UNIVERSITÉ PARIS-XII — FACULTÉ DE SCIENCES ÉCONOMIQUES ET DE GESTION
Master Professionnel *Méthodes Appliquées de la Statistique et de l'Économétrie pour la Recherche, l'Analyse et le Traitement de l'Information*
Année universitaire 2006-2007 — *François LEGENDRE*
F.Legendre@univ-paris12.fr — http://fj.legendre.free.fr

Support du cours

Ce support du cours est seulement destiné à constituer une aide en cours. L'ouvrage de référence reste STROUSTRUP (B.) – *Le langage C++*, édition spéciale, Pearson Education France, 2003.

1. — Un environnement de développement à l'ancienne

Pour programmer en C++, il faut disposer d'un compilateur C++. Un compilateur C++ est un programme qui transforme un programme source (écrit en C++) en un programme exécutable – c'est-à-dire en un programme directement exécutable sur la machine dont on dispose. Une machine est la combinaison d'une architecture matérielle et d'un système d'exploitation. La plupart des micro-ordinateurs sont dotés d'une unité centrale fabriquée par la société *Intel* – modèles Pentium en général ; ils tournent grâce à un système d'exploitation fourni par *Microsoft* – Windows, soit Windows 98, soit Windows XP.

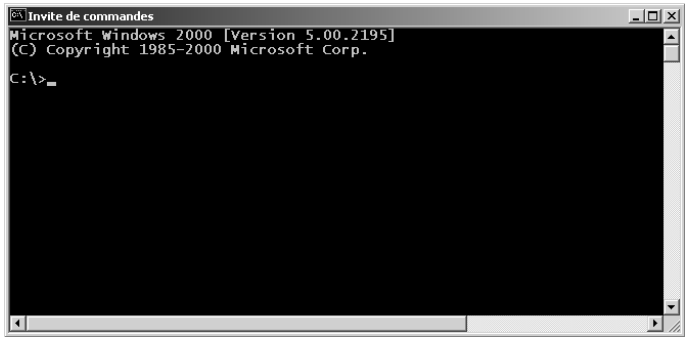
Microsoft propose un compilateur C++ ; nous allons cependant utiliser un compilateur du domaine public, le compilateur *GNU* fourni par la *Free Software Foundation*. Ce compilateur a d'abord été développé pour différentes unités centrales sous différentes variantes du système d'exploitation UNIX ; il a été assez récemment porté pour les différentes versions de Windows. Ce compilateur est «libre» : il est gratuit et, de plus, son programme source (le compilateur est écrit en langage C) est disponible librement. Notre première mission sera donc d'installer ce compilateur sur les machines de la salle MASERATI de la Faculté ; vous pourrez ainsi l'installer sur votre micro-ordinateur personnel.

Nous allons travailler *à l'ancienne*. Le programme C++ – le programme source – est tapé en utilisant les services d'un éditeur de texte. Ensuite, ce programme source est compilé à l'aide du compilateur C++. Enfin, le programme exécutable, engendré par le compilateur, est effectivement exécuté. Le cycle de développement d'un programme comporte ainsi les trois étapes suivantes :

1. À l'aide d'un éditeur de texte, il faut écrire le programme source C++.
2. À l'aide du compilateur C++, il faut compiler le programme – surviennent alors des erreurs de compilation, si le programme est invalide.
3. Le programme résultant est enfin exécuté – c'est l'heure en général des erreurs à l'exécution.

À l'inverse, il est possible de recourir à un *environnement de développement intégré*. Dans un tel environnement, on dispose d'un éditeur de texte ; on peut directement lancer la compilation du programme source puis l'exécution du programme résultant. Un tel environnement accélère en général le développement d'une application ; le gain reste relativement faible.

FIG. 1 – La boîte MS-DOS



Pour travailler *à l'ancienne*, en premier lieu, il faut se trouver dans un mode où l'utilisateur, au clavier, rentre différentes commandes pour, justement, invoquer l'éditeur de texte, puis le compilateur C++ et, enfin, le programme exécutable. J'appelle ce mode la «boîte MS-DOS» en référence au système d'exploitation de Microsoft d'avant Windows. Pour cela, cliquer sur `Démarrer` puis `Programmes` puis `Accessoires` puis `Invite de commandes`. On se retrouve dans une fenêtre où l'on peut entrer des lignes de commande (cf. la figure 1).

On va sans doute vouloir créer un répertoire (un *directory* en anglais) destiné à regrouper les premiers essais en C++. Pour cela, il faut invoquer la commande ***mkdir*** (pour *«MaKe DIRectory»*) :

`C:\>`***mkdir mes_premiers_pas***

où ***mes_premiers_pas*** est le nom que l'on donne à ce répertoire. J'utilise les conventions typographiques suivantes. Les caractères affichés par le système d'exploitation sont imprimés en utilisant une police à espacement fixe en gras ; les caractères entrés au clavier par l'utilisateur avec une police à espacement variable en italique gras. On peut ensuite se positionner, dans l'arbre du système de fichier, dans ce nouveau répertoire grâce à la commande ***cd*** (pour *«Change Directory»*) :

`C:\>`***cd mes_premiers_pas***

FIG. 2 – La fenêtre de l'éditeur de texte Write (de Microsoft)

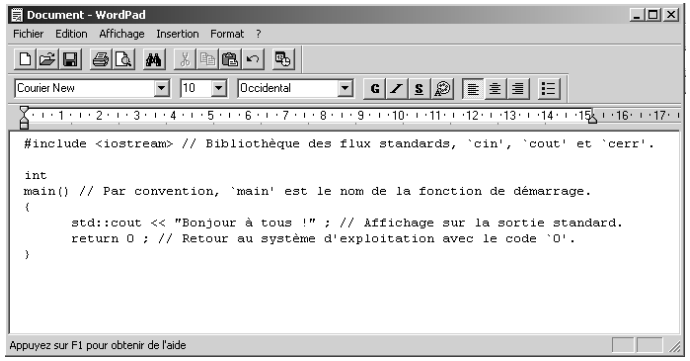
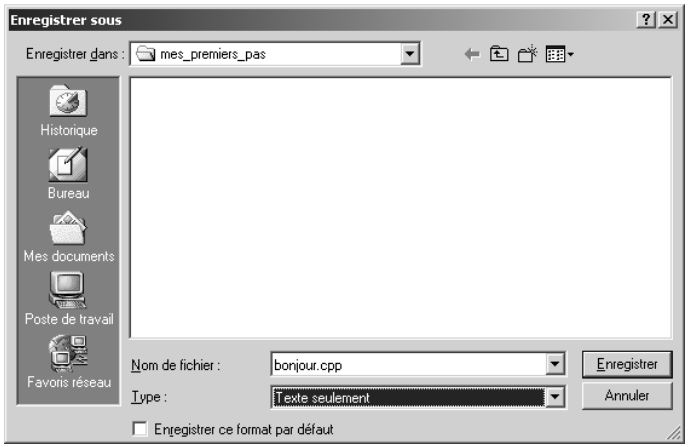


FIG. 3 – La boîte de dialogue « Enregistrer sous... » de Write



Dans ce répertoire, on invoque les services d'un éditeur de texte. L'on peut utiliser, par exemple, *Write* fourni en standard par Microsoft sur toutes les versions de Windows. Pour cela, il faut entrer la commande suivante :

```
C:\mes_premiers_pas>write
```

On bénéficie alors des services de Write. On peut ainsi entrer, au clavier, un premier programme C++ (par exemple, le programme proposé ci-après). La photo d'écran de la figure 2 illustre cela.

Il faut ensuite sauvegarder ce programme sous la forme d'un fichier : cliquer, pour cela, sur l'item du menu déroulant `Fichier` puis sur `Enregistrer sous...` (voir la figure 3). Il faut donner un nom au fichier ; il est préférable d'utiliser l'extension `cpp` qui spécifie habituellement un programme source C++ – cette convention n'est pas universelle, on trouve aussi soit `C`, soit `cc`. **Attention**, il importe de sauvegarder le texte sous la forme d'une suite toute simple de caractères (le format « texte seulement ») c'est-à-dire en acceptant de perdre toutes les informations relatives à la typographie — la police de caractères utilisée, les enrichissements (gras, italique, etc.), la valeur de l'interligne, ... On peut ensuite quitter Write.

Il est bon de s'assurer que le fichier est bien là. Pour cela, on peut utiliser la commande ***dir*** (pour *« DIRrectory »*) qui affiche toutes les entrées du répertoire courant :

```
C:\mes_premiers_pas>dir
```

On obtient alors la sortie suivante :

```
 Le volume dans le lecteur C s'appelle xxxxx
 Le numéro de série du volume est 1C4F-086B

 Répertoire de C:\mes_premiers_pas

09/11/2006  18:19       <DIR>          .
09/11/2006  18:19       <DIR>          ..
09/11/2006  18:19                   307 bonjour.cpp
               1 fichier(s)              307 octets
               2 Rép(s)   1 720 705 024 octets libres
```

On est enfin en mesure de compiler un premier programme. Pour invoquer le compilateur GNU, la commande suivante est utilisée :

```
C:\mes_premiers_pas>g++ bonjour.cpp
```

Le compilateur GNU n'est bavard qu'en cas d'erreurs ; si tout se passe bien, il reste muet. Par défaut, le nom du programme exécutable engendré par le compilateur est ***a.exe***. On le vérifie en utilisant la commande ***dir*** :

```
C:\mes_premiers_pas>dir
```

On obtient alors la sortie suivante :

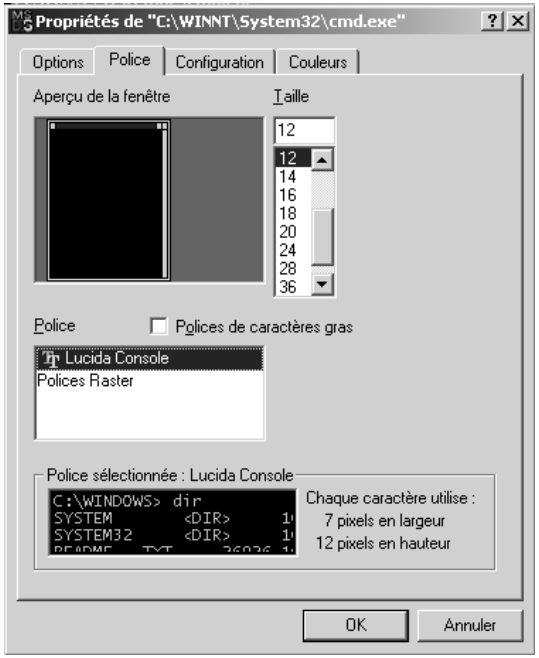
```
 Le volume dans le lecteur C s'appelle xxxxx
 Le numéro de série du volume est 1C4F-086B
```

```
 Répertoire de C:\mes_premiers_pas

09/11/2006  18:28       <DIR>          .
09/11/2006  18:28       <DIR>          ..
09/11/2006  18:27               448 537 a.exe
09/11/2006  18:19                   307 bonjour.cpp
               2 fichier(s)          448 844 octets
               2 Rép(s)   1 720 242 176 octets libres
```

On constate, effectivement, la présence du fichier de nom ***a.exe***.

FIG. 4 – La boîte de dialogue « Propriétés » de la boîte MS-DOS (onglet « Police »)



Le moment est venu d'exécuter ce premier programme. Ceci est obtenu en utilisant la commande ***a*** (en effet, le premier mot d'une commande est en général le nom d'un fichier qui, en y ajoutant l'extension « ***.exe*** », contient le programme à exécuter) :

```
C:\mes_premiers_pas>a
```

On obtient alors la sortie suivante :

```
Bonjour Ó tous !
```

D'un côté, on est radieux : ça marche ; de l'autre côté, on est agacé : le « ***à*** » s'affiche comme un « ***Ó*** » ! Cela vient ce que MS-DOS, pour les voyelles accentuées, n'utilise pas le même codage que Windows.

Pour pallier cette difficulté, le plus simple est de se fabriquer le fichier de commandes suivant :

```
1  @echo off
2  REM Fichier 'Boîte MS-DOS.bat'.
3  REM Ce fichier permet d'obtenir une fenêtre MS-DOS avec l'affichage étendu des caractères
4  REM en utilisant la "page de codes" 1252 qui semble correspondre au jeu latin-1 (celui
5  REM de Windows).
6
7  REM Affichage du jeu de caractères latin-1 ; il faut sélectionner une police True Type pour
8  REM que l'affichage soit correct dans la boîte MS-DOS.
9  mode con cp select=1252 > null
10 REM Il faut maintenant appeler l'interpréteur de commandes, sinon ce fichier de comman-
11 REM des retournerait directement. Attention de ne pas appeler 'command.com', l'antique
12 REM interpréteur MS-DOS.
13 cmd.exe
```

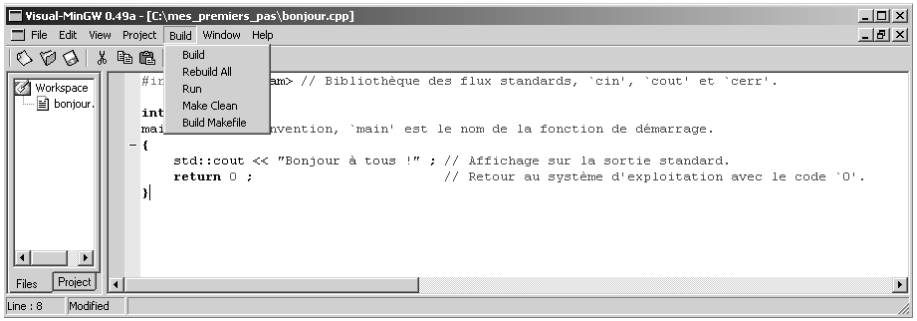
et de lui donner, par exemple, le nom ***Boîte MS-DOS.bat***. Ensuite, l'on peut déposer un raccourci vers ce fichier sur le bureau et, en double-cliquant sur ce raccourci, on se trouve directement dans une « boîte MS-DOS » qui affiche correctement les voyelles accentuées (telles qu'elles sont codées sous Windows). Il faut cependant choisir une police *True Type* pour que l'affichage soit correct (menu système, item `Propriétés` — voir la figure 4).

Sur la figure 5, on voit la fenêtre d'un environnement de développement intégré. J'ai récupéré, sur Internel, un logiciel intitulé « Visual-MinGW ». On observe notamment que l'éditeur de texte reconnaît la syntaxe du C++, en affichant distinctement les différentes unités lexicales. Le menu `Build` donne accès directement au compilateur (item `Build`) ou à l'exécution du programme (item `Run`).

2. — Mon premier programme en C++

Nous commençons par l'un des programmes les plus simples à écrire. Ce programme se contente d'afficher sur la sortie standard la chaîne de caractères « Bonjour à tous ! ».

FIG. 5 – La fenêtre d'un environnement de développement intégré



2-1. — Le programme *bonjour.cpp* et son exécution

Le texte de ce programme figure ci-après.

```
1  #include <iostream> // Bibliothèque des flux standards, 'cin', 'cout' et 'cerr'.
2
3  int
4  main() // Par convention, 'main' est le nom de la fonction de démarrage.
5  {
6    std::cout << "Bonjour à tous !" ; // Affichage sur la sortie standard.
7    return 0 ;                       // Retour au système d'exploitation avec le code '0'.
8  }
```

On pourra noter les points suivants.

1. La directive ***#include*** permet d'insérer un autre fichier ; au cas particulier, le fichier ***iostream*** qui déclare les flux par défaut de la bibliothèque standard.
2. Le système d'exploitation démarre le programme en appelant, par convention, la fonction dont le nom est ***main***.
3. L'opérateur « ***::*** » est l'opérateur de résolution de portée : « ***std::cout*** » désigne l'identificateur ***cout*** déclaré dans l'espace de noms ***std***.
4. Par convention, un programme se termine en retournant depuis la fonction ***main*** un nombre entier qui constitue le code de retour du programme.

L'exécution de ce programme donne le résultat suivant.

```
1  Bonjour à tous !
```

2-2. — L'utilisation de l'espace des noms globaux

Les identificateurs de la bibliothèque standard sont très souvent utilisés. Ils portent des noms connus qui ne sont pas en général utilisés par le programmeur. Aussi préfère-t-on, pour ne pas avoir à utiliser l'opérateur « ***::*** » pour les identificateurs de la bibliothèque standard, inclure l'espace de noms de cette bibliothèque dans l'espace de noms global grâce à l'instruction ***using***.

Le programme précédent devient ainsi le programme suivant.

```
1  #include <iostream> // Bibliothèque des flux standards, 'cin', 'cout' et 'cerr'.
2
3  using namespace std ; // Importation, dans l'espace des noms globaux, de l'espace 'std'.
4
5  int
6  main() // Par convention, 'main' est le nom de la fonction de démarrage.
7  {
8    cout << "Bonjour à tous !" ; // Affichage sur la sortie standard.
9    return 0 ;                  // Retour au système d'exploitation avec le code '0'.
10 }
```

3. — Les erreurs à la compilation et à l'exécution

Il est trop facile de faire des erreurs.

Les erreurs à la compilation sont signalées par un message du compilateur. Par exemple, l'oubli de la clause ***using*** dans le programme précédent entraîne une erreur dont le message n'est pas simple à interpréter. Ainsi le programme suivant :

```
1  #include <iostream> // Bibliothèque des flux standards, 'cin', 'cout' et 'cerr'.
2
3  int
4  main() // Par convention, 'main' est le nom de la fonction de démarrage.
5  {
6    cout << "Bonjour à tous !" ; // Affichage sur la sortie standard.
7    return 0 ;                  // Retour au système d'exploitation avec le code '0'.
8  }
```

conduit-il au message d'erreur :

```
1  bonjour_namespace_erreur.cpp: In function 'int main()':
2  bonjour_namespace_erreur.cpp:6: error: 'cout' undeclared (first use this function)
3  bonjour_namespace_erreur.cpp:6: error: (Each undeclared identifier is reported only once for each function it appears in.)
```

L'absence d'erreurs à la compilation ne veut pas dire que le programme ne soit pas exempt d'erreurs. Les erreurs à l'exécution se manifestent au moment de l'exécution du programme. Ainsi le programme suivant n'engendre pas d'erreurs à la compilation.

```
1  int
2  main()
3  {
4    int tableau[1];              // Un tableau d'entiers ne comportant qu'un seul élément.
5    tableau[1000000] = 10; // Oups, on écrit à un emplacement qui n'a pas été réservé !
6    return 0;                    // Retour au système d'exploitation avec le code '0'.
7  }
```

Son exécution conduit à l'affichage de la boîte de dialogue de la figure 6.

FIG. 6 – La boîte de dialogue d'une erreur à l'exécution

4. — Les trois modes d'allocation des variables

Le C++ distingue trois modes pour allouer des variables – allouer des variables veut dire leur réserver un emplacement en mémoire (ce qui oblige le programmeur, dans le cas d'un tableau, à connaître sa taille). Dans le tableau ci-après, je tente de synthétiser les différentes caractéristiques de ces trois modes.

Mode	Globale	Locale	Sur le tas
Emplacement	Segment du programme	Pile	Tas
Déclaration	À l'extérieur d'une fonction	Dans une fonction	Par l'opérateur ***new***
Portée	Globale	Locale	Locale
Naissance	Début du programme	Passage du flux	Exécution du ***new***
Décès	Fin du programme	Sortie de la portée	Exécution du ***delete***
Limite	Taille fixe	Taille fixe	Gestion compliquée

Le programme suivant alloue ces trois types de variables, sans jamais les initialiser – ce qui est mal.

```
1  #include <iostream> // cin, cout et cerr.
2
3  using namespace std;
4
5  int d1; // Une première variable entière globale.
6  int d2; // Une seconde variable entière globale.
7
8  // Retourne l'adresse de la variable passée en argument.
9  size_t
10 adresse(int & i)
11 {
12   // Conversion douteuse qui nécessite un 'reinterpret_cast'.
13   return reinterpret_cast<size_t>(& i);
14 }
15 int
16 main()
17 {
18   int d3;                  // Une première variable entière locale.
19   int d4;                  // Une deuxième variable entière locale.
20   int & d5 = * new int; // Une première variable entière allouée sur le tas.
21   int & d6 = * new int; // Une seconde variable entière allouée sur le tas.
22
23   cout << "Adresse de d1 " << adresse(d1) << endl;
24   cout << "Valeur de  d1 " << d1 << endl;
25   cout << "Adresse de d2 " << adresse(d2) << endl;
26   cout << "Valeur de  d2 " << d2 << endl;
27   cout << "Adresse de d3 " << adresse(d3) << endl;
28   cout << "Valeur de  d3 " << d3 << endl;
29   cout << "Adresse de d4 " << adresse(d4) << endl;
30   cout << "Valeur de  d4 " << d4 << endl;
31   cout << "Adresse de d5 " << adresse(d5) << endl;
32   cout << "Valeur de  d5 " << d5 << endl;
33   cout << "Adresse de d6 " << adresse(d6) << endl;
34   cout << "Valeur de  d6 " << d6 << endl;
35   {
36     int d7; // Une troisième variable entière locale.
37     cout << "Adresse de d7 " << adresse(d7) << endl;
38     cout << "Valeur de  d7 " << d7 << endl;
39   }
40   {
41     int d8; // Une quatrième variable entière locale.
42     cout << "Adresse de d8 " << adresse(d8) << endl;
```

```
43      cout << "Valeur de  d8 " << d8 << endl ;
44    }
45    return 0 ; // Retour au système d'exploitation avec le code '0'.
46  }
```

L'exécution de ce programme conduit aux résultats suivants.

```
1  Adresse de d1 4468752
2  Valeur de  d1 0
3  Adresse de d2 4468756
4  Valeur de  d2 0
5  Adresse de d3 2293616
6  Valeur de  d3 30
7  Adresse de d4 2293612
8  Valeur de  d4 2293672
9  Adresse de d5 4014040
10 Valeur de  d5 0
11 Adresse de d6 4014072
12 Valeur de  d6 3998088
13 Adresse de d7 2293600
14 Valeur de  d7 2009116333
15 Adresse de d8 2293600
16 Valeur de  d8 2009116333
```

Notons que ***int*** est le type intégré des entiers relatifs. À la différence des entiers des mathématiques, les entiers en informatique ont une capacité limitée. Sur les micro-ordinateurs courants, les entiers sont codés sur quatre octets. Le nombre le plus grand qui puisse être représenté est alors égal à 2^{31} – un bit est utilisé pour indiquer le signe du nombre. Le type intégré des entiers naturels est ***size_t***. Il est préférable d'utiliser ***size_t*** pour les entiers qui sont positifs ou nuls ; utiliser ***int*** dans ce cas de figure constitue une imprécision.

On observe ainsi

1. que les variables globales sont initialisées par défaut à zéro ;
2. que les variables allouées localement ou sur le tas ne sont pas initialisées ;
3. que l'emplacement alloué pour une variable locale est réutilisé : ***d7*** et ***d8*** ont la même adresse (et en conséquence la même valeur).

Les quatre règles suivantes me semblent pertinentes.

1. Utiliser le moins possible des variables globales.
2. Donner aux variables globales un nom long explicite.
3. Déclarer et initialiser dans la même instruction les variables locales.
4. Donner aux variables locales un nom court conventionnel.

5. — Les arguments d'une fonction

Pour pouvoir réutiliser le code, il importe de pouvoir définir des fonctions. Ces fonctions s'utilisent, en général, avec des arguments pour pouvoir les appeler pour exécuter une tâche particulière. Par exemple, la fonction ***sqrt(...)*** définit l'algorithme numérique de calcul de la racine carrée. Ensuite, l'utilisateur appelle la fonction avec son argument effectif : par exemple, ***sqrt(2)***.

En C, les arguments, quand la fonction est appelée, sont par défaut recopiés. Aussi, dans l'appelé, dispose-t-on librement de l'argument. Quand un argument est modifié, cette modification ne se répercute pas dans le programme de l'appelant.

Le programme suivant illustre cela.

```
1  #include <iostream> // cin, cout et cerr.
2
3  using namespace std ;
4
5  void
6  imprimer_des_dollars_v1(size_t nombre)
7  {
8    size_t n = nombre ; // Opération inutile de recopie de l'argument :
9    while ( n -- > 0 ) {    // 'nombre' est déjà une variable locale.
10     cout << '$' ; }
11 }
12 void
13 imprimer_des_dollars_v2(size_t nombre)
14 {
15   while ( nombre -- > 0 ) { // L'appelant n'est pas affecté par la décrémentation.
16     cout << '$' ; }              // L'appelé dispose librement de l'argument.
17 }
18 int
19 main()
20 {
21   imprimer_des_dollars_v1(10) ;
22   cout << endl ;
23   imprimer_des_dollars_v2(10) ;
24   return 0 ; // Retour au système d'exploitation avec le code '0'.
25 }
```

L'exécution de ce petit programme donne la sortie suivante.

```
$$$$$$$$$$
$$$$$$$$$$
```

La recopie d'un objet peut être *a priori* coûteuse. Le C++ – mais aussi pour d'autres raisons – a introduit la notion de «référence». Un argument peut être passé à l'appelé par référence. Le mot-clé ***const*** précise alors que l'argument ne doit pas être modifié par l'appelé.

Le programme précédent est alors transformé comme suit.

```
#include <iostream> // cin, cout et cerr.

using namespace std ;

void
imprimer_des_dollars_v1(size_t & nombre)
{
  while ( nombre -- > 0 ) { // Oups, l'appelant est affecté par la décrémentation.
    cout << '$' ; }
}
void
imprimer_des_dollars_v2(const size_t & nombre)
{
  size_t n = nombre ; // Recopie nécessaire de l'argument.
  while ( n -- > 0 ) {
    cout << '$' ; }
}
int
main()
{
  size_t nombre = 10 ;
  cout << "Valeur de 'nombre' avant l'appel = " << nombre << endl ;
  imprimer_des_dollars_v1(nombre) ;
  cout << "\nValeur de 'nombre' après l'appel = " << nombre << endl ;
  imprimer_des_dollars_v2(10) ;
  return 0 ; // Retour au système d'exploitation avec le code '0'.
}
```

Son exécution produit la sortie qui suit.

```
Valeur de 'nombre' avant l'appel = 10
$$$$$$$$$$
Valeur de 'nombre' après l'appel = 4294967295
$$$$$$$$$$
```

L'on peut noter le débordement de capacité, par valeur inférieure, pour une variable de type ***size_t*** : ***0 - 1*** donne ***4294967295*** (le plus grand nombre qu'une variable de type ***size_t*** puisse représenter). Notons que l'instruction, par exemple, « ***imprimer_des_dollars_v1(10) ;*** » aurait été refusée par le compilateur : une constante ne peut pas constituer une référence non ***const***.

Les arguments en entrée seront donc passés par référence ***const***. Par exemple, on va coder, pour calculer le maximum de deux entiers naturels :

```
#include <iostream> // cin, cout et cerr

size_t
max(const size_t & x, const size_t & y)
{
  if ( x < y ) {
    return y ; }
  else {
    return x ; }
}
```

Pour les résultats de la fonction, la valeur que retourne la fonction permet déjà retourner à l'appelant un premier résultat. Pöur retourner deux valeurs, on dispose de trois tactiques. Pour illustrer notre propos, prenons la fonction qui retourne le minimum et le maximum de deux entiers naturels. En premier lieu, la fonction peut retourner une ***pair***. Elle est alors définie – et utilisée – comme suit.

```
#include <iostream> // cin, cout et cerr
#include <utility>     // 'pair' et 'make_pair(...)'

using namespace std ;

pair<size_t, size_t>
min_max(const size_t & x, const size_t & y)
{
  if ( x < y ) {
    return make_pair(x, y) ; }
  else {
    return make_pair(y, x) ; }
}
int
```

```
15 main()
16 {
17   pair<size_t, size_t> r = min_max(10, 12) ;
18   cout << "Le min et le max sont " << r.first << " et " << r.second << '.' << endl ;
19   r = min_max(22, 20) ;
20   cout << "Le min et le max sont " << r.first << " et " << r.second << '.' << endl ;
21   return 0 ; // Retour au système d'exploitation avec le code '0'.
22 }
```

L'exécution de ce programme engendre la sortie suivante.

```
1 Le min et le max sont 10 et 12.
2 Le min et le max sont 20 et 22.
```

On a là un premier exemple de programmation *générique* – c'est-à-dire la programmation où les types utilisés sont des paramètres. L'objet ***pair***, dans le fichier ***utility***, est défini de façon très générale. Quand on utilise la syntaxe « ***pair<size_t, size_t>*** » cela veut dire que l'on veut utiliser la version de l'objet ***pair*** spécialisée pour les couples de ***size_t***. Pour spécifier une paire d'entiers relatifs, de type ***int***, on aurait codé « ***pair<int, int>*** ». Notons que pour fabriquer une paire, on utilise, par exemple, la syntaxe « ***make_pair(x, y)*** ». Il faudrait en fait écrire « ***make_pair<size_t, size_t>(x, y)*** ». Le compilateur est cependant suffisamment intelligent pour déduire, à partir du type des arguments, la spécialisation nécessaire. Comme cela, nous pouvons être paresseux et écrire seulement « ***make_pair(x, y)*** » à la place de « ***make_pair<size_t, size_t>(x, y)*** ».

La deuxième tactique pour retourner deux valeurs consiste à retourner la première valeur comme le résultat de la fonction et la seconde valeur comme un argument en sortie. On aurait alors la définition suivante de la fonction ***min_max***.

```
1  #include <iostream> // cin, cout et cerr
2
3  using namespace std ;
4
5  size_t
6  min_max(const size_t & x, const size_t & y, size_t & maximum)
7  {
8    if ( x < y ) {
9      maximum = y ;
10     return x ; }
11   else {
12     maximum = x ;
13     return y ; }
14 }
15 int
16 main()
17 {
18   size_t mon_max ;
19   size_t mon_min = min_max(10, 12, mon_max) ;
20   cout << "Le min et le max sont " << mon_min << " et " << mon_max << '.' << endl ;
21   mon_min = min_max(22, 20, mon_max) ;
22   cout << "Le min et le max sont " << mon_min << " et " << mon_max << '.' << endl ;
23   return 0 ; // Retour au système d'exploitation avec le code '0'.
24 }
```

L'exécution de ceci conduit à la sortie suivante.

```
1 Le min et le max sont 10 et 12.
2 Le min et le max sont 20 et 22.
```

Notons que que le troisième argument de la fonction est déclaré par la syntaxe « ***size_t &*** » : il est passé par référence non ***const*** pour que l'appelé puisse modifier la valeur de la variable chez l'appelant.

Je n'aime pas tellement cette tactique. Celle-ci conduit à ne pas symétriser les deux résultats. Je préfère la troisième tactique où les deux résultats sont retournés tous les deux comme des arguments en sortie. Cette tactique conduit au programme suivant.

```
1  #include <iostream> // cin, cout et cerr
2
3  using namespace std ;
4
5  void
6  min_max(const size_t & x, const size_t & y, size_t & minimum, size_t & maximum)
7  {
8    if ( x < y ) {
9      minimum = x ;
10     maximum = y ; }
11   else {
12     minimum = y ;
13     maximum = x ; }
14 }
15 int
16 main()
17 {
18   size_t mon_max, mon_min ;
```

```
19   min_max(10, 12, mon_min, mon_max) ;
20   cout << "Le min et le max sont " << mon_min << " et " << mon_max << '.' << endl ;
21   min_max(22, 20, mon_min, mon_max) ;
22   cout << "Le min et le max sont " << mon_min << " et " << mon_max << '.' << endl ;
23   return 0 ; // Retour au système d'exploitation avec le code '0'.
24 }
```

En exécutant ce programme, on obtient ce qui suit.

```
1  Le min et le max sont 10 et 12.
2  Le min et le max sont 20 et 22.
```

Notons le mot réservé ***void*** utilisé pour dire que la fonction ne retourne rien. Notons aussi que dans ce cas, dans le corps de la fonction, il n'est pas nécessaire de coder une instruction ***return***.

Pour conclure, un argument de type ***T*** en entrée devrait toujours être spécifié sous la forme « ***const T &*** » pour être passé par référence ***const***. On évite la recopie de l'argument – recopie qui pourrait être coûteuse s'il s'agit d'un gros objet (une matrice de 10 000 nombres par exemple) – tout en protégeant la variable chez l'appelant. Les résultats, dans la mesure du possible, devraient être retournés au moyen de l'instruction ***return***. C'est assez naturel quand le résultat est unique. Quand le résultat est un couple de variable, on peut utiliser l'objet ***pair***. Quand le nombre de résultats excède deux, on est obligé d'utiliser des arguments en sortie en utilisant la syntaxe « ***T &*** » pour passer l'argument par référence non ***const***.

6. — La récursion

Le programme qui suit a pour but d'afficher, chiffre par chiffre, un entier naturel en base 10.

Une première solution repose sur l'appel récursif d'une fonction ; une seconde sur la détermination de l'ordre de grandeur de l'entier naturel pour afficher dans un second temps les chiffres en commençant par les chiffres de « poids fort ». Il est toujours possible de « dérécursiver » un algorithme récursif – au prix, quand le problème s'exprime facilement sous une forme récursive, d'un effort de programmation. Le temps d'exécution de l'algorithme récursif est en général plus long – mais la pénalité dépend de la machine utilisée.

```
1  #include <iostream> // cin, cout et cerr.
2
3  using namespace std ;
4
5  void
6  ma_fonction_1(const size_t & n)
7  {
8    if ( n <= 9 ) {
9      cout << n ;              // Affichage du nombre qui est un chiffre.
10     return ; }               // Fin de la récursion.
11   ma_fonction_1(n/10) ; // Appel récursif.
12   cout << n%10 ;           // L'opérateur '%' donne le reste de la division euclidienne ;
13   return ;                 // on affiche ainsi le chiffre des unités.
14 }
15 void
16 ma_fonction_2(const size_t & n)
17 {
18   size_t ordre_de_grandeur = 1 ;
19   size_t m = n ;
20   // Détermination de l'ordre de grandeur du nombre.
21   while ( (m /= 10) > 0 ) {          // 'm /= 10' est équivalent à 'm = m/10'.
22     ordre_de_grandeur *= 10 ; } // 'x *= 10' est équivalent à 'x = x*10'.
23   m = n ;
24   // Affichage des chiffres à partir de la gauche.
25   while ( ordre_de_grandeur != 0 ) {
26     cout << m/ordre_de_grandeur ; // Affichage du chiffre le plus à gauche.
27     m %= ordre_de_grandeur ;        // Réduction du nombre.
28     ordre_de_grandeur /= 10 ; }     // Réduction de l'ordre de grandeur.
29 }
30 void
31 test(const size_t & n)
32 {
33   cout << n << ' ' ; ma_fonction_1(n) ; cout << ' ' ; ma_fonction_2(n) ; cout << endl ;
34 }
35 int
36 main()
37 {
38   test(0) ;
39   test(2) ;
40   test(10) ;
41   test(123) ;
42   test(12345678) ;
43   return 0 ; // Retour au système d'exploitation avec le code '0'.
44 }
```

L'exécution de ce programme conduit aux résultats suivants.

```
0 0 0
2 2 2
10 10 10
123 123 123
12345678 12345678 12345678
```

7. — Les classes

Avant qu'un nom ait été donné au C++, ce dernier s'appelait simplement « C avec classes ». Les classes ne sont que le moyen de définir de nouveaux types. Ces nouveaux types s'utilisent ensuite exactement comme les types intégrés (c'est-à-dire les types définis par le langage).

Dans un premier temps, les classes peuvent être vues comme des enregistrements qui rassemblent plusieurs variables, par exemple, un enregistrement qui décrit une personne et qui comporte le nom, le prénom, la date de naissance, etc. Mais une classe est plus que cela. Déjà parce que, lorsqu'on déclare une classe, l'on peut spécifier un couple de constructeur-destructeur pour la classe. Un constructeur est une fonction qui précise de quelle manière une instance de cette classe est initialisée. Un destructeur indique de quelle façon l'instance doit disparaître.

7-1. — Le couple constructeur-destructeur

Il nous faut dans un premier temps explorer complètement ces notions de constructeur et de destructeur.

Dans le programme qui suit, nous déclarons une classe, de nom ***Objet***, qui encapsule une seule variable, de type ***string*** et de nom ***nom***. Le constructeur utilisé comprend un seul argument, le nom à donner à l'instance de la classe. Un constructeur est une fonction propre à la classe – ces fonctions sont aussi appelées *méthodes* – qui a le même nom que la classe. Si la classe s'appelle ***Bidule***, toutes les méthodes qui s'appellent ***Bidule*** sont des constructeurs de cette classe. Le destructeur ne comprend pas, nécessairement, d'argument. Son nom est celui de la classe, précédé du caractère « **~** ».

Les trois modes d'allocation des variables – globale, locale et sur le tas – s'appliquent aussi aux classes. Nous nous proposons ainsi dans le programme de déclarer trois instances de la classe, une instance globale, une instance locale et une instance sur le tas.

```
// Attention, ce programme n'est pas nécessairement valide : il utilise la bibliothèque
// standard 'iostream' alors qu'il n'est pas sûr que cette dernière soit initialisée.

#include <iostream> // 'cin', 'cout' et 'cerr'.
#include <iomanip> // setw(...).

using namespace std ;

class Objet {
  string nom ;
public :
  // Constructeur.
  Objet(const string & arg) : nom(arg) {
    cout << "Appel du constructeur -- '" << nom << '\'' << endl ; }
  // Destructeur.
  ~Objet() {
    cout << "Appel du destructeur -- '" << nom << '\'' << endl ; }
} ;

Objet instance_globale("Instance globale") ; // Instruction potentiellement invalide.

int
main()
{
  cout << "À l'entrée de 'main()'" << endl ;

  Objet instance_locale("Instance locale") ;
  Objet & instance_sur_le_tas = * new Objet("Instance sur le tas") ;

  delete & instance_sur_le_tas ;

  cout << "À la sortie de 'main()'" << endl ;
  return 0 ; // Retour au système d'exploitation avec le code '0'.
}
```

Notons la syntaxe un peu bizarre utilisée dans le constructeur. Ce constructeur, en l'absence de l'instruction « ***cout << ... << endl ;*** », se réduirait à :

```
Objet(const string & arg) : nom(arg) {}
```

Le corps du constructeur est vide et la syntaxe « ***: nom(arg)*** » est la syntaxe utilisée pour dire que l'attribut ***nom*** – c'est ainsi que l'on appelle une variable propre à une classe – est initialisé. On aurait aussi pu coder :

```
Objet(const string & arg) {
  nom = arg ; }
```

mais cela aurait été moins expressif.

L'exécution du programme engendre la sortie suivante.

```
1  Appel du constructeur -- 'Instance globale'
2  À l'entrée de 'main()'
3  Appel du constructeur -- 'Instance locale'
4  Appel du constructeur -- 'Instance sur le tas'
5  Appel du destructeur -- 'Instance sur le tas'
6  À la sortie de 'main()'
7  Appel du destructeur -- 'Instance locale'
8  Appel du destructeur -- 'Instance globale'
```

On observe notamment

1. que le constructeur de l'instance globale est appelé avant que le flux du programme n'entre dans la fonction ***main()*** ;
2. que le destructeur de cette même instance est appelé après que le flux du programme soit passé par l'instruction « ***return 0 ;*** » ;
3. que le constructeur de l'instance locale est appelé au moment de la déclaration de cette variable ;
4. que son destructeur est appelé à la fin du bloc que constitue la définition de la fonction ***main()*** ;
5. que le constructeur de l'instance sur le tas est appelé au moment de l'exécution de l'opérateur ***new*** ;
6. que son destructeur est appelé au moment de l'exécution du l'opérateur ***delete***.

Le C++ garantit que, pour chaque instance d'une classe créée, un constructeur est effectivement appelé ; de même, chaque fois qu'une instance est détruite, un destructeur est appelé – que cette destruction vienne de la fin du programme (instance globale), de ce que le flux du programme sorte de la portée de l'instance (instance locale) ou de l'exécution de l'opérateur ***delete***.

7-2. — Les différents constructeurs

Soit la classe ***Objet***. Trois types de constructeurs peuvent être distingués.

Quand on code « ***Objet mon_premier_objet ;*** » cela veut dire, d'une part, que l'on déclare une instance de la classe ***Objet*** (le nom de cette instance est ***mon_premier_objet***) et, d'autre part, que l'on veut utiliser le constructeur par défaut – celui sans argument déclaré par la syntaxe « ***Objet () : ... { ... }*** ».

Si on code « ***Objet mon_second_objet(mon_premier_objet) ;*** », cela veut dire que l'on déclare une instance (de nom ***mon_second_objet)*** à partir d'une autre instance (de nom ***mon_premier_objet)***. C'est alors le constructeur de copie qui est utilisé. La syntaxe utilisé pour déclarer ce constructeur est de la forme « ***Objet (const Objet & arg) : ... { ... }*** ».

Enfin, le code « ***Objet mon_objet("Bidule") ;*** » veut dire que l'on utilise un constructeur « normal » avec un argument – au cas particulier une chaîne de caractères. Le constructeur est alors déclaré avec la syntaxe « ***Objet (const & string arg) : ... { ...}*** ».

Le programme ci-après illustre tout cela, en indiquant même toutes les variétés syntaxiques que l'on peut trouver.

```
1  #include <iostream> // 'cin', 'cout' et 'cerr'.
2  #include <iomanip> // setw(...).
3
4  using namespace std ;
5
6  class Objet {
7    string nom ;
8  public :
9    // Constructeur par défaut.
10   Objet() : nom("<non initialisé>") {
11     cout << "Appel du constructeur par défaut   -- " << nom << endl ; }
12   // Constructeur de copie.
13   Objet(const Objet & arg) : nom("copie de " + arg.nom) {
14     cout << "Appel du constructeur de copie     -- " << nom << endl ; }
15   // Constructeur normal.
16   Objet(const string & arg) : nom(arg) {
17     cout << "Appel du constructeur avec argument -- " << nom << endl ; }
18   // Destructeur.
19   ~Objet() {
20     cout << "Appel du destructeur -- " << nom << endl ; }
21   // Opérateur d'affectation.
22   Objet & operator=(const Objet & arg) {
23     cout << "Appel de l'opérateur d'affectation -- " << arg.nom << endl ;
24     nom = "affectation de " + arg.nom ; }
25 } ;
26 void
27 num_ligne(const size_t & num)
28 {
29   cout << "Ligne " << setw(2) << num << ' ' ;
30 }
31 int
32 main()
33 {
34   // Appel du constructeur normal.
35   num_ligne(__LINE__) ; Objet premier("'Ceci est mon objet'") ;
36   // Première syntaxe de l'appel du constructeur par défaut.
37   { num_ligne(__LINE__) ; Objet second ; }
38   // Deuxième syntaxe de l'appel du constructeur par défaut.
```

```
39    { num_ligne(__LINE__) ; Objet second = Objet() ; }
40    // Syntaxe pour déclarer la fonction 'second', sans argument, qui retourne un 'Objet'.
41    { num_ligne(__LINE__) ; Objet second() ; cout << endl ; }
42    // Première syntaxe de l'appel du constructeur de copie.
43    { num_ligne(__LINE__) ; Objet second(premier) ; }
44    // Deuxième syntaxe de l'appel du constructeur de copie.
45    { num_ligne(__LINE__) ; Objet second = premier ; }
46    // Troisième syntaxe de l'appel du constructeur de copie.
47    { num_ligne(__LINE__) ; Objet second = Objet(premier) ; }
48    // Appel de l'opérateur d'affectation.
49    Objet second ; second = premier ;
50    return 0 ; // Retour au système d'exploitation avec le code '0'.
51  }
```

L'exécution de ce programme engendre la sortie suivante.

```
1   Ligne 35 Appel du constructeur avec argument -- 'Ceci est mon objet'
2   Ligne 37 Appel du constructeur par défaut   -- <non initialisé>
3   Appel du destructeur -- <non initialisé>
4   Ligne 39 Appel du constructeur par défaut   -- <non initialisé>
5   Appel du destructeur -- <non initialisé>
6   Ligne 41
7   Ligne 43 Appel du constructeur de copie     -- copie de 'Ceci est mon objet'
8   Appel du destructeur -- copie de 'Ceci est mon objet'
9   Ligne 45 Appel du constructeur de copie     -- copie de 'Ceci est mon objet'
10  Appel du destructeur -- copie de 'Ceci est mon objet'
11  Ligne 47 Appel du constructeur de copie     -- copie de 'Ceci est mon objet'
12  Appel du destructeur -- copie de 'Ceci est mon objet'
13  Appel du constructeur par défaut   -- <non initialisé>
14  Appel de l'opérateur d'affectation -- 'Ceci est mon objet'
15  Appel du destructeur -- affectation de 'Ceci est mon objet'
16  Appel du destructeur -- 'Ceci est mon objet'
```

7-3. — Les constructeurs et l'opérateur d'affectation par défaut

Toute cette affaire est rendue encore un peu plus compliquée par le fait que le compilateur, en l'absence d'indications contraires fournies par le programmateur, offre trois méthodes par défaut. Ces trois méthodes sont les suivantes.

1. Un constructeur par défaut, celui sans argument.
2. Un constructeur de copie par défaut, où les différents attributs de l'objet sont initialisés par leur constructeur de copie.
3. Un opérateur d'affectation par défaut, où les différents attributs de l'objet sont affectés par leur opérateur d'affectation.

Le programme suivant tente d'illustrer tout cela.

```
1   #include <iostream> // 'cin', 'cout' et 'cerr'.
2
3   using namespace std ;
4
5   class Objet {
6     string nom_ ;
7     int nombre_ ;
8   public :
9     void fixer_nom(const string & arg) { nom_ = arg ; }
10    void fixer_nombre(const int & arg) { nombre_ = arg ; }
11    const string & nom() const { return nom_ ; }
12    const int & nombre() const { return nombre_ ; }
13  } ;
14  int
15  main()
16  {
17    // Appel du constructeur par défaut, offert par le compilateur.
18    Objet mon_objet ;
19    cout << "Nom = '" << mon_objet.nom() << '\'' << endl ;
20    cout << "Nombre = " << mon_objet.nombre() << endl ;
21    mon_objet.fixer_nom("Keynes") ;
22    mon_objet.fixer_nombre(99) ;
23    cout << "Nom = '" << mon_objet.nom() << '\'' << endl ;
24    cout << "Nombre = " << mon_objet.nombre() << endl ;
25    // Appel du constructeur de copie, offert par le compilateur.
26    Objet autre_objet(mon_objet) ;
27    cout << "Nom = '" << autre_objet.nom() << '\'' << endl ;
28    cout << "Nombre = " << autre_objet.nombre() << endl ;
29    // Appel de l'opération d'affectation, offerte par le compilateur.
30    Objet encore_un_autre_objet ;
31    encore_un_autre_objet = mon_objet ;
32    cout << "Nom = '" << encore_un_autre_objet.nom() << '\'' << endl ;
33    cout << "Nombre = " << encore_un_autre_objet.nombre() << endl ;
34
35    return 0 ; // Retour au système d'exploitation avec le code '0'.
36  }
```

La sortie suivante résulte de l'exécution de ce programme.

```
Nom = ''
Nombre = 4014040
Nom = 'Keynes'
Nombre = 99
Nom = 'Keynes'
Nombre = 99
Nom = 'Keynes'
Nombre = 99
```

On observe notamment les quatre points suivants.

1. Le constructeur par défaut initialise l'attribut ***nom_*** à «» parce que l'initialisation par défaut d'une ***string*** est «» («» désigne la chaîne de caractères vide).
2. Le constructeur par défaut n'initialise pas l'attribut ***nombre_*** parce qu'une variable locale de type ***int*** n'est pas initialisée par défaut – ceci est bien sûr regrettable mais résulte de la volonté d'être semblable au langage C sur ce point.
3. La sémantique du constructeur de copie est sans surprise ; chaque attribut de l'instance de destination reçoit une copie de l'attribut correspondant de l'instance source.
4. De même, la sémantique de l'opérateur d'affectation est assez évidente.

Il existe deux conventions pour accéder en lecture et en écriture à un attribut d'un objet. Soit on définit les méthodes ***get_xxx*** et ***set_xxx*** ; soit on définit la méthode ***set_xxx*** et la méthode qui permet de lire l'attribut reçoit le nom ***xxx***. La première solution conduit au fragment de programme suivant.

```
class Objet {
  int xxx ;
public :
  int get_xxx() const { return xxx ; }
  void set_xxx(const int & arg) { xxx = arg ; }
} ;
```

La seconde solution revient à coder comme suit.

```
class Objet {
  int xxx_ ;
public :
  int xxx() const { return xxx_ ; }
  void set_xxx(const int & arg) { xxx_ = arg ; }
} ;
```

Le nom de l'attribut est alors suffixé par le caractère «_».

Les méthodes qui accèdent en lecture seulement aux attributs sont déclarées ***const***. Le compilateur sait alors que l'appel de cette méthode laisse l'objet inchangé et il peut ainsi optimiser le code engendré. Par exemple, si l'on code la ligne suivante :

int y = mon_objet.get_xxx() + mon_objet.get_xxx() + mon_objet.get_xxx() ;

Comme la méthode ***get_xxx*** a été déclarée ***const***, le compilateur peut déduire que les deuxième et troisième appels à cette méthode vont retourner le même résultat. Aussi ce dernier peut-il remplacer cette ligne par la ligne suivante.

int y = 3*mon_objet.get_xxx() ;

8. — Les vecteurs de la STL

La STL – pour *Standard Template Library* – est particulièrement utile. Elle définit des conteneurs génériques (c'est-à-dire des conteneurs paramétrés par le type de leur élément) ; le plus utilisé de ces conteneurs est sans conteste le vecteur.

Un vecteur de nombres naturels est déclaré à l'aide de la syntaxe suivante.

#include <vector>
vector<size_t> mon_vecteur ;

Un vecteur de la STL est dynamique à la différence d'un tableau dont la taille doit être indiquée par le programmeur. À l'insu du programmeur, un vecteur de la STL alloue sur le tas la mémoire nécessaire pour stocker les éléments du vecteur. De plus, un vecteur se rappelle de sa taille : il suffit de coder «***mon_vecteur.size()***» pour obtenir le nombre d'éléments que le vecteur comporte.

L'exemple du calcul de la médiane permet d'illustrer l'intérêt des vecteurs de la STL. Les nombres sont lus dans un fichier et le nombre de nombres n'est pas connu à l'avance. Dans l'exemple ci-après, j'oppose trois méthodes pour calculer la médiane.

1. Se donner un nombre de nombre maximum et utiliser un tableau qui comporte ce nombre maximum d'éléments.
2. Allouer sur le tas un tableau et réallouer ce tableau s'il déborde.
3. Utiliser un vecteur de la STL.

La première méthode est clairement inacceptable. La deuxième n'est pas simple à programmer et le programmeur n'est pas à l'abri de l'erreur dite de «la fuite de mémoire». La troisième solution n'est que du bonheur.

```
#include <iostream>   // 'cin', 'cout' et 'cerr'.
#include <fstream>    // ifstream.
#include <iterator>   // istream_iterator
#include <algorithm> // sort(...) de la STL.
#include <vector>     // Vecteurs de la STL.
#include <cmath>      // _isnan(...)

using namespace std ;

double
mediane_1(const string & nom)
{
  const size_t MAX = 10000 ; // Nombre maximum de nombres.
  double tableau[MAX] ;      // Tableau pour stocker les nombres.
  size_t nombre = 0 ;        // Indice dans tableau[] ; puis nombre de nombres.
  ifstream fichier(nom.c_str()) ;
  if ( ! fichier ) {
    cerr << "Impossible d'ouvrir en lecture '" << nom << "'." << endl ;
    return 0./0. ; }
  double tmp ;
  while ( fichier >> tmp ) {
    if ( nombre == MAX ) {
      cerr << "Désolé, il y a trop de nombres ; le maximum est " << MAX << '.' << endl ;
      return 0./0. ; }
    tableau[nombre] = tmp ;
    ++ nombre ; }
  if ( nombre == 0 ) {
    cerr << "Aucun nombre lu dans le fichier." << endl ;
    return 0./0. ; }
  sort(tableau, tableau+nombre) ;
  if ( nombre % 2 == 0 ) {
    // Interpolation linéaire si l'effectif est pair.
    return .5*tableau[nombre/2-1] + .5*tableau[nombre/2] ; }
  else {
    return  tableau[nombre/2] ; }
}
double
mediane_2(const string & nom)
{
  const size_t QUANTUM = 100 ; // Incrément pour la réallocation.
  double * tableau ;           // Pointeur sur l'emplacement obtenu par 'new'.
  size_t max = 0 ;             // Nombre courant maximum de nombres.
  size_t nombre = 0 ;          // Indice puis nombre de nombres.

  ifstream fichier(nom.c_str()) ;
  if ( ! fichier ) {
    cerr << "Impossible d'ouvrir en lecture '" << nom << "'." << endl ;
    return 0./0. ; }
  double tmp ;
  while ( fichier >> tmp ) {
    if ( nombre == max ) // Oups, il faut réallouer.
    {
      // L'opérateur 'new' permet d'obtenir une région de mémoire, allouée sur le tas.
      double * nouveau = new double[max+QUANTUM] ;
      // L'opérateur 'new' retourne un pointeur nul en cas d'échec.
      if ( nouveau == 0 ) {
        // Ne pas oublier de rendre l'ancienne région, sinon "fuite de mémoire".
        if ( nombre > 0 ) {
          delete tableau ; }
        cerr << "Impossible d'allouer de la mémoire sur le tas." << endl ;
        return 0./0. ; }
      // Recopie de l'ancienne région.
      if ( nombre > 0 ) {
        for ( size_t i = 0 ; i < nombre ; ++ i ) {
          nouveau[i] = tableau[i] ; }
        delete tableau ; }    // On rend au système d'exploitation l'ancienne région.
      tableau = nouveau ;  // Le pointeur désigne maintenant la nouvelle région.
      max += QUANTUM ;
    }
    tableau[nombre] = tmp ;
    ++ nombre ;
  }
  if ( nombre == 0 ) {
    cerr << "Aucun nombre lu dans le fichier." << endl ;
    return 0./0. ; }
  sort(tableau, tableau+nombre) ;
  double valeur ;
  if ( nombre % 2 == 0 ) {
    // Interpolation linéaire si l'effectif est pair.
    valeur = .5*tableau[nombre/2-1] + .5*tableau[nombre/2] ; }
  else {
    valeur = tableau[nombre/2] ; }
  delete tableau ; // Ne pas oublier de rendre la région utilisée.
  return valeur ;
}
double
mediane_3(const string & nom)
```

```
88 {
89   ifstream fichier(nom.c_str());
90   if ( ! fichier ) {
91     cerr << "Impossible d'ouvrir en lecture '" << nom << "'." << endl;
92     return 0./0.; }
93   typedef vector<double> Vecteur;
94   // 'vecteur' est construit en appelant l'itérateur d'entrée de 'double' du fichier
95   // en lecture 'fichier'. La syntaxe 'Vecteur vecteur(..., ...);' ne convient pas;
96   // il s'agit de la  déclaration de la fonction 'vecteur' qui retourne un 'Vecteur'.
97   // Pour faire plus simple mais moins concis, on aurait pu coder :
98   //       Vecteur vecteur;
99   //       double tmp;
100  //       while ( fichier >> tmp ) {
101  //         vecteur.push_back(tmp); }
102  Vecteur vecteur((istream_iterator<double>(fichier)), istream_iterator<double>());
103  if ( vecteur.empty() ) {
104    cerr << "Aucun nombre lu dans le fichier." << endl;
105    return 0./0.; }
106  sort(vecteur.begin(), vecteur.end());
107  if ( vecteur.size() % 2 == 0 ) {
108    // Interpolation linéaire si l'effectif est pair.
109    return .5*vecteur[vecteur.size()/2-1] + .5*vecteur[vecteur.size()/2]; }
110  else {
111    return  vecteur[vecteur.size()/2]; }
112 }
113 int
114 main()
115 {
116  // Test de l'appel de la fonction sur un fichier inexistant.
117  double mediane = mediane_1("fichier inexistant");
118  // La fonction '_isnan(...)' est une extension ; ce n'est donc pas portable.
119  if ( _isnan(mediane) ) {
120    cout << "La médiane n'a pas pu être calculée." << endl; }
121  else {
122    cout << "La médiane est " << mediane << '.' << endl; }
123  // Illustration ; le fichier est constitué de la séquence 0, 1, ..., 10.
124  cout << "La médiane est " << mediane_1("médiane.txt") << '.' << endl;
125  cout << "La médiane est " << mediane_2("médiane.txt") << '.' << endl;
126  cout << "La médiane est " << mediane_3("médiane.txt") << '.' << endl;
127  return 0; // Retour au système d'exploitation avec le code '0'.
128 }
```

Quand ce programme est exécuté, on obtient la sortie suivante.

```
1 La médiane n'a pas pu être calculée.
2 La médiane est 5.
3 La médiane est 5.
4 La médiane est 5.
```

9. — Les tableaux associatifs de la STL

Souvent, le programmeur souhaite disposer d'un conteneur en forme de « tableau associatif », encore appelé « dictionnaire ». Les éléments dans un tel conteneur sont des paires (clef, valeur). La clef est unique, elle permet d'accéder à la valeur. La syntaxe, pour utiliser ce conteneur, est de la forme « ***conteneur[clef]*** » : on reprend la syntaxe de l'utilisation d'un tableau mais l'indice, au lieu d'être un entier naturel, est une variable d'un type quelconque. La clef est très souvent une chaîne de caractères (une variable de type ***string***) ; c'est en cela que ce conteneur est aussi appelé « dictionnaire ». Il faut que l'opérateur « **<** » soit défini pour les clefs : le conteneur trie, implicitement, les clefs pour pouvoir ensuite retrouver une valeur particulière.

La STL nomme ces conteneurs des « maps » (correspondances en français). Un conteneur est déclaré par la syntaxe « ***map<TypeClef, TypeValeur> mon_tableau_associatif;*** ». Ensuite, la syntaxe « ***mon_tableau_associatif[ma_clef]*** » désigne la référence, dans le couple (clef, valeur) désigné, de la variable *valeur*. Si le couple désigné n'existe pas, la clef est utilisée pour créer un nouveau couple dont la valeur est la valeur par défaut créée en appelant le constructeur sans argument de ***TypeValeur***.

Dans le programme suivant, on veut pour chaque étudiant calculer, à partir d'un nombre de notes variable, sa moyenne. On utilise un tableau associatif dont la clef est le prénom de l'étudiant – on suppose ce prénom unique. La valeur est un objet de type ***NombreEtSomme*** : pour chaque étudiant, il suffit d'enregistrer son nombre de notes et la somme de ces notes. Il faut absolument définir le constructeur par défaut ; le constructeur par défaut offert par le compilateur ne convient pas : il n'initialise pas les deux attributs ***nombre_*** et ***somme_***.

Notons combien l'utilisation de ces tableaux associatifs est puissante et parcimonieuse. La ligne de code

mon_dict[nom].nouvelle_note(note);

veut dire

1. tout d'abord, si la clef ***nom*** n'existe pas dans ***mon_dict*** alors créer une nouvelle instance de ***NombreEtSomme*** en utilisant le constructeur par défaut (celui qui initialise ***nombre_*** et ***somme_*** à zéro) ;

2. ensuite, appeler la méthode ***nouvelle_note()*** sur l'instance de ***NombreEtSomme*** repérée par la clef ***nom*** en utilisant l'argument ***note***;
3. enfin, exécuter cette méthode et donc incrémenter l'attribut ***nombre_*** et mettre à jour l'attribut ***somme_***.

Le code du programme figure ci-après.

```
1  #include <iostream> // cin, cout et cerr.
2  #include <vector>   // Vecteurs de la STL.
3  #include <map>      // Tableaux associatifs de la STL.
4
5  using namespace std ;
6
7  class NombreEtSomme {
8    size_t nombre_ ;  // Nombre de notes déjà enregistrées.
9    double somme_ ;  // Somme des notes déjà enregistrées.
10 public :
11   NombreEtSomme () : nombre_(0), somme_(0) {} // Constructeur par défaut.
12   size_t nombre () const { return nombre_ ; }      // Méthode qui retourne le nombre de notes.
13   double somme () const { return somme_ ; }        // Méthode qui retourne la somme des notes.
14   void nouvelle_note (const double & arg) {        // Méthode qui ajoute une nouvelle note.
15     ++ nombre_ ;
16     somme_ += arg ; }
17 } ;
18
19 int
20 main()
21 {
22   typedef map<string, NombreEtSomme> Dictionnaire ; // Pour simplifier la syntaxe
23   Dictionnaire mon_dict ;
24   string nom ; double note ;
25   while ( cin >> nom >> note ) {
26     mon_dict[nom].nouvelle_note(note) ; }
27   // Affichage des résultats.
28   cout << "Étudiant\tNombre\tSomme\tMoyenne" << endl ;
29   for ( Dictionnaire::iterator i = mon_dict.begin() ; i != mon_dict.end() ; ++ i ) {
30     cout << i->first << '\t' << i->second.nombre() << '\t' << i->second.somme() <<
31       '\t' << i->second.somme()/i->second.nombre() << endl ; }
32   return 0 ; // Retour au système d'exploitation avec le code '0'.
33 }
```

Le fichier de données utilisé pour exécuter ce programme est le suivant.

```
1  Aristide    10
2  Béatrice    11
3  Aristide    9
```

Le résultat est alors le suivant.

```
1  Étudiant    Nombre  Somme  Moyenne
2  Aristide    2       19     9.5
3  Béatrice    1       11     11
```

Un itérateur est utilisé pour afficher les résultats. Un itérateur permet de parcourir les éléments d'un conteneur en les désignant tour à tour. Le premier élément d'un conteneur est désigné par la syntaxe « ***conteneur.begin()*** ». L'élément virtuel après le dernier élément est désigné par « ***conteneur.end()*** ». L'auto-incrémentation est l'opérateur qui permet de désigner l'élément suivant dans le conteneur. Le parcours de tous les éléments d'un conteneur est ainsi obtenu au moyen de la boucle ***for*** suivante.

for (Conteneur::iterator i = conteneur.begin() ; i != conteneur.end() ; ++ i)

Enfin, un élément est désigné par la syntaxe « ****i*** ». La syntaxe « ***i->méthode()*** » est un raccourci pour désigner « ***(*i).méthode()*** ». La syntaxe « ***i->second.nombre()*** » est donc équivalente à « ***((*i).second) . nombre()*** ».

10. — Les multimaps

Le conteneur ***map*** exige que la clef soit unique. Un conteneur qui associe à une clef plusieurs valeurs est un ***multimap***. Ce conteneur est déclaré par la syntaxe « ***multimap<TypeClef, TypeValeur> mon_multi_map ;*** ». La syntaxe « ***conteneur[clef]*** » ne convient pas pour un ***multimap***. Il faut utiliser les méthodes ***insert(...)*** pour insérer un élément et ***equal_range(...)*** pour retrouver les différents éléments qui ont la même clef.

Dans l'exemple suivant, j'utilise un ***multimap*** pour gérer un répertoire téléphonique dont la clef est le prénom et les valeurs les différents numéros de téléphone des gens.

```
1  #include <iostream> // cin, cout et cerr.
2  #include <string>   // Chaînes de caractères.
3  #include <map>      // Tableaux associatifs de la STL -- map et multimap.
4
5  using namespace std ;
6
7  typedef multimap<string, string> MultiDict ; // Pour simplifier la syntaxe.
8  MultiDict mon_multi_dict ;                   // Répertoire téléphonique.
9
10 // Insertion d'une nouvelle entrée dans le répertoire.
```

```
11 void
12 insertion(const string & nom, const string & num_tel)
13 {
14   // Pour les multimaps, on ne peut pas coder ‘mon_multi_dict[nom] = num_tel ;’ ;
15   // Il faut utiliser ‘mon_multi_dict.insert(make_pair(nom, num_tel)) ;’.
16   mon_multi_dict.insert(make_pair(nom, num_tel)) ;
17 }
18 int
19 main()
20 {
21   insertion("Anne", "01 23 45 67 89") ;
22   insertion("Anne", "06 23 45 67 89") ;
23   insertion("Bernard", "01 98 76 54 32") ;
24   insertion("Bernard", "06 98 76 54 32") ;
25   while ( true )
26   {
27     cout << "Nom (Ctrl-C pour terminer) ? " ;
28     string ligne ;
29     if ( ! getline(cin, ligne) ) {
30       break ; }
31     pair<MultiDict::iterator, MultiDict::iterator> r = mon_multi_dict.equal_range(ligne) ;
32     if ( r.first == r.second ) {
33       cout << "Nom absent du répertoire." << endl ; }
34     else {
35       for ( MultiDict::iterator i = r.first ; i != r.second ; ++ i ) {
36         cout << i->first << ’\t’ << i->second << endl ; } }
37   }
38   return 0 ; // Retour au système d’exploitation avec le code ‘0’.
39 }
```

Ci-après, je figure une hypothétique session d’utilisation du programme.

```
Nom (Ctrl-C pour terminer) ? Anne
Anne 01 23 45 67 89
Anne 06 23 45 67 89
Nom (Ctrl-C pour terminer) ? ^C
```