;   // 3 éléments de type string initialisés à "xxx"
```

Notez que, comme pour les chaînes, l'utilisation des accolades (C++11) ici conduirait à une toute autre signification (une liste de valeurs) comme nous le verrons un peu plus loin.

- On peut également **créer un vecteur « vide »**, c'est-à-dire ne contenant aucun élément :

```
vector <string> v ;   // vecteur de chaînes, ne contenant aucun élément
```

Bien entendu, cette dernière possibilité ne présentera d'intérêt que dans l'un des cas suivants :

  – on sera amené par la suite à affecter à *v* la valeur d'un autre vecteur (de type *vector<string>*) comme nous le verrons au paragraphe 1.5

  – on sera amené à ajouter des éléments à ce vecteur, comme nous apprendrons à le faire dans le chapitre 27.

- Depuis C++11, on peut **fournir explicitement toutes les valeurs initiales** (d'un type implicitement compatible avec le type des éléments) :

```
  // vecteur de 4 éléments de type double
vector<double> v {1, 2.5, 8, 3.2}              // initialisation directe
vector<double> v = {1, 2.5, 8, 3.2}            // initialisation copie
  // vecteur de 2 éléments de type string
vector <string> vs {"bonjour", "bonsoir"} ;    // initialisation directe
vector <string> vs = {"bonjour", "bonsoir"} ;  // initialisation copie
```

Remarque

Comme avec les chaînes, on notera bien que :

```
vector<int> v (5) ;
```

correspond à un vecteur de 5 éléments tandis que :

```
vector<int> v {5} ;   //  C++11
```

correspond à un vecteur d'un seul élément de valeur 5.

1.4 Parcours d'un vecteur avec *for* pour les séquences

Si nous avons déclaré :

```
vector<int> v(n) ;
```

nous pouvons bien sûr parcourir tous les éléments de *v* avec une boucle *for* classique :

```
for (unsigned int i=0 ; i<n ; i++) { // traitement de v[i] }
```

Notez que, comme pour le type *string*, on dispose d'une f**onction *size*** (*v.size()* représente le nombre d'éléments de *v*), de sorte que cette boucle peut aussi s'écrire :

```
for (unsigned int i=0 ; i<v.size() ; i++) { // traitement de v[i] }
```

Mais, comme nous l'avons déjà vu, C++11 a introduit une version de *for* adaptée aux « séquences » d'éléments comme le type *string* ou le type *vector*. Aussi, pouvons-nous procéder de cette manière pour notre vecteur *v* :

```
for (int e : v)      // ou for (auto e : v)
{ // traitement de e  (copie d'un élément de v)
}
```

Cela signifie que les instructions de la boucle, écrites ici pour un élément *e*, seront appliquées à chacun des éléments du vecteur *v*. Mais, alors que dans la première formulation, *v[i]* désignait bien la *lvalue* modifiable correspondant à un élément d'indice *i*, dans la seconde formulation, cette fois, ***e* désigne une copie d'un élément de *v***. Il est toufefois possible de demander de travailler directement sur l'élément lui-même en utilisant une référence comme dans :

```
for (int & e : v)      // ou for (auto & e : v)
{ // traitement de e (référence à un élément de v)
}
```

Voici un petit programme (C++11) illustrant ces possibilités :

```
// VectorForGene
#include <iostream>
#include<vector>
using namespace std;
int main()
{ vector<int> v { 1, 3, 5 } ;
  cout << "En A, v = " ;
  for (auto e : v) cout << e << " " ; cout << endl ;
  for (unsigned int i=0 ; i<v.size() ; i++) v[i]++ ;
  cout << "En B, v = " ;
  for (auto e : v) cout << e << " " ; cout << endl ;
  for (auto e : v) e++ ;
  cout << "En C, v = " ;
  for (auto e : v) cout << e << " " ; cout << endl ;
  for (auto & e : v) e++ ;
  cout << "En D, v = " ;
  for (int e : v) cout << e << " " ; cout << endl ;
}
```

```
En A, v = 1 3 5
En B, v = 2 4 6
En C, v = 2 4 6
En D, v = 3 5 7
```

Différentes façons de parcourir les éléments d'un vecteur

1.5 Affectation de vecteurs

Si l'on a déclaré deux vecteurs *v1* et *v2* ayant **exactement le même type** d'éléments (mais pas nécessairement le même nombre), comme dans

```
vector<int> v1 { 1, 3, 5 } ;       // Avant C++11, il faudrait initialiser
vector<int> v2 { 2, 4, 6, 8 } ;    // individuellement chacun des éléments
```

il est possible d'écrire une affectation telle que :

```
v1 = v2 ;
```

Elle conduit bien à un vecteur *v1* qui contient maintenant 4 valeurs, copies de celles de *v2*. C'est ce que montre ce petit programme (ici, par souci de simplicité, nous utilisons la boucle *for* adaptée aux séquences introduite par C++11 et présentée au paragraphe 1.4) :

```
// AffectationVector
#include <iostream>
#include <vector>
using namespace std;
int main()
{ vector<int> v1 { 1, 3, 5 } ;
  vector<int> v2 { 2, 4, 6, 8 } ;
  cout << "En A, les " << v1.size() << " valeurs de v1 sont : " ;
  for (auto v : v1) cout << v << " " ; cout << endl ;    // C++11 (voir § 1.4)
  cout << "En A, les " << v2.size() << " valeurs de v2 sont : " ;
  for (auto v : v2) cout << v << " " ; cout << endl ;    // C++11 (voir §1.4)
  v1 = v2 ;
  cout << "En B, les " << v1.size() << " valeurs de v1 sont : " ;
  for (auto v : v1) cout << v << " " ; cout << endl ;    // C++11 (voir § 1.4)
  cout << "En B, les " << v2.size() << " valeurs de v2 sont : " ;
  for (auto v : v2) cout << v << " " ; cout << endl ;    // C++11 (voir § 1.4)
}
```

```
En A, les 3 valeurs de v1 sont : 1 3 5
En A, les 4 valeurs de v2 sont : 2 4 6 8
En B, les 4 valeurs de v1 sont : 2 4 6 8
En B, les 4 valeurs de v2 sont : 2 4 6 8
```

Affectation (globale) de vecteurs

Ce comportement, pour naturel qu'il paraisse, conduit ici (implicitement) à allouer dynamiquement un nouvel emplacement pour les 4 nouvelles valeurs de *v1* et à libérer l'ancien emplacement de 3 valeurs.

Remarque

Nous verrons que cette affectation globale n'existera pas pour les tableaux natifs.

1.6 Vecteurs transmis en argument d'une fonction

1.6.1 Par défaut, la transmission se fait par valeur

Nous avons vu comment, par défaut, les variables d'un type simple ou d'un type *string* étaient transmises par valeur. Il en va de même pour les vecteurs, ce qui implique donc une recopie de leurs valeurs, comme le montre cet exemple :

```
// vecteurArgument
#include <iostream>
#include <vector>
using namespace std ;
void f (vector<int> v) ;
void f(vector<int> v)
{ cout << "En 1 de f, v = " ;
  for (auto n : v) cout << n << " " ; cout << endl ;    // C++11 (voir § 1.4)
  for (auto & n : v) n++ ;                              // C++11 (voir § 1.4)
  cout << "En 2 de f, v = " ;
  for (auto n : v) cout << n << " " ; cout << endl ;    // C++11 (voir § 1.4)
}
int main()
{ vector<int> v1 {1, 3, 5 } ;
  cout << "En A,      v = " ;
  for (auto n : v1) cout << n << " " ; cout << endl ;   // C++11 (voir § 1.4)
  f (v1) ;
  cout << "En B,      v = " ;
  for (auto n : v1) cout << n << " " ; cout << endl ;   // C++11 (voir § 1.4)
}
```

```
En A,      v = 1 3 5
En 1 de f, v = 1 3 5
En 2 de f, v = 2 4 6
En B,      v = 1 3 5
```

Transmission d'un vecteur par valeur

Les valeurs de *v1* ont été recopiées dans un vecteur *v*, local à *f*[1]. Les modifications effectuées sur *v* ne se retrouvent pas sur *v1*, après retour dans la fonction *main*.

1.6.2 Transmission d'un vecteur par référence ou par pointeur

Le mode de transmission par valeur sera rarement utilisé pour d'évidentes raisons d'efficacité. Mais il est naturellement possible d'utiliser une transmission par référence ou par pointeur.

Ainsi, en modifiant simplement de cette manière l'en-tête (et la déclaraton) de *f* dans le précédent programme :

```
void f (vector<int> & v)
```

nous obtiendrons les résultats suivants, montrant bien que les modifications ont été opérées par *f* directement sur *v1* :

```
En A,      v = 1 3 5
En 1 de f, v = 1 3 5
En 2 de f, v = 2 4 6
En B,      v = 2 4 6
```

1. En toute rigueur, seules les informations de taille et d 'adresse des éléments sont de type automatique ; l'emplacement pour les copies des valeurs étant alloué dynamiquement à l'entrée dans *f* et libéré à la sortie.

Nous obtiendrions exactement les mêmes résultats, avec une transmission par pointeur, en écrivant *f* de cette manière :

```
void f (vector<int> * v)
{ cout << "En 1 de f, v = " ;
  for (auto n : *v) cout << n << " " ;
  cout << endl ;
  for (auto & n : *v) n++ ;
  cout << "En 2 de f, v = " ;
  for (auto n : *v) cout << n << " " ;
  cout << endl ;
}
```

et en remplaçant l'appel *f(v1)* par *f(&v1)*.

1.7 Autres possibilités du type *vector*

Comme nous l'avons dit en introduction, le type *vector* sera étudié en détail au chapitre 27. Nous y verrons qu'il possède les propriétés générales des conteneurs séquentiels en ce qui concerne notamment :

- la notion d'itérateur ;
- les possibilités de comparaison ;
- l'insertion et la suppression d'éléments.

Nous y verrons également que les vecteurs disposent d'une fonction *pushback* d'ajout d'un élément en fin de vecteur et d'outils spécifiques d'optimisation de la gestion de la mémoire qui leur est allouée.

2 Les tableaux natifs

Bien que le type *vector* existe depuis C++98, les tableaux natifs restent utilisés dans des codes postérieurs à cette date pour des raisons diverses : habitude, efficacité, simplicité d'apprentissage. Aussi, bien qu'il soit possible d'écrire des codes sans recourir aux tableaux natifs, il est difficile de les ignorer, d'où la présence de ce grand paragraphe.

N.B. *Seuls les paragraphes 2.1, 2.3, 2.4 et 2.7 seront utilisés dans la suite de cet ouvrage.*

2.1 Les tableaux natifs

2.1.1 Exemple d'utilisation d'un tableau natif

Comme les vecteurs, le tableaux natifs sont définis par le type de leurs éléments et leur nombre (dimension). Mais, et c'est déjà là une lacune par rapport aux vecteurs, cette dimension doit être connue à la compilation. Voici une adaptation du programme du paragraphe 1.1 où nous avons dû prévoir cette fois un nombre donné de notes (ici 6) :

```
// ExempleTableauNatif
#include <iostream>
using namespace std ;
int main()
{ const int NB = 6 ;   // nombre de notes
  float notes[NB] ;    // pour les NB notes
  for (int i=0 ; i<NB ; i++)   // simplifiable avec C++11 - voir & 2.1.3
  { cout << "Donnez la note numero " << i+1 << " : " ;
    cin >> notes[i] ;
  }
  float som = 0 ;
  for (int i=0 ; i<NB ; i++)  som += notes[i] ; // Avec C++11 - voir & 2.1.3
  float moy = som / NB ;
  cout << "Moyenne de la classe : " << moy << endl ;
  int nbm = 0 ;
  for (int i=0 ; i<NB ; i++ )      // simplifiable avec C++11 - voir & 2.1.3
    if (notes[i] > moy) nbm++ ;
  cout << nbm << " eleves ont plus de cette moyenne" ;
}
```

```
Donnez la note numero 1 : 12.5
Donnez la note numero 2 : 11
Donnez la note numero 3 : 8.25
Donnez la note numero 4 : 13.5
Donnez la note numero 5 : 15.25
Donnez la note numero 6 : 13
Moyenne de la classe : 12.25
4 eleves ont plus de cette moyenne
```

Exemple d'utilisation d'un tableau

La déclaration (notez bien l'emploi de crochets ici et non de parenthèses) :

```
float notes[NB] ;    // pour les NB notes
```

réserve l'emplacement pour *NB* éléments de type *float*. Cette fois, contrairement à ce qui se produisait avec les vecteurs, nous avons affaire à une allocation automatique (sur la pile)[2]. Là encore, les éléments sont repérés par un indice, fourni sous forme d'une expression entière, le premier élément portant le numéro 0.

2.1.2 Quelques règles

La dimension d'un tableau natif (son nombre d'éléments) doit donc être connue au moment de la compilation. Ces déclarations sont correctes :

```
const int N = 50 ;   // ou , depuis C++11 : constexpr int N = 50 ;
int t[N] ;
float h [2*N+1] ;
```

2. Du moins pour les tableaux déclarés dans une fonction, car il peut exister des tableaux de classe statique.

En revanche, cette construction est incorrecte :

```
int nel ;
 .....
cin >> nel ;
int u[nel] ;   // non : nel n'est pas connu du compilateur
```

Les propriétés des indices et des éléments de tableau sont exactement les mêmes que pour les vecteurs :

- l'indice peut prendre la forme de n'importe quelle expression arithmétique entière ;
- si *t* est un tableau et *exp* une expression entière, une notation telle que *t[exp]* est une *lvalue* modifiable (sauf, bien sûr, si *t* a été déclaré constant).

Comme avec les vecteurs, aucun contrôle d'indice n'est mis en place. En revanche, il n'existe pas l'équivalent de la fonction *at*. En outre, il n'existe pas d'affectation globale entre tableaux de même type (même s'ils sont de même dimension) :

```
int t1[10] ;
int t2[10] ;
  .....
t1 = t2 ;    // erreur de compilation
```

2.1.3 Parcours des éléments avec *for* pour les séquences (C++11)

On peut parcourir l'ensemble des éléments d'un tableau avec l'instruction *for* comme nous l'avons fait dans notre exemple. Mais on peut également, depuis C++11, utiliser l'instruction *for* adaptée aux séquences, en n'oubliant pas de prévoir une référence dans le cas où l'on souhaite modifier la valeur courante dans la boucle. Par exemple, les instructions de lecture des notes de notre exemple pourraient s'écrire (notez bien ici le &) :

```
for (auto & note:notes)              // depuis C++11
 { cout << "donnez une note " << " : " ;
   cin >> note ;
 }
```

Celles de calcul de la somme pourraient s'écrire ainsi (cette fois, le & n'est pas utile) :

```
for (auto note:notes) som += note ;
```

Remarque

Attention, cette forme d'instruction *for* adaptée aux séquences ne pourra s'appliquer que **lorsque la dimension du tableau est connue du compilateur**, ce qui ne sera plus le cas pour des tableaux transmis en argument d'une fonction. C'est une des raisons pour laquelle nous la mettrons moins en évidence dans la suite de ce paragraphe consacré aux tableaux natifs.

2.2 Les tableaux natifs à plusieurs indices

2.2.1 Leur déclaration

Comme la plupart des langages, C++ autorise les tableaux à plusieurs indices (on dit aussi à plusieurs dimensions).

Par exemple, la déclaration :

```
int t[5][3]
```

réserve un tableau de *15 (5 x 3)* éléments. Un élément quelconque de ce tableau se trouve alors repéré par deux indices comme dans ces notations :

```
t[3][2]      t[i][j]      t[i-3][i+j]
```

Notez bien que la notation désignant un élément d'un tel tableau est une *lvalue* modifiable. Il n'en ira toutefois pas de même de notations telles que *t[3]* ou *t[j]* bien que, comme nous le verrons un peu plus tard, de telles notations aient un sens en C++.

Aucune limitation ne pèse sur le nombre d'indices que peut comporter un tableau. Seules les limitations de taille mémoire liées à un environnement donné risquent de se faire sentir.

2.2.2 Arrangement en mémoire des tableaux à plusieurs indices

Les éléments d'un tableau sont rangés suivant l'ordre obtenu en faisant varier le dernier indice en premier. Ainsi, le tableau *t* déclaré précédemment verrait ses éléments ordonnés comme suit :

```
t[0][0]
t[0][1]
t[0][2]
t[1][0]
t[1][1]
t[1][2]
  ....
t[4][0]
t[4][1]
t[4][2]
```

Nous verrons que cet ordre a une incidence dans au moins trois circonstances :

- lorsque l'on omet de préciser certaines dimensions d'un tableau ;
- lorsque l'on souhaite accéder à l'aide d'un pointeur aux différents éléments d'un tableau ;
- lorsque l'un des indices « déborde ». Suivant l'indice concerné et les valeurs qu'il prend, il peut y avoir débordement d'indice sans sortie du tableau. Par exemple, toujours avec notre tableau *t* de *5* x *3* éléments, vous voyez que la notation *t[0][5]* désigne en fait l'élément *t[1][2]*. En revanche, la notation *t[5][0]* désigne un emplacement situé juste au-delà du tableau.

Remarque

Bien entendu, les différents points évoqués, dans le paragraphe 1.2, à propos des tableaux à une dimension, restent valables dans le cas des tableaux à plusieurs dimensions.

2.2.3 Parcours des éléments d'un tableau à plusieurs indices (C++11)

Si l'on a déclaré :

```
int t[5][3]
```

on pourra naturellement parcourir chacun des éléments du tableau en utilisant des instructions imbriquées, par exemple :

```
for (int i=0 ; i<5 ; i++)
   for (int j=0 ; j<3 ; j++)
     { // traitement de l'élément t[i][j] }
```

Notez qu'on peut tout à fait inverser l'ordre des instructions *for*, ce qui conduirait simplement à un autre ordre de parcours.

Depuis C++11, on peut utiliser l'instruction *for* pour les séquences, de la manière suivante :

```
for (auto & col : t)
   for (auto e : col)     // ou auto & e : col   si on doit modifier e
     { // traitement de l'élément e }
```

Cette fois, il est obligatoire de considérer le tableau *t* comme un tableau de tableaux de 3 entiers. La première instruction parcourt les 5 tableaux de 3 entiers de *t*. La transmission doit alors **obligatoirement** se faire par référence (d'où le &). Pour la seconde instruction, le & ne sera nécessaire que si les valeurs de *t* doivent être modifiées. On notera bien que, cette fois, on ne dispose plus d'aucune latitude quant à l'ordre de parcours.

Enfin, on ne perdra pas de vue que cette démarche n'est plus applicable aux tableaux reçus en argument par une fonction.

2.3 Initialisation des tableaux natifs

Comme les variables scalaires, les tableaux peuvent être, suivant leur déclaration, de classe statique ou automatique. Les tableaux de classe statique sont, par défaut, initialisés à « zéro »[3] ; les tableaux de classe automatique ne sont pas initialisés implicitement.

Il est possible, comme on le fait pour une variable scalaire, d'initialiser (partiellement ou totalement) un tableau lors de sa déclaration. En voici d'abord quelques exemples.

2.3.1 Initialisation de tableaux natifs à un indice

La déclaration :

```
int tab[5] = { 10, 20, 5, 0, 3 } ;  // on peut omettre = depuis C++11
```

3. Rappelons que « zéro » est un terme générique pouvant représenter différentes choses suivant le type concerné : valeur entière 0, valeur flottante 0, pointeur nul, appel du constructeur par défaut pour un objet...

place les valeurs *10*, *20*, *5*, *0* et *3* dans chacun des cinq éléments du tableau *tab*.

Il est possible de ne mentionner dans les accolades que les premières valeurs, comme dans ces exemples :

```
int tab[5] = { 10, 20 } ;      // on peut omettre = depuis C++11
int tab[5] = { 10, 20, 5 } ;   // on peut omettre = depuis C++11
```

Les valeurs manquantes seront, suivant la classe d'allocation du tableau, initialisées à zéro (statique) ou aléatoires (automatique).

De plus, il est possible d'omettre la dimension du tableau, celle-ci étant alors déterminée par le compilateur par le nombre de valeurs énumérées dans l'initialisation. Ainsi, la première déclaration de ce paragraphe est équivalente à :

```
int tab[] = { 10, 20, 5, 0, 3 } ;
```

On peut déclarer un tableau constant et l'initialiser comme dans :

```
const char voyelles [] = {'a', 'e', 'i', 'o', 'u', 'y'} ;
```

Bien entendu, toute tentative ultérieure de modification du tableau sera rejetée :

```
voyelles [2] = 'u' ;      // interdit
```

Remarques

1 La notation {} est obligatoire ici.

2 Nous verrons (paragraphe 4.3.1) qu'il existe un cas particulier d'initialisation d'un tableau de caractères par une chaîne littérale.

3 Contrairement aux vecteurs, les tableaus natifs ne sont pas des objets, ce qui explique que leurs initialisations soient différentes.

2.3.2 Initialisation de tableaux natifs à plusieurs indices

Voyez ces deux exemples équivalents (nous avons volontairement choisi des valeurs consécutives pour qu'il soit plus facile de comparer les deux formulations) :

```
int tab [3] [4] = { { 1, 2, 3, 4 },
                    { 5, 6, 7, 8 },
                    { 9,10,11,12 }  }
int tab [3] [4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 } ;
```

La première forme revient à considérer notre tableau comme composé de trois tableaux de quatre éléments chacun. La seconde exploite la manière dont les éléments sont effectivement rangés en mémoire, et elle se contente d'énumérer les valeurs du tableau suivant cet ordre.

Là encore, à chacun des deux niveaux, les dernières valeurs peuvent être omises. Les déclarations suivantes sont correctes (mais non équivalentes) :

```
int tab [3] [4] = { { 1, 2 } , { 3, 4, 5 } } ;
int tab [3] [4] = { 1, 2 , 3, 4, 5 } ;
```

2.3.3 Initialiseurs et classe d'allocation

Les initialiseurs utilisables pour les éléments d'un tableau suivent les mêmes règles que les initialiseurs des variables scalaires, à savoir :

- Pour un tableau statique, les valeurs d'initialisation doivent être des expressions constantes, calculables à la compilation, d'un type compatible par affectation avec le type des éléments du tableau ; on peut y faire apparaître des variables statiques ou des variables automatiques déclarées avec l'attribut *const* ; en voici un exemple :

```
void f()
{ const int N = 10 ;       // ou constexpr depuis C++11
  static int delta = 3 ;
     .....
  int tab[5] = { 2*N-1, N-1, N, N+1, 2*N+1} ;
  int t[3] = { 0, delta, 2*delta } ;
}
```

- Pour un tableau automatique, on peut utiliser n'importe quelle expression d'un type compatible par affectation avec le type des éléments du tableau. En voici un exemple d'école :

```
const int NEL = 10 ;
void fct (int p)
{ int n ;
     .....
  int tab[] = {NEL, p, 2*p, n+1, n+p} ;
  .....
}
```

2.4 L'équivalence entre tableau natif et pointeur

En C++, l'identificateur d'un tableau, lorsqu'il est employé seul (sans indices à sa suite), est considéré comme un pointeur (constant) sur le début du tableau. Nous allons en examiner les conséquences en commençant par le cas des tableaux à un indice. Mais auparavant, il nous faut apporter quelques compléments d'information sur les opérations applicables aux pointeurs (incrémentation et comparaison).

2.4.1 Incrémentation de pointeurs

Dans le chapitre consacré aux pointeurs, nous nous sommes contentés de manipuler, non pas les variables pointeurs elles-mêmes, mais les valeurs pointées. Or, si une variable pointeur *ad* a été déclarée ainsi :

```
int * ad ;
```

une expression telle que :

```
ad + 1
```

a un sens pour C++.

En effet, *ad* est censée contenir l'adresse d'un entier et, pour C++, l'expression ci-dessus représente **l'adresse de l'entier suivant**. Certes, dans notre exemple, cela n'a guère d'intérêt,

car nous ne savons pas avec certitude ce qui se trouve à cet endroit. Mais cela va s'avérer utile dans le traitement de tableaux natifs.

Notez bien qu'il ne faut pas confondre un pointeur avec un nombre entier. En effet, l'expression précédente ne représente pas l'adresse de *ad* augmentée de *1* (octet). Plus précisément, la différence entre *ad+1* et *ad* est ici de *sizeof(int)* octets (l'opérateur *sizeof* fournit la taille, en octets, d'un type donné). Si *ad* avait été déclarée par :

```
double * ad ;
```

cette différence serait de *sizeof(double)* octets.

De manière comparable, l'expression :

```
ad++
```

incrémente donc l'adresse contenue dans *ad* de manière qu'elle désigne **l'élément** suivant.

Notez bien que des expressions telles que *ad+1* ou *ad++* sont, en général, valides, quelle que soit l'information se trouvant réellement à l'emplacement correspondant. Par ailleurs, il est possible d'incrémenter ou de décrémenter un pointeur de n'importe quelle quantité entière. Par exemple, avec la déclaration précédente de *ad*, nous pourrions écrire ces instructions :

```
ad += 10 ;
ad -= 25 ;
```

Remarque

Il existe une exception à ces possibilités, à savoir le cas des pointeurs sur des fonctions : on comprend bien qu'incrémenter un pointeur d'une quantité correspondant à la taille d'une fonction n'a pas de sens en soi.

2.4.2 Comparaison et soustraction de pointeurs

Nous avons déjà vu que l'on pouvait appliquer les opérateurs == et != aux pointeurs natifs. Mais il est possible de les comparer avec les opérateurs <, >, <= ou >=. Dans ce cas, la comparaison porte tout simplement sur les adresses. Si cela n'a guère de sens dans le cas de variables simples, nous verrons que cela peut avoir un intérêt dans le cas où les deux pointeurs désignent des éléments d'un même tableau.

Enfin, il est théoriquement possible d'efffectuer la différence entre deux pointeurs de même type, ce qui fournit le nombre d'éléments du type en question, situés entre les deux adresses correspondantes. Cette possibilité est rarement utilisée.

2.4.3 Équivalence tableau à un indice et pointeur

Supposons que l'on effectue la déclaration suivante :

```
int t[10] ;
```

La notation *t* est alors totalement équivalente à *&t[0]*.

L'identificateur *t* est considéré comme étant de type « pointeur sur le type correspondant aux éléments du tableau », c'est-à-dire, ici, *int* * (et même plus précisément *const int* *). Ainsi, voici quelques exemples de notations équivalentes :

```
t+1                 &t[1]
t+i                 &t[i]
t[i]                * (t+i)
```

Pour illustrer ces nouvelles possibilités de notation, voici deux façons de placer la valeur *1* dans chacun des 10 éléments de notre tableau *t* :

```
for (int i=0 ; i<10 ; i++)
  *(t+i) = 1 ;

int i ;
int *p :
for (p=t, i=0 ; i<10 ; i++, p++)
  *p = 1 ;
```

Dans la seconde façon, nous avons dû recopier la valeur représentée par *t* dans un pointeur nommé *p*. En effet, il ne faut pas perdre de vue que le symbole *t* représente une adresse constante (*t* est une constante de type pointeur sur des entiers). Autrement dit, une expression telle que *t++* aurait été invalide, au même titre que, par exemple, *3++*. **Un nom de tableau est un pointeur constant ; ce n'est pas une *lvalue* modifiable**.

Nous pouvons également utiliser la possibilité de comparer des pointeurs :

```
int t[10] ;
for (int *p=t ; p<t+10 ; p++)
   *p = 1 ;
```

Remarques

1 Les trois formulations précédentes sont équivalentes (en C++11) à :

```
for (auto & v : t) v = 1 ;
```

2 Nous venons de voir que la notation *t[i]* est équivalente à **(t+i)* lorsque *t* est déclaré comme un tableau. En fait, cela reste vrai, quelle que soit la manière dont *t* a été déclaré. Ainsi, avec :

```
int *t ;
```

les deux notations précédentes resteraient équivalentes. Autrement dit, on peut utiliser *t[i]* dans un programme où *t* est simplement déclaré comme un pointeur (encore faudra-t-il, toutefois, disposer à cette adresse de l'espace mémoire nécessaire).

2.5 Équivalence tableau à plusieurs indices et pointeur

Comme pour les tableaux à un indice, l'identificateur d'un tableau, employé seul, représente toujours son adresse de début. Toutefois, si l'on s'intéresse à son type exact, il ne s'agit plus d'un pointeur sur des éléments du tableau. En pratique, ce point n'a d'importance que lorsque l'on effectue des calculs arithmétiques avec ce pointeur (ce qui est assez rare) ou lorsque l'on doit transmettre ce pointeur en argument d'une fonction ; dans ce dernier cas, cependant, nous verrons que le problème est automatiquement résolu par la mise en place de conversions, de sorte qu'on peut ne pas s'en préoccuper.

À simple titre indicatif, nous vous présentons ici les règles employées par C++, en nous limitant au cas de tableaux à deux indices.

Lorsque le compilateur rencontre une déclaration telle que :

```
int t[3] [4] ;
```

il considère en fait que *t* désigne un tableau de 3 éléments, chacun de ces éléments étant lui-même un tableau de 4 entiers. Autrement dit, si *t* représente bien l'adresse de début de notre tableau *t*, il n'est plus de type *int* * (comme c'était le cas pour un tableau à un indice) mais d'un type « pointeur sur des blocs de 4 entiers », type qui devrait se noter théoriquement (vous n'aurez probablement jamais à utiliser cette notation) :

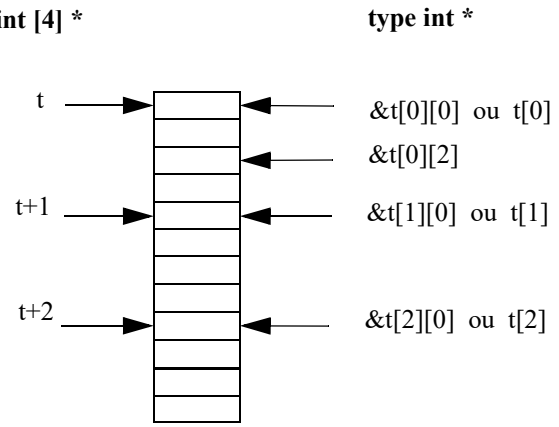
```
int [4] *
```

Dans ces conditions, une expression telle que *t+1* correspond à l'adresse de *t*, augmentée de 4 entiers (et non plus d'un seul !). Ainsi, les notations *t* et *&t[0][0]* correspondent toujours à la même adresse, mais l'incrémentation de *1* n'a pas la même signification pour les deux.

Par ailleurs, les notations telles que *t[0]*, *t[1]* ou *t[i]* ont un sens. Par exemple, *t[0]* représente l'adresse de début du premier bloc (de 4 entiers) de *t*, *t[1]*, celle du second bloc... Cette fois, il s'agit bien de pointeurs de type *int **. Autrement dit, les notations suivantes sont totalement équivalentes (elles correspondent à la même adresse et elles sont de même type) :

```
t[0]    &t[0][0]
t[1]    &t[1][0]
```

Voici un schéma illustrant ce que nous venons de dire :



Remarque

t[1] est une constante ; ce n'est pas une *lvalue* modifiable. L'expression *t[1]++* est invalide. En revanche, *t[1][2]* est bien une *lvalue* modifiable.

Informations complémentaires

La notion de référence à un tableau n'a pas de sens en C++. Une déclaration telle que *int & t[10]* est interdite (elle représenterait en théorie un tableau de références, ce qui n'est pas accepté par C++). Quant à la notation *int * & t*, elle représente la référence à un pointeur sur un tableau d'entiers et elle est correcte.

2.6 Transmission de tableaux natifs en argument

Nous avons vu que le type *vector* pouvait être transmis par valeur (cas par défaut) ou par adresse (référence ou pointeur). En revanche, lorsque l'on place le nom d'un tableau en argument effectif de l'appel d'une fonction, on transmet obligatoirement (la valeur de) l'adresse du tableau à la fonction, ce qui lui permet d'effectuer toutes les manipulations voulues sur ses éléments, qu'il s'agisse d'utiliser leur valeur ou de la modifier. Tout se passe donc comme si l'on avait affaire à une transmission par pointeur et il n'existe aucun autre mode. En outre, contrairement à ce qui se passait pour les vecteurs, si la fonction à laquelle on transmet un tableau a besoin d'en connaître la dimension, cette dernière devra également lui être transmise. Voyons quelques exemples pratiques.

2.6.1 Cas des tableaux à un indice

a) Premier exemple : tableau de taille fixe

Voici un exemple de fonction qui met la valeur *1* dans tous les éléments d'un tableau de 10 éléments, l'adresse de ce tableau étant transmise en argument.

```
void fct (int t[10])
{ for (int i=0 ; i<10 ; i++) t[i]=1 ;
    // for (int & n : t) n=1  serait refuse - voir remarque
}
```

Exemple de tableau à un indice transmis en argument d'une fonction

Voici deux exemples d'appels possibles de cette fonction :

```
int t1[10], t2[10] :
  .....
fct(t1) ;
  ....
fct(t2) ;
```

L'en-tête de *fct* peut être indifféremment écrit de l'une des manières suivantes :

```
void fct (int t[10])
void fct (int * t)
void fct (int t[])
```

La dernière écriture se justifie par le fait que *t* désigne un argument muet. La réservation de l'emplacement mémoire du tableau dont on recevra ici l'adresse est réalisée par ailleurs dans

la fonction appelante (de plus cette adresse peut changer d'un appel au suivant). De plus, la connaissance de la taille exacte du tableau n'est pas indispensable au compilateur ; il est en effet capable de déterminer l'adresse d'un élément quelconque, à partir de son rang et de l'adresse de début du tableau (nous verrons qu'il n'en ira plus de même pour les tableaux à plusieurs indices). Dans ces conditions, on comprend qu'il soit tout à fait possible de ne pas mentionner la dimension du tableau dans l'en-tête de la fonction. En fait, le *10* qui figure dans le premier en-tête n'a d'intérêt que pour le lecteur du programme, afin de lui rappeler la dimension effective du tableau sur lequel travaillait notre fonction.

Par ailleurs, comme d'habitude, **quel que soit l'en-tête employé**, on peut, dans la définition de la fonction, utiliser indifféremment le formalisme tableau ou le formalisme pointeur. Voici plusieurs écritures possibles de *fct* qui s'accommodent de n'importe lequel des trois en-têtes précédents :

```
for (int i=0 ; i<10 ; i++)      t[i] = 1 ;
for (int i=0 ; i<10 ; i++, t++) *t = 1 ;
for (int i=0 ; i<10 ; i++)      *(t+i) = 1 ;
for (int i=0 ; i<10 ; i++)      t[i] = 1 ;
```

Ici encore, l'expression *t++* ne pose aucun problème car *t* représente une copie de l'adresse d'un tableau ; *t* est donc bien une *lvalue* et elle peut donc être incrémentée.

Voici enfin une dernière possibilité dans laquelle nous recopions l'adresse *t* dans un pointeur *p*, et où nous utilisons les possibilités de comparaison de pointeurs :

```
for (int * p = t ; p<t+10 ; p++) *p = 1 ;
```

Remarques

1 Notez bien que, quelle que soit la forme d'en-tête utilisée, il n'est pas possible d'employer l'instruction *for* généralisée aux séquences en écrivant :

```
for (int & n : t) n=1 ;    // erreur
```

Ceci est dû à ce que l'information de dimension de *t* n'est pas transmise à *f*.

2 N'oubliez pas que si vous définissiez *fct* avec l'un de ces en-têtes :

```
void fct (const int * t)
void fct (const int t[])
```

t serait un pointeur sur des entiers constants. Il ne serait alors plus possible dans *fct* de modifier les valeurs du tableau reçu en argument.

3 Vous pouvez penser à utiliser pour *fct* l'en-tête :

```
void fct (int * const t)
```

Dans ce cas, *t* est un pointeur constant sur des entiers. Ce n'est donc que la valeur de *t* qui ne peut pas être modifiée, alors que les entiers du tableau peuvent toujours l'être. Cette possibilité a généralement peu d'intérêt : elle interdit d'incrémenter directement la valeur de *t* dans *fct*, laquelle n'est, de toute façon, qu'une copie de la valeur de l'argument effectif...

b) Second exemple : tableau de taille variable

Comme nous venons de le voir, lorsqu'un tableau à un seul indice apparaît en argument d'une fonction, le compilateur n'a pas besoin d'en connaître la taille exacte. Il est ainsi facile de réaliser une fonction capable de travailler avec un tableau de dimension quelconque, à condition de lui en transmettre la taille en argument. Voici, par exemple, une fonction qui calcule la somme des éléments d'un tableau d'entiers de taille quelconque :

```
int som (int t[],int  nb)
{  int s = 0 ;
   for (int i=0 ; i<nb ; i++)
         s += t[i] ;
   return (s) ;
}
```

Fonction travaillant sur un tableau à une dimension de taille variable

Voici quelques exemples d'appels de cette fonction :

```
int main()
{  int t1[30], t2[15], t3[10] ;
   int s1, s2, s3 ;
      .....
   s1 = som(t1, 30) ;
   s2 = som(t2, 15) + som(t3, 10) ;
    .....
}
```

2.6.2 Cas des tableaux à plusieurs indices

a) Premier exemple : tableau de taille fixe

Voici un exemple d'une fonction qui place la valeur *1* dans chacun des éléments d'un tableau de dimensions *10* et *15* :

```
void raun (int t[10][15])
{  for (int i=0 ; i<10 ; i++)
      for (int j=0 ; j<15 ; j++)
         t[i][j] = 1 ;
}
```

Exemple de transmission en argument d'un tableau à deux dimensions (fixes)

Ici, on pourrait, par analogie avec ce que nous avons dit pour un tableau à un indice, utiliser d'autres formes de l'en-tête. Toutefois, il faut bien voir que, pour trouver l'adresse d'un élément quelconque d'un tableau à deux indices, le compilateur ne peut plus se contenter de connaître son adresse de début ; il doit également connaître la seconde dimension du tableau (la première n'étant pas nécessaire compte tenu de la manière dont les éléments sont disposés

en mémoire : revoyez le paragraphe 2.6.1). Ainsi, l'en-tête de notre fonction aurait pu être *rau (int t[][15])* mais pas *rau (int t[][])*.

En revanche, cette fois, quel que soit l'en-tête utilisé, cette fonction ne convient plus pour un tableau de dimensions différentes de celles pour lesquelles elle a été prévue. Plus précisément, nous pourrons certes toujours l'appeler, comme dans cet exemple :

```
int mat [12][20] ;
    .....
raun (mat) ;
    .....
```

Mais, bien qu'aucun diagnostic ne nous soit fourni par le compilateur, l'exécution de ces instructions placera 150 fois la valeur *1* dans certains des 240 emplacements de *mat*. Qui plus est, avec des tableaux dont la deuxième dimension est inférieure à 15, notre fonction placerait des *1*... en dehors de l'espace attribué au tableau !

Remarque

On pourrait songer, par analogie avec ce qui a été fait pour les tableaux à un indice, à mélanger le formalisme pointeur et le formalisme tableau, à la fois dans l'en-tête et dans la définition de la fonction ; cela pose toutefois quelques problèmes que nous allons évoquer dans l'exemple suivant consacré à un tableau de dimensions variables (et dans lequel le formalisme précédent n'est plus applicable).

b) Second exemple : tableau de dimensions variables

Supposons que nous cherchions à écrire une fonction qui place la valeur 0 dans chacun des éléments de la diagonale d'un tableau carré de taille quelconque. Une façon de résoudre ce problème consiste à adresser les éléments voulus par des pointeurs en effectuant le calcul d'adresse approprié.

```
void diag (int * p, int n)
{ for (int i=0 ; i<n ; i++)
   { * p = 0 ;
     p += n+1 ;
   }
}
```

Fonction travaillant sur un tableau carré de taille variable

Notre fonction reçoit donc, en premier argument, l'adresse du premier élément du tableau, sous forme d'un pointeur de type *int **. Ici, nous avons tenu compte de ce que deux éléments consécutifs de la diagonale sont séparés par *n* éléments. Ainsi, si un pointeur désigne un élément de la diagonale, pour pointer sur le suivant il suffit d'incrémenter ce pointeur de *n+1* unités (l'unité étant ici la taille d'un entier).

Remarques

1 Un appel de notre fonction *diag* se présentera ainsi :

```
int t[30] [30] ;
diag (t, 30)
```

Or l'argument effectif *t* est, certes, l'adresse de *t*, mais d'un type pointeur sur des blocs de 10 entiers et non pointeur sur des entiers. En fait, la présence d'un prototype pour *diag* fera qu'il sera converti en un *int* *. Ici, il n'y a aucun risque de modification d'adresse liée à des contraintes d'alignement, car on passe de l'adresse d'un objet de taille *10n* à l'adresse d'un objet de taille *n*. Il n'en irait pas de même avec la conversion inverse.

2 Cette fonction pourrait également s'écrire en y déclarant un tableau à une seule dimension dont la taille (*n*n*) devrait alors être fournie en argument (en plus de *n*). Le même mécanisme d'incrémentation de *n+1* s'appliquerait alors, non plus à un pointeur, mais à la valeur d'un indice.

2.7 Les tableaux dynamiques

Nous avons déjà appris à allouer et à libérer dynamiquement (dans le tas) des emplacements pour des valeurs de type simple, soit en utilisant des pointeurs natifs, soit à l'aide de pointeurs intelligents.

Ces possibilités s'appliquent aux tableaux natifs, moyennant une adaptation des opérateurs *new* et *delete* qui deviennent les opérateurs *new[]* et *delete[]*. Avec C++14, on peut également utiliser *make_unique*.

C'est ce que nous allons examiner ici, sachant que ces connaissances seront surtout utiles pour exploiter d'anciens codes ; en effet, la plupart du temps, les types *string* et *vector* offriront des fonctionnalités comparables, de façon plus fiable.

2.7.1 Avec des pointeurs natifs et les opérateurs *new[]* et *delete[]*

L'opérateur *new[]* accepte une syntaxe de la forme :

new type [n]

où *n* désigne une expression entière quelconque (non négative). Cette instruction alloue alors l'emplacement nécessaire pour *n* éléments du *type* indiqué ; si l'opération a réussi, elle fournit en résultat un pointeur (toujours de type *type* *) sur le premier élément de ce tableau. Le comportement en cas de manque de mémoire est le même que celui de *new*, présenté au paragraphe 1.3 du chapitre 10.

Par exemple, avec :

```
int n ;
  .....
int * ad = new int[n] ;   //  ou auto ad = new int[n] ; depuis C++11
```

on allouera un emplacement pour un tableau de *n* éléments de type *int* et on placera son adresse (c'est-à-dire l'adresse du premier élément) dans *ad*.

L'accès aux différents éléments de ce tableau pourra se faire avec l'opérateur [], compte tenu de l'équivalence entre nom de tableau et pointeur natif :

```
for (int i=0 ; i<n ; i++)
{ // traitement de ad[i] (lvalue modifiable)
}
```

En revanche, il n'est pas possible ici d'utiliser l'instruction *for* généralisée aux séquences :

```
for (auto b : ad) {.....}   // n'a pas de sens ici
```

L'emplacement ainsi alloué devra être libéré en utilisant l'opérateur *delete[]* et non simplement *delete* qui ne libèrerait que le premier élément...

2.7.2 Avec le type *unique_ptr* (C++11)

Le type *unique_ptr* permet de manipuler des tableaux dynamiques (ce n'est pas le cas du type *shared_ptr*). L'exemple précédent pourra s'écrire :

```
unique_ptr<int[]> adu ( new int[n]) ;
  // attention, avec auto, on obtiendrait le type int * et non int[] *
```

Cette fois, le recours à *delete* est inutile (et interdit) : le tableau désigné par *adu* sera libéré quand *adu* sera détruit.

De plus, depuis C++14, la fonction *make_unique* peut s'appliquer à des tableaux, de sorte que l'instruction précédente peut être avantageusement remplacée par :

```
unique_ptr<int[]> adu = make_unique<int[]> (n)  ;
```

ou, mieux :

```
auto adu = make_unique<int[]> (n) ;  // ou : auto adu (make_unique<int[]>(n)) ;
```

Notez que, si l'accès au tableau peut toujours se faire avec l'opérateur [], comme dans *adu[i]*, i**l n'est plus possible d'employer l'équivalence tableau/pointeur natif** en utilisant à la place **(adu + i)*.

2.7.3 Exemple

Voici un exemple d'école reprenant les diverses possibilités évoquées.

```
// GestDynTableauxNatifs
#include <iostream>
#include <memory>   // pour unique_ptr
using namespace std ;
int main()
{ cout << "Combien de valeurs : " ;
  int nb ; cin >> nb ;
  //----------- allocation C++03 d'un emplacement pour nb entiers ------------
  //           dans lequel on place les carrres des nombres 1 à nb
  int * adi = new int [nb] ;
  cout << "Allocation de " << nb << " int en      : " << adi << endl ;
  for (int i=0 ; i<nb ; i++) adi[i]=(i+1)*(i+1) ; // ou *(adi+i) au lieu de adi[i]
```

```
  cout << "Voici les carres des nombres de 1 a " << nb << " : \n" ;
  for (int i=0 ; i<nb ; i++) cout << adi[i] << " " ;
    // ou : for (int *adibis=adi ; adibis<adi+nb ; adibis++) cout << *adibis << " " ;
  cout << endl ;
  delete [] adi ;   // liberation des nb entiers
  cout << "Liberation des " << nb << " entiers" << endl ;
    //----------------- Le même tableau en C++11 -----------------------------
  unique_ptr<int[]> upi (new int [nb]) ;
    // auto upi = make_unique<int[]>(nb) ;    C++14 uniquement
  cout << "Allocation de nb entiers en : " << upi.get() << "\n" ;
  for (int i = 0 ; i<nb ; i++) upi[i] = (i+1)*(i+1) ;
    // *(upi+i) = (i+1) * (i+1)         ne fonctionnerait pas ici
    // *(upi.get()+i) = (i+1) * (i+1)   fonctionnerait - mais deconseille
  cout << "Voici les carres des nombres de 1 a " << nb << " : \n" ;
  for (int i=0 ; i<nb ; i++) cout << adi[i] << " " ;
    // le tableau sera libere lorsque upi sera detruit, ici en fin de main
}
```

```
Combien de valeurs : 9
Allocation de 9 int en      : 0x170e40
Voici les carres des nombres de 1 a 9 :
1 4 9 16 25 36 49 64 81
Liberation des 9 entiers
Allocation de nb entiers en : 0x170e40
Voici les carres des nombres de 1 a 9 :
1 4 9 16 25 36 49 64 81
```

Allocation dynamique de tableaux natifs avec pointeurs natifs ou avec unique_ptr

2.7.4 Initialisation de tableaux dynamiques

Nous avons vu comment on peut initialiser les tableaux natifs (statiques ou automatiques) lors de leur déclaration. Les choses sont plus restrictives dans le cas des tableaux dynamiques.

Tout d'abord, lorsque l'on utilise l'opérateur *new*, comme nous l'avons fait précédemment, sous la forme *new T[n]*, sans autres paramètres (aussi bien avec un pointeur natif qu'un *unique_ptr*), les éléments du tableau reçoivent l'initialisation par défaut prévue pour le type *T*, à savoir : pas d'initialisation pour les types de base, y compris les pointeurs natifs, chaîne vide pour *string*, vecteur vide pour *vector*...

Si l'on utilise une syntaxe de la forme *new T[n] ()*, les éléments sont initialisés à « zéro » (terme générique représentant la valeur 0 pour des types numériques, *nullptr* ou 0 pour un pointeur, mais la chaîne vide pour *string*...).

Depuis C++11, on peut spécifier, entre accolades, des valeurs initiales sous forme d'expression d'un type compatible avec *T*, comme dans :

```
int * ad  = new int [5] {2, 4, 6, 8, 10}  ;
unique_ptr<int[]> up1 (new int [5] {2, 4, 6, 8, 10}) ;
```

En revanche, il n'existe aucune possibilité comparable lorsque l'on utilise *make_unique*.

3 Les vecteurs multidimensionnels

***N.B**. Ce paragraphe peut être ignoré dans un premier temps.*

Nous avons vu comment réaliser des tableaux natifs à plusieurs dimensions. Le type *vector* peut, lui aussi, se « composer » pour aboutir à une situation semblable, mais non identique. Considérons cette déclaration d'un tableau natif qui correspond à un tableau de 3 tableaux de 4 éléments de type *int* :

```
int t[3] [4] ;
```

Nous pouvons, par analogie, chercher à utiliser :

```
vector<vector<int>> v(3)
```

qui correspond à un vecteur de 3 éléments de type *vector<int>*.

Mais, pour l'instant ces éléments sont vides. Pour leur attribuer une valeur, il faut faire appel aux possibilités d'extension dynamiques des vecteurs (fonction *pushback*). Il est tout à fait possible que chacun de ces éléments possède une dimension différente... En outre, chacun des 3 éléments de notre vecteur *v*, c'est-à-dire chacun des 3 vecteurs est alloué de façon indépendante ; ils ne seront plus contigüs comme le sont ceux d'un tableau natif.

Depuis C++11, il est possible d'initialiser un vecteur dans sa déclaration, de sorte qu'il est possible de définir :

```
vector<vector<int>> v = { {1, 2, 3, 4}, {2, 3, 4, 5}, {6, 7, 8, 9} } ;
```

On obtiendra ici un vecteur de 3 vecteurs de 4 éléments. Mais, il est tout aussi possible de définir

```
vector<vector<int>> v = { {1, 2}, {2, 3, 4, 5}, {6, 7, 8} } ;
```

qui conduit à un vecteur de 3 éléments comportant chacun respectivement 2, 4 et 3 éléments.

Voici un petit exemple d'utilisation d'un vecteur bidimensionnel :

```
// Vector2D
#include <iostream>
#include <vector>
using namespace std;
int main()
{ vector<vector<int>> v = { {1, 2}, {2, 3, 4, 5}, {6, 7, 8} } ;   // C++11
  for (auto l : v)
  { for (auto e : l) cout << e << " " ;
    cout << endl ;
  }
}
```

```
1 2
2 3 4 5
6 7 8
```

Vecteur bidimensionnel

4 Les chaînes de style C

Depuis C++98, nous pouvons manipuler des chaînes de caractères en utilisant le type *string* dont nous avons présenté les propriétés les plus importantes au chapitre 11. Auparavant, C++ représentait les chaînes d'une manière artificielle consistant à les placer dans des tableaux de caractères et à marquer leur fin par un caractère de code nul (il n'existait donc aucune information de longueur !). Pour des raisons historiques, cette convention ne peut pas être totalement ignorée du C++ moderne puisqu'elle reste utilisée :

- par le compilateur pour représenter les chaînes littérales (telles que "bonjour"), ce qui a une incidence en cas de « mélange » entre chaînes littérales et valeurs de type *string* ;
- pour les éventuels arguments que l'on peut transmettre à la fonction *main*.

Ce sont donc les propriétés de ces pseudo-chaînes (que nous nommerons chaînes de style C) que nous allons étudier ici. Nous en profiterons pour apporter quelques informations générales concernant les fonctions (héritées du C) manipulant ces chaînes, afin de vous faciliter l'éventuelle utilisation d'anciens codes.

4.1 Représentation des chaînes de style C

Une chaîne de style C est représentée par une suite d'octets correspondant à chacun de ses caractères (plus précisément à chacun de leurs codes), le tout étant terminé par un octet supplémentaire de code nul. Cela signifie que, d'une manière générale, une chaîne de *n* caractères occupe en mémoire un emplacement de *n+1* octets.

C'est cette convention qu'utilise le compilateur pour représenter les chaînes littérales écrites dans vos programmes sous la forme :

```
"bonjour"
```

De plus, une telle notation sera traduite par le compilateur en un pointeur de type *char* * sur le premier caractère de la chaîne.

Voici un programme illustrant ces deux particularités :

```
#include <iostream>
using namespace std ;
int main()
{ char * adr = "bonjour" ;
  while (*adr) { cout << *adr ;
                 adr++ ;
               }
}
```

```
bonjour
```

Convention de représentation des chaînes

La déclaration :

```
char *adr ;
```

réserve simplement l'emplacement pour un pointeur sur un caractère (ou une suite de caractères). En ce qui concerne la constante :

```
"bonjour"
```

le compilateur a créé en mémoire la suite d'octets correspondants mais, dans l'affectation :

```
adr = "bonjour"
```

la notation *bonjour* a comme valeur, non pas la valeur de la chaîne elle-même, mais son adresse ; on retrouve là le même phénomène que pour les tableaux.

La boucle *while* permet ensuite de parcourir chacun des caracètres de la chaîne.

4.2 Lecture et écriture de chaînes de style C

Nous avons déjà vu comment lire au clavier et afficher à l'écran des valeurs des différents types de base et du type *string*. Dans le cas des chaînes de style C, il n'y a pas de type spécifique. Il faut prévoir un emplacement pour « héberger » ces chaînes. Un tableau de caractères (dont le nom est, rappelons-le un pointeur constant de type *char* *) convient parfaitement. Voyez cet exemple dans lequel les chaînes C lues au clavier sont rangées dans les tableaux natifs *nom*, *prenom* et *ville* :

```
// LectureChainesC
#include <iostream>
using namespace std ;
int main()
{  char nom [20], prenom [20], ville [25] ;
   cout << "quelle est votre ville : " ;
   cin >> ville ;
   cout << "donnez votre nom et votre prenom : " ;
   cin >> nom >> prenom ;
   cout << "bonjour cher " << prenom << " "<< nom <<  " qui habitez a " << ville ;
}
```

```
quelle est votre ville : Paris
donnez votre nom et votre prenom : Dupont Yves
bonjour cher Yves Dupont qui habitez a Paris
```

Lecture et écriture de chaînes de style C

Remarques

1 Rappelons que les informations lues sur *cin* sont délimitées par des caractères séparateurs. Cette remarque vaut pour les chaînes de style C et il n'est donc pas possible de lire une chaîne renfermant un espace ou une fin de ligne. Nous verrons comment y parvenir au paragraphe 7.2 du chapitre 30.

2 Notez bien que la lecture de *n* caractères implique le stockage en mémoire de *n+1* caractères. Par exemple, ici, le nom fourni par l'utilisateur ne doit pas contenir plus de 19 caractères. Dans la pratique, pour éviter tout débordement du tableau accueillant la chaîne, on pourrait limiter la taille des informations lues sur le flot en recourant au « manipulateur paramétrique » *setw* qui sera étudié dans le chapitre consacré aux flots.

3 Jusqu'ici, nous avions affiché intuitivement la valeur de chaînes constantes dans des instructions du genre :

```
cout << "bonjour" ;
```

À la compilation, un emplacement est réservé pour la chaîne *"bonjour"*. Lors de l'exécution, son adresse est transmise à l'opérateur << qui envoie sur le flot *cout* tous les caractères trouvés à partir de cette adresse, jusqu'à la rencontre d'un caractère de code nul.

4.3 Initialisation de tableaux natifs par des chaînes de style C

4.3.1 Initialisation de tableaux de caractères

Nous venons de voir comment placer des chaînes de style C dans des tableaux de caractères. Toutefois, si vous déclarez par exemple :

```
char ch[20] ;
```

vous ne pourrez pas pour autant transférer une chaîne constante dans *ch*, en écrivant une affectation du genre :

```
ch = "bonjour" ;
```

En effet, *ch* est une constante pointeur qui correspond à l'adresse que le compilateur a attribuée au tableau *ch* ; ce n'est pas une *lvalue* ; il n'est donc pas question de lui attribuer une autre valeur (ici, il s'agirait de l'adresse attribuée par le compilateur à la constante chaîne *"bonjour"*).

En revanche, C++ vous autorise à initialiser votre tableau de caractères à l'aide d'une chaîne constante. Ainsi, vous pourrez écrire :

```
char ch[20] = "bonjour" ;
```

Cela sera parfaitement équivalent à une initialisation de *ch* réalisée par une énumération de caractères (en n'omettant pas le code zéro – noté *\0*) :

```
char ch[20] = { 'b','o','n','j','o','u','r','\0' }
```

N'oubliez pas que, dans ce dernier cas, les 12 caractères non initialisés explicitement seront :

- soit initialisés à zéro (pour un tableau de classe statique) : on voit que, dans ce cas, l'omission du caractère *\0* ne serait (ici) pas grave (sauf si l'on avait fourni 20 caractères !) ;
- soit aléatoires (pour un tableau de classe automatique) : dans ce cas, l'omission du caractère \0 serait nettement plus gênante.

De plus, comme C++ autorise l'omission de la dimension d'un tableau lors de sa déclaration, lorsqu'elle est accompagnée d'une initialisation, il est possible d'écrire une instruction telle que :

```
char message[] = "bonjour" ;
```

Celle-ci réserve un tableau, nommé *message*, de **8 caractères** (compte tenu du *0* de fin).

Remarques

1 Nous avons vu que la déclaration :

```
auto ch = "bonjour" ;
```

pouvait poser problème car *ch* est en fait un pointeur de type *char* * qui reçoit l'adresse de la chaîne C *"bonjour"*. Ce n'est donc pas une *lvalue* de type *string*, contrairement à ce qui se passe avec :

```
string ch = "bonjour" ;
```

2 Nous avons vu qu'il est possible de concaténer (opérateur +) une chaîne de type *string* avec une chaîne littérale. En revanche, il n'est pas possible de concaténer deux littéraux car une expression telle que *"bonjour"* + *"monsieur"* correspondrait à l'addition de deux pointeurs de type *char* * et elle serait donc illégale.

4.3.2 Initialisation de tableaux natifs de pointeurs natifs sur des chaînes

Nous avons vu qu'une chaîne constante était traduite par le compilateur en une adresse que l'on pouvait, par exemple, affecter à un pointeur sur une chaîne. Cela peut se généraliser à un tableau natif de pointeurs, comme dans :

```
char * jour[7] = { "lundi", "mardi", "mercredi", "jeudi",
                   "vendredi", "samedi", "dimanche" } ;
```

Cette déclaration réalise donc à la fois la création des 7 chaînes littérales correspondant aux 7 jours de la semaine et l'initialisation du tableau *jour* avec les 7 adresses de ces 7 chaînes. Voici un exemple employant cette déclaration :

```
// InitTabPtrChainesC
#include <iostream>
using namespace std ;
int main()
{  char * jour[7] = { "lundi", "mardi", "mercredi", "jeudi",
                      "vendredi", "samedi", "dimanche" } ;
   cout << "donnez un entier entre 1 et 7 : " ;
   int i ; cin >> i ;
   cout << "le jour numero " << i << " de la semaine est " << jour[i-1]  ;
}
```

```
donnez un entier entre 1 et 7 : 6
le jour numéro 6 de la semaine est samedi
```

Initialisation d'un tableau de pointeurs sur des chaînes de style C

4.4 Les arguments transmis à la fonction main

La fonction *main* peut récupérer les valeurs des arguments fournis au programme lors de son lancement. Le mécanisme utilisé par l'utilisateur pour fournir ces informations dépend de l'environnement. Il peut s'agir de commandes de menus pour des environnements dits graphiques ou intégrés. Dans les environnements fonctionnant en mode texte (on parle souvent de « fenêtre de commande »), il s'agit de valeurs associées à la commande de lancement du programme (d'où le terme d'arguments de la ligne de commande encore utilisé parfois pour décrire ce mécanisme). En voici un exemple où l'on demande l'exécution du programme nommé *test*, en lui transmettant les arguments *arg1*, *arg2* et *arg3* :

```
test arg1 arg2 arg3
```

Ces paramètres sont toujours des chaînes de style C (lorsqu'ils sont fournis dans une commande de lancement du programme, ils sont séparés par des espaces). Leur transmission à la fonction *main* (réalisée par le système) se fait selon les conventions suivantes :

- le premier argument reçu par *main* sera de type *int* et il représentera le nombre total de paramètres fournis dans la ligne de commande (le nom du programme compte lui-même pour un paramètre) ;
- le second argument reçu par *main* sera l'adresse d'un tableau de pointeurs, chaque pointeur désignant la chaîne correspondant à chacun des paramètres, le premier étant le nom du programme.

Ainsi, en remplaçant l'en-tête de la fonction *main* par celui-ci :

```
int main (int nbarg, char * argv[])
```

nous obtiendrons :

- dans *nbarg*, le nombre total de paramètres ;
- à l'adresse *argv[0]*, le premier paramètre, c'est-à-dire le nom du programme (dans notre exemple précédent, il s'agirait donc de la chaîne *test*) ;
- à l'adresse *argv[1]*, le second paramètre (dans notre exemple, il s'agirait donc de la chaîne *arg1*) ;
- etc.

Voici un exemple de programme utilisant ces possibilités. Il est accompagné de trois exemples d'exécution. Nous avons supposé que notre programme se nommait *ArgLigneCommande*, et nous l'avons exécuté à trois reprises en introduisant, dans une fenêtre de commandes, les commandes suivantes (suivant les implémentations, le nom de programme affiché en résultat pourra différer quelque peu) :

```
ArgLigneCommande
ArgLigneCommande parametre
ArgLigneCommande donnes.dat sortie.txt 25 septembre 2018
```

```
// ArgLigneCommande
#include <iostream>
using namespace std ;
int main (int nbarg, char * argv[])
{  int i ;
   cout << "Nom programme : " << argv[0]<< "\n" ;
   if (nbarg>1) for (i=1 ; i<nbarg ; i++)
                cout << "argument numero " << i << " : " << argv[i] << "\n" ;
        else cout << "pas d'arguments\n" ;
}
```

```
Nom programme : D:\C++MODERNE\Projets\ArgLigneCommande\bin\Debug\ArgLigneCommande.exe
pas d'arguments
```

```
Nom programme : D:\C++MODERNE\Projets\ArgLigneCommande\bin\Debug\ArgLigneCommande.exe
argument numero 1 : parametre
```

```
Nom programme : D:\C++MODERNE\Projets\ArgLigneCommande\bin\Debug\ArgLigneCommande.exe
argument numero 1 : donnees.dat
argument numero 2 : sortie.txt
argument numero 3 : 25
argument numero 4 : septembre
argument numero 5 : 2018
```

Exemple de récupération des arguments de la ligne de commande

4.5 Généralités sur les fonctions traitant des chaînes de style C

C++ a hérité du C de nombreuses fonctions de manipulation de chaînes de style C. Comme nous l'avons vu le type *string* offre les mêmes possibilités, sous une forme beaucoup plus fiable, et il devra donc être privilégié dans l'écriture de nouveaux codes. Pour vous permettre cependant d'exploiter d'anciens codes, les fonctions manipulant des chaînes de style C sont présentées au paragraphe 3 de l'Annexe G fournie sur le site. Ici, nous vous fournissons quelques considérations générales concernant ces fonctions.

4.5.1 Ces fonctions travaillent toujours sur des adresses

La chaîne de style C ne constitue pas un type à part entière, mais simplement une convention de représentation. On ne peut donc jamais transmettre la valeur d'une chaîne, mais seulement son adresse, ou plus précisément un pointeur sur son premier caractère. Ainsi, pour comparer deux chaînes, on transmettra à la fonction concernée (par exemple, *strcmp*) deux pointeurs de type *char**.

Mieux, pour recopier une chaîne d'un emplacement à un autre, on fournira à la fonction voulue (par exemple, *strcpy*) l'adresse de la chaîne à copier et l'adresse de l'emplacement où devra se faire la copie. Encore faudra-t-il avoir prévu de disposer de suffisamment de place à cet endroit ! En effet, rien ne permet à la fonction de reconnaître qu'elle a écrit au-delà de ce

que vous vouliez. En fait, vous disposerez cependant d'une façon de vous prémunir contre de tels risques ; en effet, toutes les fonctions qui placent ainsi une information (susceptible d'être d'une longueur quelconque) à un emplacement d'adresse donnée possèdent deux variantes : l'une travaillant sans contrôle, l'autre possédant un argument supplémentaire permettant de limiter le nombre de caractères effectivement copiés à l'adresse concernée.

4.5.2 La fonction *strlen*

La fonction *strlen* fournit en résultat la longueur d'une chaîne dont on lui a transmis l'adresse en argument. Cette longueur correspond tout naturellement au nombre de caractères trouvés depuis l'adresse indiquée jusqu'au premier caractère de code nul, ce caractère n'étant pas pris en compte dans la longueur. Par exemple, l'expression :

```
strlen ("bonjour")
```

vaudra *7* ; de même, avec :

```
char * adr = "salut" ;
```

l'expression :

```
strlen (adr)
```

vaudra *5*.

4.5.3 Le cas des fonctions de concaténation

Il existe des fonctions dites de concaténation, c'est-à-dire de mise bout à bout de deux chaînes. A priori, de telles fonctions créent une nouvelle chaîne à partir de deux autres. Elles devraient donc recevoir en argument trois adresses ! En fait, ces fonctions se limitent à deux adresses en convenant arbitrairement que la chaîne résultante serait obtenue en ajoutant la seconde à la fin de la première, laquelle se trouve donc détruite en tant que chaîne (en fait, seul son *\0* de fin a disparu...). Là encore, on trouvera deux variantes dont l'une permet de limiter la longueur de la chaîne résultante.

4.6 Quelques précautions à prendre avec les chaînes de style C

Voici ici quelques compléments d'information concernant des situations moins usitées, mais dont la méconnaissance peut nuire à l'adaptation d'anciens codes.

4.6.1 Une chaîne de style C possède une vraie fin, mais pas de vrai début

Comme nous l'avons vu, il existe effectivement une convention de représentation de la fin d'une chaîne ; en revanche, rien de comparable n'est prévu pour son début. En fait, toute adresse de type *char* * peut toujours faire office d'adresse de début de chaîne.

Par exemple, avec cette déclaration :

```
char * adr = "bonjour" ;
```

une expression telle que :

```
strlen (adr+2)
```

serait acceptée : elle aurait pour valeur *5* (longueur de la chaîne commençant en *adr+2*). De même, avec :

```
char * adr = "bonjour" ;
```

l'instruction suivante sera acceptée :

```
cout << adr+10 ;    // affiche des caractères à partir de l'adresse adr+10
                    // tant qu'on n'a pas trouvé de zero de fin !
```

Mais on affichera des caractères assez peu prévisibles et le zéro de fin pourra éventuellement se situer très loin !

4.6.2 Les risques de modification des chaînes constantes

Nous avons vu que, dans une instruction telle que :

```
char * adr = "bonjour" ;
```

le compilateur remplace la notation *"bonjour"* par l'adresse d'un emplacement dans lequel il a rangé la succession de caractères voulus. Dans ces conditions, on peut se demander ce qui va se produire si l'on tente de modifier l'un de ces caractères par une banale affectation telle que :

```
*adr = 'x' ;         /* bonjour va-t-il se transformer en xonjour ? */
* (adr+2) = 'x' ;    /* bonjour va-t-il se transformer en boxjour ? */
```

A priori, la norme interdit la modification de quelque chose de constant. En pratique, beaucoup de compilateurs l'acceptent, de sorte que l'on aboutit à la modification de notre constante *bonjour* en *xonjour* ou *boxjour* !

Signalons qu'une constante chaîne apparaît également dans une instruction telle que :

```
cout << "bonjour" ;
```

Ici, on pourrait penser que sa modification n'est guère possible puisque nous n'avons pas accès à son adresse. Cependant, lorsque cette même constante (*bonjour*) apparaît en plusieurs emplacements d'un programme, certains compilateurs peuvent ne la créer qu'une fois ; dans ces conditions, la chaîne transmise au flot *cout* peut très bien se trouver modifiée par le processus décrit précédemment...

Remarque

Dans une déclaration telle que :

```
char ch[20] = "bonjour" ;
```

il n'apparaît pas de chaîne littérale, et ceci malgré la notation employée (*"..."*) laquelle, ici, n'est qu'une facilité d'écriture remplaçant l'initialisation des premiers caractères du tableau *ch*. En particulier, toute modification de l'un des éléments de *ch*, par une instruction telle que :

```
*(ch + 3) = 'x' ;
```

est parfaitement licite (nous n'avons aucune raison de vouloir que le contenu du tableau *ch* reste constant).