9

Entrées et sorties

Avec ce chapitre, nous entamons une partie beaucoup plus sérieuse, qui va nous permettre d'écrire de vrais programmes. Jusqu'à maintenant, toutes les entrées se faisaient sur le clavier et toutes les sorties sur la console. Nous allons à présent examiner comment lire et écrire des fichiers qui se trouvent sur le disque local. Ces fichiers peuvent contenir du texte que nous pourrions lire avec un éditeur traditionnel, des documents avec un format particulier ou encore des fichiers totalement binaires comme des exécutables ou des images, qui sont traités par des programmes spécifiques. Nous irons même lire, en Java, un document HTML sur un site Web qui pourrait se situer à l'extérieur de notre environnement. Nous allons examiner, en première partie de ce chapitre, un cas bien particulier : un fichier texte délimité.

Ce sujet va aussi nous permettre d'entrer dans la jungle des classes C++ et Java qui traitent des entrées et sorties. Un débutant en C++ ou en Java risque de se retrouver perdu s'il essaie de considérer chaque classe indépendamment et s'il essaie de comprendre son usage et son utilité. En Java notamment, c'est encore plus complexe, car parfois plusieurs classes doivent être associées en parallèle. Un exercice pourrait consister à écrire un exemple par classe, mais cela prendrait vite la place d'un ouvrage substantiel. Le meilleur moyen de s'en sortir est encore de se poser les deux questions suivantes :

- Est-ce que nous lisons ou nous écrivons ?
- Est-ce un fichier binaire ou simplement du texte ?

Lorsque nous avons répondu à ces deux questions, il suffit alors de consulter le sommaire de ce chapitre, de sauter au paragraphe approprié et d'examiner le code associé. Cette manière de faire peut sembler bizarre, mais elle a le mérite d'être pragmatique et efficace pour s'y retrouver dans cette jungle.

Avant de passer aux cas d'étude, nous allons présenter dans ce tableau les différents exemples que nous avons choisis de traiter :

Tableau 9-1 Les combinaisons de lecture et d'écriture

	Lecture texte	Lecture binaire	Écriture texte	Écriture binaire
C++	Lecture d'un fichier Access délimité.	Commande Linux `strings` : extraction de chaînes visibles.	Information structurée au format XML.	100 octets binaires au hasard.
Java	Lecture d'un fichier Access délimité.	Chargement d'une sauvegarde du jeu d'Othello.	Sauvegarde d'une partie du jeu d'Othello.	Sauvegarde d'une partie du jeu d'Othello.

Comme extra nous avons ajouté la lecture en Java d'une page Web (HTML). Une page Web peut aussi être considérée comme un fichier texte. Cependant, elle n'est pas accessible sur le disque de notre PC, mais sur Internet.

Du texte délimité à partir de Microsoft Access

Access est le fameux outil de bases de données Microsoft. Si nous ne le possédons pas, ce n'est pas très important, car le fichier texte que nous allons créer peut être construit à la main.

Au chapitre 4, nous avons créé notre première classe nommée `Personne`, qui contenait un nom, un prénom et une année de naissance. Nous allons donc créer une table dans Access qui contient ces trois champs. Nous pouvons prendre, par exemple, `bd1.mdb` comme nom de base de données, `Personnes` comme nom de table et `Nom`, `Prenom` et `Annee_Naissance` comme noms de champs. `Nom` et `Prenom` sont des champs texte alors que `Annee_Naissance` sera de type numérique. Nous définirons sur cette table une clé primaire nommée `Numero`. Finalement, nous entrerons les enregistrements suivants :

Tableau 9-2 Notre table Microsoft Access

Nom	Prenom	Annee_Naissance
Haddock	Capitaine	1907
Kaddock	Kaptain	1897

Lorsque la table aura été enregistrée, nous allons l'exporter (Fichier > Enregistrer sous > Exporter) en tant que fichier texte avec le nom `Personnes.txt` après avoir choisi le format délimité avec le point-virgule comme séparateur. Le fichier texte peut être lu par un éditeur de texte traditionnel comme Crimson :

```
1;"Haddock";"Capitaine";1907
2;"Kaddock";"Kaptain";1897
```

Ce qui est intéressant ici, c'est de constater que le fichier `Personnes.txt` peut aussi être créé et modifié par un traitement de texte ou par un programme, comme nous allons le

voir ici. Le nouveau fichier `Personnes.txt` peut très bien être importé après modification dans **Access** de Microsoft, ce qui peut se faire avec le menu Importer (Fichier > Données externes > Importer). Il est évident que cette méthode n'est pas appropriée s'il fallait changer continuellement des données. Il faudrait alors travailler directement avec un serveur SQL ou une interface ODBC, ce que nous verrons au chapitre 20.

Lecture de fichiers texte en C++

Nous allons utiliser à présent la classe `ifstream` pour lire notre fichier `Personnes.txt`. Cette classe fait partie de la bibliothèque `iostream` du C++ traditionnel. Nous aurions pu utiliser des fonctions C comme `open()` ou `fopen()`, mais elles ne supportent pas les opérateurs << ou >>, qui sont devenus les outils traditionnels du C++. Nous allons donc les laisser aux oubliettes, comme d'ailleurs toutes les fonctions C qui peuvent être avantageusement remplacées par des méthodes de classe de la bibliothèque du Standard C++, lorsque celles-ci sont disponibles. Nous allons directement écrire le code pour exécuter cette tâche, puis passer aux explications. Cette méthode de présentation est plus directe et se retrouvera tout au long de cet ouvrage.

```
// lecture_texte.cpp
#include <string>
#include <iostream>
#include <fstream>

using namespace std;

class Lecture_texte {
 private:
   string nom_fichier;

 public:
   Lecture_texte(const string le_fichier);
   void lecture();
};

Lecture_texte::Lecture_texte(const string le_fichier)
              : nom_fichier(le_fichier) {
}

void Lecture_texte::lecture() {
  ifstream ifichier(nom_fichier.c_str());

  if (!ifichier) {
    cerr << "Impossible d'ouvrir le fichier " << nom_fichier << endl;
    return;
  }

  string une_ligne;
  int numero_de_ligne = 1;
```

```
  while (ifichier >> une_ligne) {
    cout << numero_de_ligne << ":  " << une_ligne << endl;
    numero_de_ligne++;
  }
}

int main() {
  Lecture_texte les_personnes("Personnes.txt");
  les_personnes.lecture();
}
```

Le `<fstream>` fait son apparition. C'est dans ce fichier d'en-tête que la classe `ifstream` est définie. Le constructeur d'`ifstream` s'attend à recevoir un pointeur à une chaîne de caractères correspondant au nom du fichier. Ce dernier pourrait contenir le chemin d'accès complet du fichier. La méthode `c_str()` de la classe `string` fait le travail. La construction :

```
if (!ifichier) {
```

est un peu particulière. Elle détermine en fait si le fichier `Personnes.txt` est effectivement accessible. La boucle :

```
while (ifichier >> une_ligne) {
```

nous permet de lire ligne par ligne notre fichier `Personnes.txt`. Une ligne de texte dans un fichier se termine par un `\n` ou `\r\n`, mais ces derniers ne sont pas copiés dans le `string une_ligne`, car ils sont filtrés par l'opérateur `>>`. Lorsque la fin du fichier est atteinte, la boucle se terminera.

Note

`\n` (octal 012) termine en général une ligne de texte édité ou construit sous Linux.

La séquence `\r \n` (octal 015 et 012) est applicable dans notre cas, car le fichier est préparé et enregistré sous DOS.

Le programme ci-dessus nous sortira le résultat suivant :

```
1:  1;"Haddock";"Capitaine";1907
2:  2;"Kaddock";"Kaptain";1897
```

qui sera identique au programme Java qui va suivre. Il est important de noter que si une ligne vide est ajoutée en fin de fichier, elle sera traitée comme un enregistrement vide par Access de Microsoft en cas d'importation. Le `7` de `1897` devra être le dernier caractère du fichier.

La méthode getline()

Que se passerait-il avec le programme précédent si les données contenaient des espaces ? Ou, en d'autres mots, si notre fichier `Personne.txt` contenait les données suivantes :

```
1;"Haddock";"Le Capitaine";1907
2;"Kaddock";"Kaptain";1897
```

Il suffit de l'essayer pour constater que le résultat est loin de nous satisfaire :

```
1:  1;"Haddock";"Le
2:  Capitaine";1907
3:  2;"Kaddock";"Kaptain";1897
```

Le problème vient de l'instruction suivante :

```
while (ifichier >> une_ligne) {
```

Si une espace ou autre tabulateur est découvert dans le fichier, l'opérateur >> stoppera son transfert. Le nom de la variable n'est vraiment pas approprié, et `une_ligne` devrait plutôt être nommée `un_mot`. En fait, les points-virgules (;) ne sont pas considérés comme des séparateurs par l'opérateur >> de la classe `istream`. Pour que notre programme fonctionne avec `"Le Capitaine"`, il nous faudrait modifier le code comme suit :

```
// lecture_texte2.cpp
#include <string>
#include <iostream>
#include <fstream>

using namespace std;

class Lecture_texte {
 private:
   string nom_fichier;

 public:
   Lecture_texte(const string le_fichier);
   void lecture();
};

Lecture_texte::Lecture_texte(const string le_fichier)
              : nom_fichier(le_fichier) {
}

void Lecture_texte::lecture() {
  ifstream ifichier(nom_fichier.c_str());

  if (!ifichier) {
    cerr << "Impossible d'ouvrir le fichier " << nom_fichier << endl;
    return;
  }

  char une_ligne[1024];
  int numero_de_ligne = 1;

  while (ifichier.getline(une_ligne, 1024)) {
    cout << numero_de_ligne << ":  " << une_ligne << endl;
    numero_de_ligne++;
  }
}
```

```
int main() {
  Lecture_texte les_personnes("Personnes2.txt");
  les_personnes.lecture();
}
```

Nos données sont lues du fichier `Personnes2.txt` et le résultat sera correct :

```
1:  1;"Haddock";"Le Capitaine";1907
2:  2;"Kaddock";"Kaptain";1897
```

La différence vient de notre :

```
while (ifichier.getline(une_ligne, 1024))
```

qui va lire une ligne entière jusqu'à un maximum de 1 024 caractères. Le `getline()` va en fait s'arrêter dès qu'il trouve un `\n` ou un `\r`, qui correspond à la fin de la ligne. Le `while()` se terminera aussi dès que la fin du fichier sera atteinte.

Pour terminer, il nous faut absolument indiquer une autre construction :

```
char un_char;
ifichier.get(un_char);
cout << "Un caractère: " << un_char << endl;
```

`get()` est une méthode de la classe `istream` (classe de base d'`ifstream`) qui peut être utile dans des situations où il se révèle nécessaire de lire caractère par caractère. Nous l'utiliserons dans l'exercice 2 avec son équivalent `put()` pour l'écriture dans la classe `ostream` (classe de base d'`ofstream`) :

```
ofstream ofichier(…);
char onechar = …;
ofichier.put(onechar);
```

Lecture de fichiers texte en Java

Comme pour le programme ci-dessus, qui lit notre fichier délimité `Personne.txt` exporté depuis Microsoft Access, la version Java (`LectureTexte.java`) s'écrira de cette manière :

```
import java.io.*;

public class LectureTexte {
 private String nom_fichier;

 public LectureTexte(String le_fichier) {
   nom_fichier = le_fichier;
 }

 public void lecture() throws Exception {
   String une_ligne;
   BufferedReader in = new BufferedReader(new FileReader(nom_fichier));
   int numero_de_ligne = 1;
```

```
    for (;;) {
      une_ligne = in.readLine();
      if (une_ligne == null)  break;
      System.out.println(numero_de_ligne + ":  " + une_ligne);
      numero_de_ligne++;
    }
  }

  public static void main(String[] args) throws Exception {
    LectureTexte les_personnes = new LectureTexte("Personnes.txt");
    les_personnes.lecture();
  }
}
```

et nous obtenons le même résultat qu'en C++.

La première remarque doit se faire sur ce `throws Exception`, car nous avons ici décidé de rejeter toutes les exceptions qui peuvent être générées dans la méthode `lecture()`. Le code serait cependant plus robuste si nous retournions, par exemple, un `boolean` de la méthode `lecture()`.

La classe `BufferedReader` possède une méthode `readline()` qui permet de lire un tampon de lecture ligne par ligne, de la même manière que notre exemple en C++. Le `Buffered Reader` est associé à un fichier ouvert en lecture au travers de la classe `FileReader`. Une question se pose immédiatement : que se passe-t-il si notre fichier `Personnes.txt` n'existe pas ? Lorsque le constructeur de `FileReader` ne peut ouvrir ce fichier, il va générer une exception nommée `FileNotFoundException`. Ici, nous utilisons un mécanisme particulier pour ignorer les exceptions : la combinaison `throws Exception`. Ces exceptions seront ignorées non pas à l'exécution, mais seulement dans le code où les séquences de `try` et `catch` ne sont pas nécessaires.

Utilisation de la variable separatorChar

Nous devons aussi dire quelques mots sur le caractère de séparation des répertoires. Ce petit programme :

```
import java.io.*;
public class FichierSeparateur {
  public static void main(String[] args) throws Exception {
    File mon_fichier = new File("Personnes.txt");

    System.out.println("Chemin: " + mon_fichier.getAbsolutePath());
    System.out.println("Séparateur: " + File.separatorChar);
  }
}
```

nous donnera :

```
Chemin: E:\JavaCpp\EXEMPLES\Ch06\Personnes.txt
Séparateur: \
```

Cela nous indique la location du fichier sur le disque. Le séparateur sera différent sous Linux (/), et attention : la variable statique `File.separatorChar` doit être absolument utilisée si nous devons accéder ou rechercher un répertoire dans le chemin complet et bénéficier d'un programme qui fonctionne aussi sous Linux.

Si nous déplaçons notre fichier `Personne.txt` dans un sous-répertoire `xxx`, les deux formes suivantes sont acceptables aussi bien sous Windows que sous Linux :

```
File mon_fichier = new File("xxx/Personnes.txt");
File mon_fichier = new File("xxx\\Personnes.txt");
```

Les deux `\\` sont nécessaires, car le premier est le caractère d'échappement pour le suivant. Si nous voulions rester totalement portables entre différents systèmes, une instruction telle que :

```
String fichier_complet = System.getProperty("user.dir")
                         + File.separator + "Personnes.txt";
```

serait souhaitable. La propriété `user.dir`, qui est reconnue par toutes les machines virtuelles de Java, nous donne le répertoire courant.

Lecture de fichiers sur Internet en Java

Lire des fichiers sur Internet en C++ ne sera pas traité dans cet ouvrage car accéder à Internet en C++ nécessite des bibliothèques spécialisées qui sont spécifiques à la machine et au système d'exploitation. En Java, cette fonctionnalité est extrêmement simple. Lire un fichier sur le disque local ou sur Internet est tout à fait similaire.

Pour cet exercice, nous avons choisi un fichier HTML très compact, que nous avons nommé `mon_test.html` et installé sur un serveur Web Apache (`http://www.apache.org`) sur notre machine. L'adresse URL est donc `http://localhost/mon_test.html` et apparaît de cette manière avec Firefox ou Microsoft Internet Explorer :

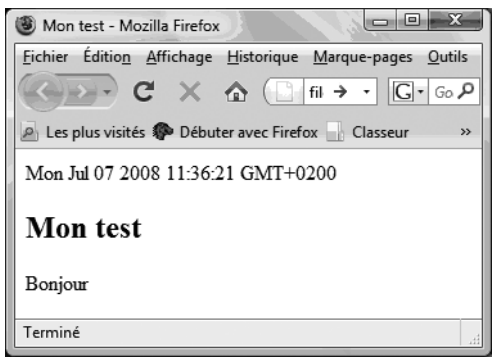

Figure 9-1

Page Web mon_test.html avec Firefox

```
Sun Oct 31 11:18:35 UTC+0100 1999
Mon test
Bonjour
```

Pour ceux qui ne désirent pas installer un serveur Apache, il est tout à fait possible de spécifier une autre adresse URL en utilisant la forme :

```
file:///C:/JavaCpp/EXEMPLES/Chap09/mon_test.html
```

sous Firefox qui correspond à un fichier sur le disque local. Le code Java simplifié (`LectureUrl.java`) se présente ainsi :

```
import java.net.*;
import java.io.*;

public class LectureUrl {
  public static void main(String[] args) throws Exception {
    URL apache_local = new URL("http://localhost/mon_test.html");

    BufferedReader in = new BufferedReader(new
                          InputStreamReader(apache_local.openStream()));

    String inputLine;
    while ((inputLine = in.readLine()) != null) {
      System.out.println(inputLine);
      in.close();
    }
}
```

et le résultat sera le suivant :

```
<html>
<head><title>Mon test</title></head>
<body>
<script language="JavaScript">document.write(new  Date());</script>
<h2>Mon test</h2>
Bonjour
</body>
</html>
```

Si nous utilisons la forme fichier, il nous faudra ajouter sous Windows une série de `\\`, car le caractère `\` doit être entré avec son caractère d'échappement :

```
URL apache_local = new URL("file:\\\\\\C:\\JavaCpp\\EXEMPLES \\Ch09\\mon_test.html");
```

En ce qui concerne le code, nous retrouvons la classe `BufferedReader`, que nous utilisons cette fois-ci sur un `InputStreamReader`. Cette dernière classe va nous offrir une transparence complète au niveau du protocole HTTP, protocole nécessaire pour lire un document sur le réseau Internet.

Dans l'annexe E, nous présentons NetBeans, environnement qui permet de développer des applications Web. Le serveur http Apache Tomcat est intégré à NetBeans.

Lecture de fichier binaire en C++

Il nous est parfois nécessaire de devoir extraire d'un fichier binaire, par exemple d'un programme exécutable, la partie visible de son contenu, c'est-à-dire tout ce qui correspond à du texte bien visible (caractères ASCII entre l'espace et le caractère binaire 127 (0x7F)). Un outil Linux, **strings**, existe d'ailleurs et s'avère souvent utilisé avec d'autres filtres pour rechercher des chaînes de caractères bien particuliers.

Pour ce faire, nous allons lire le fichier par bloc et ne sortir sur la console que les parties qui excèdent cinq caractères. Chaque séquence de caractères visibles sera précédée de sa position dans le fichier, ceci entre parenthèses et chaque fois sur une nouvelle ligne.

```
// strings.cpp
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(int argc, char **argv) {
 if (argc != 2) {
   cerr << "Nombre d'arguments invalides" << endl;
   cerr << "Strings Fichier" << endl;
   return -1;
 }

 ifstream infile(argv[1], ios::in|ios::binary);
 if (!infile) {
   cout << "Fichier d'entrée n'existe pas";
   return -1;
 }

 const int bloc   = 512;  // dim bloc lu
 int   avant_bloc = 0;    // dim * n lu
 int   ncinq_char = 0;    // nombre de premiers chars lus
 char  cinq_char[6];      // cinq premiers chars

 cinq_char[5] = 0;        // toujours 5 imprimés

 char tampon[bloc];
 int  octets_lus;
 for (;;) {                      // lecture par bloc
   infile.read(tampon, bloc);
   octets_lus = infile.gcount();
   for (int i = 0; i < octets_lus; i++) {
     if ((tampon[i] >= ' ') && (tampon[i] < 127)) {
       if (ncinq_char < 5) {
         cinq_char[ncinq_char] = tampon[i];
         ncinq_char++;
       }
```

```
        else {
          if (ncinq_char == 5) {
            cout << (avant_bloc + i - 5) << ": " << cinq_char << tampon[i];
            ncinq_char++;
          }
          else  cout << tampon[i];
        }
      }
      else {  // char binaire
        if (ncinq_char > 5) {
          cout << endl;
        }
        ncinq_char = 0;
      }
    }
    if (octets_lus < bloc)  break;   // dernier bloc
    avant_bloc += bloc;              // avance compteur relatif
  }

  infile.close();
}
```

La constante `bloc` a pour valeur 512, et ce n'est pas par hasard. Lorsque nous devons lire des fichiers binaires, il n'y a pas de restrictions ni d'obligations de lire les octets l'un après l'autre. L'opérateur >> en C++ de la classe `ifstream` ou le `readline()` en Java, que nous avons vus tous deux précédemment, possèdent en fait un test interne sur chaque caractère pour identifier une fin de ligne. Ici, ce n'est pas nécessaire, et nous choisirons des dimensions de blocs suffisamment grandes. Si nous avons choisi une dimension de 512 et non pas de 100, de 1 000, de 511 ou de 513, cela vient sans doute de vieilles habitudes de l'auteur, qui considère encore qu'il est avantageux de choisir une dimension équivalant ou correspondant à un nombre entier de blocs sur un secteur du disque. À un moment ou à un autre, les primitives du système devront bien accéder au disque ou à son cache ! Ce qui est important ici, c'est le nombre d'appels de `infile.read()` qui se fera selon cette formule :

```
(dimension du fichier)/(dimension du bloc (512 ici)) + 1
```

Si nous lisons le fichier octet par octet, nous multiplierons ce nombre par 512. C'est de cette manière que nous obtiendrons les plus mauvaises performances. Choisir des blocs correspondant à la dimension du fichier n'est pas non plus un choix sensé, car certains fichiers pourraient dépasser les capacités de mémoire vive ou virtuelle. Si nous devions travailler avec des fichiers plus importants, il serait avantageux d'augmenter le bloc à 4 096, par exemple, ce qui n'entamerait pas trop les ressources en mémoire du programme. Cette discussion correspond aux calculs de performance que nous effectuerons au chapitre 15.

Nous lisons donc par blocs de 512, ce qui limite le nombre d'accès aux primitives qui vont finalement accéder au disque, à travers le système d'exploitation, et au contrôleur du disque. Ensuite, chaque caractère sera lu de la mémoire et donc vérifié plus rapidement. Le tampon `cinq_char` garde une suite consécutive de caractères qui sont présentés à l'écran

lorsque le sixième apparaît. Dans le cas contraire, nous remettons le compteur `ncinq_char` à zéro et recommençons. Cette manière de faire nous permet de sortir des séquences de caractères même s'ils se trouvent à cheval sur deux blocs de 512 octets. Sinon il faudrait avoir deux blocs `tampon` de travail, garder leurs index respectifs et aller de l'un à l'autre.

Écriture d'un fichier binaire en C++

Nous avons choisi ici un exemple un peu particulier, guère vraisemblable, mais qui nous permettra d'utiliser des fonctions pouvant poser problème avec d'autres compilateurs : c'est en effet un aspect important qu'un programmeur C++ doit absolument maîtriser. Nous allons écrire un fichier binaire de 100 octets avec des valeurs totalement aléatoires.

```
// ecritbin.cpp
#include <iostream>
#include <fstream>
#include <ctime>
#include <cstdlib>

using namespace std;

const int dim = 100;

int main(int argc, char* argv[])
{
  ofstream outfile("au_hasard.bin", ios::out | ios::binary);
  if (!outfile) {
    cerr << "Fichier de sortie ne peut être créé" << endl;
    return false;
  }

  char tampon[dim] ; // nos valeurs aléatoires
  srand(time(NULL));

  for (int i = 0; i < dim; i++) {
    tampon[i] = (char)((rand() * 256)/RAND_MAX);  // entre 0 et 255
    //tampon[i] = (char)rand(256);  // entre 0 et 255 certains compilateurs
  }

  outfile.write(tampon, dim) << endl;
  outfile.close();

  return 0;
}
```

La fonction C `srand()` va définir un point de départ pour le générateur de nombres aléatoires assuré par la fonction `rand()`. Pour ce faire, nous utiliserons l'heure actuelle en secondes, qui nous fournira des résultats différents à chaque utilisation du programme. Le fichier `au_hasard.bin` va contenir 100 octets de données aléatoires. La fonction `rand()` peut avoir différentes implémentations suivant les compilateurs : il vaut donc mieux consulter

attentivement la documentation ou plus simplement faire un test préalable. Nous comprendrons ainsi ce que nous retourne `rand()`, puisqu'il faut ensuite le multiplier par `256/RAND_MAX`. Nous verrons les détails un peu plus loin.

Sous Windows, nous pouvons associer le `.bin` à un éditeur hexadécimal pour visionner le résultat. Les trois indicateurs `ios` doivent être inclus (avec l'opérateur logique OU (`|`)) pour indiquer que :

- `ios::out` – fichier d'écriture ;
- `ios::binary` – fichier binaire.

Nous trouvons la définition de ces bits dans le fichier d'en-tête `streambuf.h`, qui est lui-même inclus dans `iostream`. Les `out`, `binary` et autres `noreplace` sont des énumérations publiques de la classe `ios`, et il y a parfois des variantes selon les implémentations de la bibliothèque du Standard C++ et les compilateurs.

Dans le code précédent, le fichier `au_hasard.bin` est écrit à chaque exécution du programme. Si nous ne voulions pas le remplacer, il faudrait s'assurer au préalable qu'il existe bien.

La classe `ofstream` est extrêmement simple à utiliser. De la même manière qu'`ifstream`, le `(!outfile)` nous permet de vérifier si l'ouverture du fichier a été faite correctement. Le `rand(256)` nous retourne un entier entre 0 et 255, c'est-à-dire une des valeurs possibles pour représenter un octet. Notre tampon de 100 caractères est donc du type `char`. Certains nous diront qu'un `unsigned char` (0 à 255) aurait été plus approprié qu'un char (–128 à 127). Cependant, le transtypage fonctionne correctement, et, pour être convaincus, nous pouvons le vérifier avec un éditeur hexadécimal, qui devrait nous montrer qu'il y a bien des octets avec le bit 7 correctement positionné. Le `rand(256)` devrait d'ailleurs nous allouer environ la moitié de l'ensemble !

La méthode `write()` de la classe `ofstream` a besoin ici de deux paramètres, le `tampon` (un `char *`) et le nombre d'octets à écrire (`dim`). C'est une bonne habitude de fermer correctement le `stream` avec un `close()` en fin d'opération.

Compilation conditionnelle

Comme nous venons de le voir dans l'exemple ci-dessus, il y a parfois des circonstances où une compilation conditionnelle s'avère nécessaire. Ce peut être un problème de plate-forme (DOS ou Linux (Unix)) ou de compilateur ne supportant pas certaines fonctions, ou les supportant différemment. Le morceau de code suivant devrait nous aider à pouvoir contourner ce genre de difficulté avec des compilations conditionnelles.

```
// define.cpp
#ifdef GNU
#define GNU_UNIX
#endif

#ifdef UNIX
#define GNU_UNIX
#endif
```

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
#ifdef GNU
    cout << "Avec GNU" << endl;
#endif

#ifdef GNU_UNIX
    cout << "Avec GNU et Unix" << endl;
#endif

#ifndef DOS
    cout << "Sans DOS" << endl;
#endif

  cout << "Avec tous" << endl;
  return 0;
}
```

Si nous compilons ce code sans options de compilation, c'est-à-dire avec :

```
g++ -o define.exe define.cpp
```

nous obtiendrons :

```
Sans DOS
Avec tous
```

Dans ce code, nous découvrons les directives de compilation conditionnelle. Le message « Sans DOS » apparaît car la constante du préprocesseur n'a pas été définie. `#ifndef` veut donc dire : si DOS n'est pas défini, alors je compile le code jusqu'au prochain `#endif`, qui indique la fin d'une condition de compilation.

Si nous voulons à présent compiler avec la constante GNU, il faudra compiler le programme de cette manière :

```
g++ -DGNU -o define.exe define.cpp
```

ou bien :

```
g++ -DGNU -c define.o define.cpp
g++ -DGNU -o define.exe define.o
```

Ce qui nous donnera :

```
Avec GNU
Avec GNU et Unix
Sans DOS
Avec tous
```

Le `#define GNU_UNIX` permet de définir une nouvelle constante à l'intérieur du code source. Cela permet de ne pas avoir à recopier le même code pour des conditions différentes.

> **Note**
>
> Des constantes de compilations sont parfois utilisées pour du code de test durant les phases de développement. Lorsque le produit final est délivré, il suffit alors d'ajouter ou d'enlever la constante dans un `Makefile`.
>
> ```
> define.o:  define.cpp
> g++ $BTEST -c define.cpp
> ```

Nous avons déjà parlé de l'utilisation des `#define` au chapitre 6 pour la manipulation de fichiers d'en-tête multiples.

Écriture d'un fichier binaire en Java

Pour cette partie, nous allons revenir à notre jeu d'Othello (voir chapitres 5 et 20), dans lequel nous allons à présent sauvegarder et recharger une partie en cours (fichiers source `SauveOthello.java` et `ChargeOthello.java`).

Au chapitre 5, nous avions défini notre échiquier comme un `int jeu[10][10]`, avec quatre possibilités : 0 pour libre, 1 pour noir, 2 pour blanc et –1 pour les bords. Comme nous ne nous intéressons qu'à l'intérieur, un simple fichier de 64 octets (8×8) sera suffisant, et nous n'avons pas à traiter la valeur de –1 sur un octet de 8 bits.

Nous commencerons par l'écriture de notre fichier binaire, que nous allons nommer `othello.bin` :

```
import java.io.*;

public class SauveOthello {
  private String fichier;

  private static final int dim = 10;
  private int[][] othello = new int[dim][dim];

  public SauveOthello(String fichier_de_sauvetage) {
    fichier = fichier_de_sauvetage;
    for (int j = 1; j < dim-1; j++) {  // intérieur vide
      for (int i = 1; i < dim-1; i++) {
        othello[i][j] = 0;
      }
    }

    othello[4][5] = 1;  // blanc
    othello[5][4] = 1;  // blanc
    othello[4][4] = 2;  // noir
    othello[5][5] = 2;  // noir
```

```
  }

  public boolean sauve() {
    File outFile = new File(fichier);

    try {
      FileOutputStream out = new FileOutputStream(outFile);
      for (int j = 1; j < dim-1; j++) {
        for (int i = 1; i < dim-1; i++) {
          out.write(othello[i][j]);
        }
      }
      out.close();
    }
    catch (IOException e) {
      return false;
    }
    return true;
  }

  public static void main(String[] args) {
    SauveOthello mon_sauvetage = new SauveOthello("othello.bin");
    if (mon_sauvetage.sauve()) {
      System.out.println("Sauvegarde effectué");
    }
    else {
      System.out.println("Erreur de sauvegarde");
    }
  }
}
```

La première partie de ce code nous est déjà familière. Nous positionnons nos quatre pions de départ (voir chapitre 20 pour les règles du jeu) afin de vérifier le résultat produit. Pour ce qui est de l'écriture, nous allons essayer d'éclairer le lecteur sur la manière de procéder pour savoir quelles classes utiliser. Quelque part, il nous faut rechercher une méthode `write()`. Celle-ci doit pouvoir écrire des octets et non pas du texte ou des objets de taille variable.

`FileOutputStream` est la classe la plus élémentaire pour envoyer des données binaires dans un fichier au travers d'un flux. Elle possède des méthodes `write()` pour envoyer des octets, mais aussi des entiers. Elle reçoit aussi le nom du fichier de sortie comme paramètre de constructeur. `othello` étant un tableau d'entier, le fichier aura donc un format binaire déterminé par le type `int` du langage Java. Si nous devions lire `othello.bin` depuis un autre langage ou avec d'autres classes, nous pourrions rencontrer des difficultés. La séquence `try` et `catch()` devrait nous sortir les erreurs. Dans notre cas, il y a peu d'erreurs possibles. Si nous voulions provoquer une erreur, nous pourrions alors essayer d'écrire dans un fichier tel que `Z:\othello.bin`. Il faudrait alors modifier le code pour vérifier le type d'erreur et définir une action spécifique pour chaque type. Si nous écrivions :

```
    catch (IOException e) {
      System.out.println(e);
      return false;
    }
```

et vérifiions le résultat avec `Z:\othello.bin`, nous obtiendrions ceci :

```
java.io.FileNotFoundException: Z:\othello.bin (Le chemin d'accès spécifié est
➥introuvable)
Erreur de sauvegarde
```

Un bon traitement des erreurs nécessiterait donc un peu plus de code.

Lecture d'un fichier binaire en Java

Nous terminons avec la relecture (`ChargeOthello.java`) de notre fichier binaire `othello.bin` précédemment sauvegardé :

```
import java.io.*;

public class ChargeOthello {
  private String fichier;

  private static final int dim = 10;
  private int[][] othello = new int[dim][dim];

  public ChargeOthello(String fichier_de_sauvetage) {
    fichier = fichier_de_sauvetage;
  }

  public boolean charge() {
    File inFile = new File(fichier);

    try {
      FileInputStream in = new FileInputStream(inFile);
      for (int j = 1; j < dim-1; j++) {
        for (int i = 1; i < dim-1; i++) {
          othello[i][j] = in.read();
        }
      }
      in.close();
    }
    catch (IOException e) {
      return false;
    }
    return true;
  }

  public void test() {
    int i = 0;  // position horizontale
    int j = 0;  // position verticale
```

```
    for (j = 0; j < dim; j++) {
      for (i = 0; i < dim; i++) {
        if (othello[i][j] >= 0) System.out.print(" ");
        System.out.print(othello[i][j]);
      }
      System.out.println();
    }
  }

  public static void main(String[] args) {
    ChargeOthello mon_sauvetage = new ChargeOthello("othello.bin");
    if (mon_sauvetage.charge()) {
      System.out.println("Chargement effectué");
      mon_sauvetage.test();
    }
    else {
      System.out.println("Erreur de chargement");
    }
  }
}
```

Ici, nous faisons l'inverse : nous utilisons la classe `FileInputStream` et la méthode `read()`. Il n'y a aucun contrôle du format du fichier. Nous assumons le fait que le fichier contienne des entiers et possède le nombre correct de caractères. Il n'y a, à nouveau, aucun véritable traitement des erreurs. Pour vérifier que le programme lit correctement, il faudrait sans doute réutiliser la classe précédente et vérifier un plus grand nombre de possibilités que nos quatre premiers pions !

Comme les fichiers de sauvegarde du jeu d'Othello peuvent être réutilisés afin de tester notre jeu pendant la construction du programme, il est évident que l'utilisation d'un fichier binaire n'est pas intéressante. Il serait donc avantageux d'utiliser un format texte simple, afin de pouvoir modifier les sauvegardes avec un éditeur traditionnel.

Un format tel que :

```
VVVVVVVV
VVVVVVVV      V - position vide
VVVVBNVV
VVNNNVVV      B - pion blanc
VVVNBVVV
VVVVVVVV      N - pion noir
VVVVVVVV
VVVVVVVV
```

se comprendrait de lui-même. C'est ce que nous allons aborder à présent.

Écriture d'un fichier texte en Java

Nous allons continuer de jouer avec notre exemple d'Othello, représenté ci-dessous dans le code source `WriteOthello.java`, où les noirs viennent de jouer.

Le fichier produit, `othello.txt`, sera cette fois-ci un fichier de type texte traditionnel :

```
import java.io.*;

public class WriteOthello {
  private String nomFichier;

  private static final int dim = 10;
  private int[][] othello = new int[dim][dim];

  public WriteOthello(String fichierDeSauvetage) {
    nomFichier = fichierDeSauvetage;

    for (int j = 1; j < dim-1; j++) {  // intérieur vide
      for (int i = 1; i < dim-1; i++) {
        othello[i][j] = 0;
      }
    }

    othello[3][4] = 2;  // noir joue

    othello[4][4] = 2;  // noir
    othello[5][4] = 2;  // noir
    othello[4][5] = 2;  // noir
    othello[5][5] = 1;  // blanc
  }

  public boolean sauve() {
    try {
      PrintWriter out = new PrintWriter(new FileWriter(nomFichier));

      char pion;
      StringBuffer tampon = new StringBuffer("12345678");
      for (int j = 0; j < 8; j++) {
        for (int i = 0; i < 8; i++) {
          pion = 'V';  // vide
          if (othello[i + 1][j + 1] == 1)  pion = 'B';  // blanc
          else  if (othello[i + 1][j + 1] == 2)  pion = 'N';  // noir
          tampon.setCharAt(i, pion);
        }
        out.println(tampon);
      }

      out.close();
    }
    catch(IOException ioe) {
```

```
      return false;
    }
    return true;
  }

  public static void main(String[] args) {
    String fichierDeSauvetage = "othello.txt";
    WriteOthello mon_sauvetage = new WriteOthello("othello.txt");

    if (mon_sauvetage.sauve()) {
      System.out.println("Sauvegarde effectué dans " + fichierDeSauvetage);
    }
    else {
      System.out.println("Erreur de sauvegarde dans " + fichierDeSauvetage);
    }
  }
}
```

Nous rencontrons ici une nouvelle classe, `PrintWriter`. Cette classe nous permet d'écrire dans le fichier de la même manière que nous le faisons pour une sortie à l'écran avec notre `println()` habituel.

Lors de l'écriture binaire en Java, nous avions utilisé `FileOutputStream`. Ici, c'est presque pareil avec `PrintWriter`, mais `write()` est remplacé par `println()`. Cette dernière méthode nous écrira une chaîne de caractères terminée par une nouvelle ligne. Notre variable `tampon` possède une réserve pour 8 octets, qui seront initialisés, avant l'écriture, avec le contenu d'une ligne de notre tableau `othello`. Les lettres employées, `V`, `B` et `N`, remplaceront des valeurs binaires 0, 1 et 2, illisibles avec un éditeur.

Nous rappellerons que `StringBuffer` est mutable, au contraire de la classe `String`. Nous pourrions aussi construire notre chaîne avec cette dernière, mais cela serait moins efficace, puisqu'il faudrait composer le `String` avec une série de `+` (objet `String` régénéré à chaque fois).

Écriture d'un fichier texte en C++

Nous allons profiter de cette occasion pour présenter le XML, qui va nous permettre d'enregistrer des données dans un fichier texte traditionnel dans le cas de données structurées telles que des bases de données. Ce sera à la fois un exercice de style en C++, pouvant être adapté sans difficulté en Java, mais aussi une introduction essentielle pour cette technologie de plus en plus utilisée dans le commerce électronique.

Le XML pour l'information structurée

Le XML, ou *e**X**tensible **M**arkup **L**anguage*, a été conçu pour remédier aux insuffisances du HTML, ***H**yper **T**ext **M**arkup **L**anguage*, dans l'exploitation des informations structurées. Tout comme le HTML, le XML découle du SGML, ou ***S**tandard **G**eneral **M**arkup **L**anguage*,

qui a été développé pour maintenir une structure et un contenu standard pour des documents électroniques. Sur le site `http://www.w3.org` du World Wide Web Consortium, nous trouverons les recommandations pour les technologies du Web et les spécifications du XML.

Le XML a hérité du HTML, mais sans garder ses défauts. Il reste aussi beaucoup plus simple que son aïeul, le SGML. La structure hiérarchique d'un document est propre et l'information intégrée entre des balises symétriques, claires et concises. Le balisage XML permet une identification rapide du contenu des données.

Comme exemple, nous allons reprendre notre classe `Personne`, que nous avions développée au chapitre 4, mais sans l'année de naissance. Nous garderons les deux premiers attributs, le nom et le prénom. Le document XML devrait se présenter ainsi :

```
<?xml version="1.0"?>
<carnet>
  <personne>
     <nom>Haddock</nom>
     <prenom>Capitaine</prenom>
  </personne>
  <personne>
     <nom>Boichat</nom>
     <prenom>Jean-Bernard</prenom>
  </personne>
</carnet>
```

Les balises `personne`, `nom` et `prenom` sont similaires à celle du HTML. Dans notre `carnet`, nous avons deux `personne`, qui possèdent chacune un `nom` et un `prenom`. L'oubli du `é` est volontaire pour des raisons de programmation. Contrairement au HTML, la fin de balise, comme `</personne>`, est essentielle.

Sur la deuxième ligne, il n'y a pas de DTD, c'est-à-dire de déclaration de document type. Elle n'est pas obligatoire en XML, car notre document est bien formé, rigoureux et explicite. Un processeur XML n'aura aucune difficulté pour le traiter.

Écriture du fichier XML

L'exemple ci-dessous est simplifié à l'extrême et devait être en fait intégré à la classe `Personne` du chapitre 4, avec une méthode que nous pourrions appeler par exemple `write_XML()`. Nous avons ici créé une classe nommée `WriteXML`, qui va générer un fichier `personne.xml` sur le répertoire courant ::

```
// write_xml.cpp
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

class WriteXML {
```

```
  private:
    ofstream outfile;
    string avecBalise(const string balise, const string valeur) {
      return "     <" + balise + ">" + valeur +  "</" + balise + ">";
  }

  public:
    bool write_debut(const char *fichier) {
      outfile.open(fichier, ios::out);

      if (!outfile.is_open()) {
        return false;
      }

      outfile << "<?xml version=\"1.0\"?>" << endl;
      outfile << "<carnet>" << endl;
    }

    void write_record(const char *nom, const char *prenom) {
      outfile << "  <personne>" << endl;
      outfile << avecBalise("nom", nom) << endl;
      outfile << avecBalise("prenom", prenom) << endl;
      outfile << "  </personne>" << endl;
    }

    void write_fin() {
      outfile << "</carnet>" << endl;
      outfile.close();
    }
};
int main() {
  WriteXML mon_xml;

  if (!mon_xml.write_debut("personne.xml")) {
    cout << "Ne peut pas écrire dans le fichier personne.xml" << endl;
    return -1;
  }

  mon_xml.write_record("Haddock", "Capitaine");
  mon_xml.write_record("Boichat", "Jean-Bernard");
  mon_xml.write_fin();

  cout << "Fichier personne.xml généré" << endl;
}
```

Dans ce code, nous utilisons la méthode `open()` de la classe `ofstream`. Elle nous retourne un `void`, et il est donc essentiel d'appeler la méthode `bool is_open()` de cette même classe pour vérifier si le fichier a effectivement été créé et s'il est bien accessible pour y écrire nos données. Bien que le traitement des erreurs d'écriture soit réduit au minimum, nous fermons tout de même le fichier avec la méthode `close()`. L'emploi de l'opérateur `<<` de

la classe `ofstream` et du manipulateur `endl`, pour écrire le caractère de fin de ligne, relève vraiment d'une écriture élégante. Nous écrivons dans un fichier texte comme sur une console. La méthode privée `avecBalise()` nous permet d'exécuter le travail répétitif d'une manière simplifiée, sans une réflexion trop approfondie, car le contenu même d'un objet `personne` pourrait aussi y être intégré. Il nous faut aussi mentionner ici qu'il est tout à fait possible d'inclure, en XML, plusieurs objets d'un même type dans une structure. `personne` pourrait contenir plusieurs blocs de prénoms : ce serait d'ailleurs absolument nécessaire si nous voulions, par exemple, inclure la liste des enfants de cette même personne.

Après avoir exécuté le programme ci-dessus, qui crée le fichier `personne.xml`, nous pourrions le visualiser, par exemple avec Microsoft Internet Explorer, qui va filtrer ce document pour nous retourner ceci :

```
<?xml version="1.0" ?>
- <carnet>
- <personne>
  <nom>Haddock</nom>
  <prenom>Capitaine</prenom>
  </personne>
- <personne>
  <nom>Boichat</nom>
  <prenom>Jean-Bernard</prenom>
  </personne>
  </carnet>
```

Accès des répertoires sur le disque

De nombreux outils informatiques accèdent aux répertoires et aux fichiers d'un disque. Il est donc essentiel de couvrir ce sujet. Lorsque nous accédons au répertoire d'un disque et utilisons la commande `dir` sous DOS ou `ls -l` sous Linux (voir annexe C), le résultat nous est présenté sous forme de liste de fichiers ou de répertoires, et nous obtenons par exemple des données sur la protection, la dimension ou encore la date des fichiers. Les deux morceaux de code suivants peuvent être le départ d'un grand nombre de petits programmes qui peuvent être conçus plus particulièrement dans le cadre d'exercices de programmation. Nous pourrions nous imaginer des programmes de sauvegarde incrémentiel, de statistique, de compression ou encore de contrôle de versions (plusieurs versions du même code dans le cadre de projets informatiques, de développement ou de maintenance).

Lecture d'un répertoire en C++

Le code suivant utilise un certain nombre de fonctions de la bibliothèque C pour accéder au répertoire d'un disque :

```
// listdir.cpp
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
```

```
#include <ctime>

#include <iostream>

using namespace std;
int main() {
 DIR    *pdir;
 struct dirent *pdirent;
 struct stat statbuf;
 if ((pdir = opendir(".")) == NULL) {
   cout << "Ne peut ouvrir le répertoire ." << endl;
   return -1;
 }

 while ((pdirent = readdir(pdir)) != NULL) {
   cout << "Fichier: " << pdirent->d_name << endl;
   if (stat(pdirent->d_name, &statbuf) != 0)  continue;
   if (statbuf.st_mode & S_IWRITE) {
      cout << "Nous avons la permission d'écrire" << endl;
   }
   else {
      cout << "Nous n'avons pas la permission d'écrire" << endl;
   }
   if (statbuf.st_mode & S_IFDIR) {
      cout << "C'est un répertoire" << endl;
   }
   cout << "La lettre du disque: " << (char)('A'+statbuf.st_dev) << endl;
   cout << "La dimension en octets: " << statbuf.st_size << endl;
   cout << "La dernière fois qu'il fut ouvert: "
        << ctime(&statbuf.st_ctime) << endl;
 }
 closedir(pdir);
 return 0;
}
```

Il est vraiment parfait, ce petit programme ! Nous montrerons ici une partie du résultat :

```
Fichier: Lecture_texte.java
Nous avons la permission d'écrire
La lettre du disque: E
La dimension en octets: 684
La dernière fois qu'il fut ouvert: Thu Nov 11 18:27:32 1999
Fichier: strings.cpp
Nous avons la permission d'écrire
La lettre du disque: E
La dimension en octets: 1473
La dernière fois qu'il fut ouvert: Tue Nov 16 19:06:10 1999
```

Les trois premiers fichiers d'en-tête, `<sys/types.h>`, `<sys/stat.h>` et `<dirent.h>`, nous indiquent que nous utilisons des fonctions C. Cela signifie que ce programme pourrait rencontrer des difficultés en cours de compilation ou d'utilisation sur une autre plate-forme, ou bien

en utilisant, sous Windows, des outils tels que C++ Builder de Borland ou Visual C++ de Microsoft. Certaines adaptations pourront donc être nécessaires.

La fonction C `opendir()` nous permet d'accéder au répertoire courant qui est spécifié avec la notation `"."`. Nous aurions pu donner le chemin complet, en respectant la notation `/` ou `\` suivant le système d'exploitation. `opendir()` est défini dans `dirent.h` sur le répertoire `i386-mingw32\include` du compilateur et retourne un pointeur à une structure `DIR` définie dans ce même fichier d'en-tête. Nous conseillerons, comme ici, de vérifier les cas d'erreur.

La partie intéressante commence avec `readdir()`, qui nous permet d'obtenir un premier niveau d'information d'un fichier dans la structure `dirent`. Chaque fois que nous rappelons `readdir()`, jusqu'à l'obtention du `NULL`, nous recevons l'information du prochain fichier. Ici, nous n'utilisons que le nom du fichier obtenu avec `d_name`, car les autres données de la structure `pdirent` ne nous apporteraient rien. Avec celui-ci, nous pouvons utiliser une autre fonction C, beaucoup plus intéressante, `stat()` :

```
stat(pdirent->d_name, &statbuf)
```

Il faut être attentif à la manière de passer l'adresse de `statbuf` à la fonction `stat()`. Enfin, nous utilisons un certain nombre de données disponibles dans la structure `stat` pour nous retourner l'information telle qu'elle nous apparaît à la console. Il faudra se méfier de certains de ces attributs, comme `st_dev`, qui peuvent avoir une autre signification sur un autre système d'exploitation.

Le fichier d'en-tête `stat.h`, dans le répertoire `386-mingw32\include\sys`, doit absolument être consulté pour vérifier les définitions utilisées. `S_IFDIR`, par exemple, nous permettrait d'exécuter une nouvelle recherche récursive sur les sous-répertoires, ce que nous ferons d'ailleurs en exercice.

Lecture d'un répertoire en Java

Nous allons à présent faire le même travail en Java, c'est-à-dire obtenir la liste de tous les fichiers d'un répertoire. Nous verrons que la création d'une liste de fichiers est nettement plus élégante et flexible. Voici donc le code que nous allons commenter :

```
import java.io.*;
import java.util.Date;

class MonFileFilter implements FilenameFilter {
    public boolean accept(File dir, String name) {
        String tempname = name.toLowerCase();
        return (tempname.endsWith ("doc") || tempname.endsWith ("txt"));
    }
}

public class ListDir {
  private String repertoire;

  public static void main (String args[]) {
```

```
    if ( args.length != 0 ) {
      ListDir dr2 = new ListDir(args[0]);
    }
    else {
      ListDir dr2 = new ListDir("E:\\Mes Documents");
    }
  }

  public ListDir(String leRepertoire) {
    repertoire = leRepertoire;

    if (leRepertoire.charAt(leRepertoire.length() - 1) !=  File.separatorChar) {
      repertoire += File.separatorChar;
    }
    File monRep = new File(repertoire);

    if (monRep.exists()) {
      String lesFichiers[] = monRep.list(new MonFileFilter());
      int nombre = lesFichiers.length;
      System.out.println("Nombre de fichiers: " + lesFichiers.length);

      for (int i = 0; i < nombre; i++) {
        File leFichier = new File(repertoire + lesFichiers[i]);
        Date dernModif = new Date(leFichier.lastModified());
        System.out.println(lesFichiers[i] + " " + dernModif);
      }
    }
    else {
      System.out.println("Le répertoire " + repertoire + " n'existe pas");
    }
  }
}
```

Le résultat apparaîtra alors ainsi :

```
Nombre de fichiers: 3
document1.doc Sun Apr 25 11:23:08 GMT+02:00 1999
document2.doc Wed May 26 21:21:56 GMT+02:00 1999
readme.txt Sun Jun 06 18:28:04 GMT+02:00 1999
```

Dans cette version Java, au contraire de la version C++, aucune discussion de portabilité n'est à prévoir : le fonctionnement de ce code est garanti sur toutes les plates-formes. Comme ce code est de plus bien structuré, dans une classe, il nous donne tout de suite une meilleure impression.

Le `main()` est traditionnel ; il contient un répertoire par défaut, `E:\\Mes Documents`, si aucun n'est spécifié comme argument à l'exécution du programme. L'utilisation du `File.separatorChar` est essentielle si nous voulons que le programme soit portable, car il vérifiera et ajoutera le caractère `\` ou `/` si nécessaire.

Lire la liste des fichiers se fait à l'aide de la classe `File` et la méthode `list()`. Cette dernière reçoit un paramètre tout à fait particulier, et d'une puissance extraordinaire, un objet d'une classe héritée de `FilenameFilter`. Nous allons très vite comprendre l'utilisation de ce filtre. En effet, la méthode `accept()`, de la classe `MonFileFilter`, qui hérite de `FilenameFilter`, va être exécutée à l'appel de `monRep.list()`. Après une conversion locale du nom du fichier en minuscules pour accepter tous les `.txt`, `.Txt`, `.doc` ou autres `.DoC`, nous retournons un booléen au cas où le fichier posséderait une extension `.txt` ou `.doc`.

Au retour de `monRep.list()`, la variable `lesFichiers` contiendra un tableau avec tous les fichiers désirés. Nous passerons au travers du tableau `lesFichiers`, non sans avoir consulté au passage la documentation de la classe `File` ; celle-ci nous donnera l'usage de méthode `lastModified()`, qui nous retourne un `long`, valeur que la classe `Date` va accepter comme paramètre de constructeur.

Nous devrons nous demander comment notre programme fonctionne, puisque la méthode `exists()` de la classe `File` teste si le fichier existe alors que nous désirons contrôler l'existence du répertoire. C'est en effet correct, car nous avons toujours un `File.separatorChar` en fin de fichier.

Les flux en mémoire (C++)

Les streams en C++ ont été utilisés jusqu'à présent pour connecter des entrées et sorties sur des fichiers. Nous allons maintenant continuer notre tour d'horizon et parler de la fonction `sprintf()` de la bibliothèque C et des classes `istringstream` et `ostringstream`.

Afin de ne pas prendre les unes pour les autres, nous commencerons donc par définir deux catégories pour ces classes :

- la classe `istringstream` pour extraire des caractères ;
- la classe `ostringstream` pour composer des caractères.

Avant de passer à la description et à l'utilisation de ces deux classes, il est essentiel de revenir en arrière, dans le temps, et de se demander comment les programmeurs C se débrouillaient avant.

sprintf() de la bibliothèque C

Un programmeur C qui utilise régulièrement les fonctions C telles qu'`atoi()` (conversion chaîne de caractères à un `int`) ou `atof()` (conversion à un `float`) pour extraire des caractères ou `sprintf()` pour composer de nouvelles chaînes reconnaîtra rapidement ce type de code :

```
// sprintf.cpp
#include <iostream>
#include <cstdio>
#include <cstdlib>

using namespace std;
int const protect = 10;
```

```
int main()
{
  unsigned char protection1[protect];
  char tampon[25];
  unsigned char protection2[protect];
  int i;
  for (i = 0; i < protect; i++) {
    protection1[i] = 255;
    protection2[i] = 255;
  }
  sprintf(tampon, "Bonjour monsieur %s Salut %d\n",
                  "ABCDEFG", atoi("1234"));
  cout << tampon;
  cout << "12345678901234567890123456789012345" << endl;

  for (i = 0; i < protect; i++) {
    if (protection1[i] != 255) {
      cout << "Erreur dans protection1 à l'index " << i << "("
           << protection1[i] << ")" << endl;
    }
    if (protection2[i] != 255) {
      cout << "Erreur dans protection2 à l'index " << i << "("
           << protection1[i] << ")" << endl;
    }
  }
}
```

et son résultat :

```
Bonjour monsieur ABCDEFG Salut 1234
12345678901234567890123456789012345
Erreur dans protection1 à l'index 0(2)
Erreur dans protection1 à l'index 1(3)
Erreur dans protection1 à l'index 2(4)
Erreur dans protection1 à l'index 3(
)
Erreur dans protection1 à l'index 4(
```

Ce morceau de code pourrait sembler vraiment particulier à première vue, mais il n'est ici que pour montrer les problèmes potentiels de fonctions C comme `sprintf()`. Les fichiers d'en-tête `cstdio` et `cstdlib` sont nécessaires pour disposer des fonctions C que sont `sprintf()` et `atoi()`.

Nous avons encastré la zone `tampon` entre deux blocs de chaîne de caractères, `protection1` et `protection2`, qui sont initialisés avec des octets de bits à 1 (255) afin de vérifier si le programme est écrit dans ces zones. C'est effectivement le cas, car le 25 pour la dimension de notre `tampon` est beaucoup trop petit. Le résultat nous montre clairement les caractères `234` bien visibles et qui ont débordé. Ce n'est pas très important de comprendre plus en détail le positionnement exact car il est évidemment dépendant de la machine et du compilateur. La corruption pourrait être beaucoup plus vicieuse, comme créer des erreurs à d'autres endroits de la mémoire, du système ou du programme et ne pas se reproduire à

chaque exécution ! La solution parfaite à ce problème serait en fait de défendre les tampons `protection1` et `protection2` avec des moyens hardware, afin de protéger la mémoire contre l'écriture et de générer des exceptions par le logiciel.

L'`atoi("1234")` n'est ici que pour montrer son utilisation, la chaîne de caractères `1234` étant convertie en un entier qui sera repris par le `%d` du `sprintf()`. Pour les détails de `sprintf()`, il faudrait consulter la documentation. Nous retiendrons cependant la première partie, `tampon`, qui recevra le résultat du formatage. Dans la deuxième partie, nous avons des parties fixes et variables, `"Bonjour monsieur %s Salut %d\n"`. Comme nous avons deux parties variables, `%s` (pour une chaîne de caractères) et `%d` (pour un nombre décimal), nous obtiendrons aussi deux parties, qui doivent absolument apparaître, les `ABCDEFG` et `atoi("1234")`.

Les fonctions C `sprintf()` et son équivalent `printf()` pour la sortie à l'écran possèdent de nombreuses fonctions de formatage comme celle-ci :

```
sprintf(tampon,"AA%10.4fBB", 1.2);   // résultat : AA    1.2000BB
```

Le `%10.4f` nous indique un nombre avec dix positions et quatre chiffres après la virgule. Nous comprenons donc les espaces après les deux premiers `AA`. Cependant, elles peuvent être avantageusement remplacées par des fonctionnalités équivalentes, qui sont à disposition dans les classes `strstream` ou `stringstream`.

istringstream et ostringstream

Pour illustrer l'utilisation de ces deux classes, nous allons présenter un exemple simplifié qui comporte quelques variantes concrètes et utilisables sans grandes difficultés :

```
// FluxMem.cpp
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main(int argc, char **argv) {
    // Lit une ligne de la console :
    cout << "Entrer deux chiffres en HEXA (ex: 12 A0): ";
    string input;
    getline(cin, input);

    // Décode l'entrée
    istringstream is(input);
    int c1;
    int c2;

    is.setf(ios::hex, ios::basefield); // on travaille en hexadécimal
    is >> c1 >> c2;
    cout << "test1: " << c1 <<endl;
    cout << "test2: " << c2 <<endl;
```

```
    int resultat = c1 * c2;

    // Affiche le résultat
    ostringstream os;
    os << "Le resultat est " << resultat << endl;

    cout << os.str();

    // Empêche la fenêtre DOS de se fermer
    cout << "Entrer retour";
    getline(cin, input);
    return 0;
}
```

Après avoir compilé `FluxMem.cpp` avec g++ ou le `make` dans l'éditeur Crimson, il faudra l'exécuter sans capture (voir annexe C) ou alors avec `FluxMem.exe` depuis l'explorateur de Windows et dans le répertoire `C:\JavaCpp\EXEMPLES\Chap09`.

Figure 9-2
Exécution de FluxMem.exe

L'instruction :

```
is >> c1 >> c2;
```

fonctionne, car nous avons un espace entre les deux chiffres et l'`istringstream` possède par défaut l'espace comme séparateur pour l'opérateur >>.

Les instructions suivantes sont pratiques :

```
cout << "test1: " << c1 <<endl;
cout << "test2: " << c2 <<endl;
```

dans le cas où nous n'avons pas de débogueur pour faire du pas à pas (voir annexe E, section « Présentation de NetBeans »). Toute la partie du `ostringstream` fait davantage partie ici d'un gadget, car le résultat aurait pu être présenté plus simplement : nous envoyons tout dans le stream `os` avant de le récupérer avec `os.str()`.

L'exemple ci-dessus pourrait être étendu à l'infini (avec ou sans `sstream`), par exemple pour écrire un petit calculateur !

Un exemple complet avec divers formatages

Après notre exemple simplifié à l'extrême, nous allons présenter quelques fonctions de formatage qui se trouvent à disposition dans la bibliothèque des `iostream` et `sstream`. Au fil des exemples suivants, nous allons découvrir toute une série d'options de formatage possibles.

Dans ce long exemple, nous avons encore gardé notre mécanisme de protection, seulement pour nous convaincre qu'il faut oublier nos anciennes habitudes de programmeur C. Il faut noter ici que ce code est extrêmement délicat et qu'il faudrait utiliser des outils comme PurifyPlus (Rational Software) pour vérifier si tous les tampons sont correctement initialisés et qu'il n'y a pas de fuite de mémoire.

```
// strstr.cpp
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main(int argc, char* argv[])
{
  ostringstream sortie1;
  sortie1 << "Bonjour Monsieur ";
  sortie1 << "1234567890abcdefg" << ends;
  cout << "Test1: " << sortie1.str() << endl;

  ostringstream sortie3;
  sortie3 << "12345" << ends;
  sortie3 << "67890" << ends;
  cout << "Test2: " << sortie3.str() << endl;

  stringstream entree_sortie1;
  entree_sortie1 << "abcd\n1234" << ends;

  cout << "Test3: " << entree_sortie1.str() << endl;

  stringstream entree_sortie2;
  int nombre1 = 10;
  int nombre2 = 20;
  string un_string1 = "Bonjour";
  entree_sortie2 << nombre1 << "." << nombre2 << un_string1 << ends;
  cout << "Test4: " << entree_sortie2.str() << endl;

  double double1;
  entree_sortie2 >> double1;
  cout << "Test5: " << double1 << endl;

  string un_string2;
  entree_sortie2 >> un_string2;
  cout << "Test6: " << un_string2 << endl;
```

```
  ostringstream sortie4;
  sortie4.fill('0');
  sortie4.setf(ios::right, ios::adjustfield);
  sortie4.width(10);
  double double2 = 1.234;
  sortie4 << double2;
  sortie4 << "|";
  sortie4.setf(ios::left, ios::adjustfield);
  sortie4.width(8);
  sortie4.precision(3);
  sortie4 << double2;
  sortie4 << "|";
  int nombre5 = 31;
  sortie4 << hex << nombre5 << ":" << oct << nombre5 << ends;

  cout << "Test7: " << "[" << sortie4.str() << "]" << endl;

  return 0;
}
```

Ce code nous donnera le résultat suivant sans entrer de valeurs comme arguments :

```
Test1: Bonjour Monsieur 1234567890abcdefg
Test2: 12345
Test3: abcd
1234
Test4: 10.20Bonjour
Test5: 10.2
Test6: Bonjour
Test7: [000001.234|1.230000|1f:37
```

Le `Test1` utilise un `ostringstream` dynamique. La mémoire sera allouée automatiquement, et `ends` terminera l'entrée sur le flux. La méthode `str()` va nous retourner la chaîne de caractères.

Le `Test2` est très similaire au `Test1`, mais le premier `ends` va fermer le flux : le `67890` est perdu.

Dans le cas des `Test3`, `Test4`, `Test5` et `Test6`, il est intéressant de noter l'utilisation de `stringstream` au lieu de `ostringstream`.

Remarquons que notre chaîne de caractères `"abcd\n1234"` est sur deux lignes, ce qui explique le résultat du `Test3` sur deux lignes. Dans `entree_sortie1`, nous pourrions avoir un fichier texte complet.

Dans le `Test4`, nous montrons ce qui a été effectivement stocké dans `entree_sortie2`.

Dans les `Test5` et `Test6`, nous faisons l'extraction dans l'autre sens. Nous voyons que nous trichons en quelque sorte, puisque nous traitons un `double` et un `String` : `double1` et `un_string2`. Si nous avions un autre format ou une entrée totalement aléatoire et confuse, nous devrions traiter les erreurs correctement.

Le Test7, qui est en effet un concentré d'options de formatage, nous sortira le résultat suivant, composé de trois morceaux :

```
[000001.234|1.230000|1f:37]
```

Les deux premières parties sont composées du nombre 1.234, qui est une fois aligné à droite avec setf(ios::right, ios::adjustfield) et l'autre fois à gauche. Nous contrôlons le remplissage, à gauche ou à droite, grâce à fill('0'), mais le caractère 0 pourrait très bien être remplacé par autre chose. La largeur des champs, qui vaut 10 et 8, est modifiable avant l'opérateur << par la méthode width(). Le deuxième nombre a-t-il perdu des chiffres ? Non, car 1.23 représente une précision de trois chiffres, indépendamment du nombre de chiffres après la virgule. Enfin, le 1f:37 représente la nombre décimal 33 dans les formats hexadécimal et octal.

Le printf Java du JDK 1.5

La fonction printf() est une fonction bien connue des programmeurs C. Nous avons écrit un petit morceau de code en C :

```
// printfC.c          //.c et non .cpp
#include <stdio.h>   //du C pas du C++

main() {
  int  nombre = 17;
  char *text  = "hexa";

  printf("Le nombre %d est %x en %s\n", nombre, nombre, text);
}
```

Le résultat nous donne :

```
Le nombre 17 est 11 en hexa
```

Les %d, %x et %s (décimal, hexadécimal et string) vont prendre et convertir les données des trois variables qui suivent la partie "…" !

Dans le Makefile de ce chapitre, nous avons ajouté une entrée c et un appel au compilateur C de cette manière :

```
gcc -o printfC.exe printfC.c
```

En revanche, en éditant avec Crimson le fichier printF.c, nous pourrons tout aussi bien le compiler avec le compilateur C++, c'est-à-dire g++ dans le menu.

Depuis la version 1.5 de Java, nous pouvons écrire le même programme qui donnera le même résultat :

```
import java.io.*;

class Printf  {
  public static void main (String args[]) {
    int   nombre = 17;
```

```
    String text  = "hexa";

    System.out.printf("Le nombre %d est %x en %s\n", nombre, nombre, text);
  }
}
```

istrstream et ostrstream

Si le lecteur rencontre ces anciennes formes dépréciées (éditions précédentes de cet ouvrage), il devra les convertir avec les `stringstream`. Lors de la compilation de ces anciennes classes avec g++ (version récente), les messages d'erreur nous aiderons dans ce travail. Si le langage C++ avait été développé dès le départ avec une bibliothèque standard et une classe `String` comme en Java, nous n'aurions peut-être jamais inventé les `strstream` ni entendu parler d'eux. Nous allons maintenant présenter d'autres exemples variés que nous devrions rencontrer dans la programmation de tous les jours.

Formatage en Java

Passons à présent au formatage et à la conversion en Java, car nous nous trouvons dans le même contexte : c'est en effet l'une des premières difficultés rencontrées par le programmeur. Nous allons reprendre ici un certain nombre de formes déjà utilisées, que nous utilisons par exemple pour la conversion des arguments reçus par le `main()`. C'est une sorte de synthèse à titre comparatif puisqu'en C++ les flux seront utilisés en général.

La méthode `println()` de la classe `System.out` appliquée à l'opération suivante :

```
int valeur1 = 8;
double valeur2 = 1423.35 * 0.033;
System.out.println(valeur1 + ":" + valeur2);
```

et qui nous sortira le résultat suivant :

```
8:46.970549999999996
```

ne nous donnera certainement pas satisfaction. Ce résultat ne correspondra sans doute pas à celui désiré, avec uniquement deux décimales imprimées, des 0 ou des espaces à gauche et un alignement précis du style :

```
00008:00046.97
```

Ce format peut être obtenu, d'une manière extrêmement simple, grâce au code suivant :

```
DecimalFormat df1 = new DecimalFormat("00000");
DecimalFormat df2 = new DecimalFormat("00000.00");
System.out.println(df1.format(valeur1) + ":" + df2.format(valeur2));
```

La classe `DecimalFormat`, d'une simplicité déconcertante, nous permet d'obtenir le résultat désiré. Les 0 indiquent la position des chiffres, ces derniers pouvant prendre toutes les valeurs, y compris 0. Il faut être très attentif aux résultats dans les cas où nous dépasserions

les limites indiquées. Il faut vraiment vérifier si cela correspond à notre spécification, car nous pourrions être surpris :

```
System.out.println(df2.format(7654321.999));
```

Dans ce cas, nous sommes en mesure de nous poser deux questions :

1. Que se passe-t-il avec le .999 car nous avons choisi deux décimales ?
2. Notre maximum spécifié dans `df2` est en principe 99999.99. Que va-t-il se passer ?

Le résultat ne nous surprendra qu'à moitié :

```
7654322.00
```

Filtrer du texte en Java avec StringTokenizer et StreamTokenizer

Un *token*, en anglais, signifie une marque. En effet, lorsque nous voulons analyser, par exemple, une chaîne de caractères, nous pourrions utiliser une ou plusieurs marques pour identifier, filtrer ou chercher des mots ou des parties de mots. L'une des premières applications est l'extraction de données. Il est souvent plus facile d'utiliser `StringTokenizer` au lieu de se lancer dans du code complexe utilisant les méthodes de la classe `String`, ou dans une recherche caractère par caractère.

Outre la classe `java.util.StringTokenizer`, qui utilise comme entrée un `String`, la bibliothèque Java possède une autre classe, le `java.io.StreamTokenizer`, qui a lui besoin d'un `InputStream` comme entrée. Cet `InputStream` peut représenter un fichier ou encore correspondre à une entrée de données directement sur la console. C'est ce cas-là que nous avons choisi dans l'exemple qui suit. Nous commencerons par trois exemples de filtres avec la classe `StringTokenizer` :

```
import java.util.StringTokenizer;
import java.io.*;

class MonTokenizer {
  public static void main(String[] args) {
    StringTokenizer st = new StringTokenizer("Nous faisons un premier test");
    while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
    }
    st = new StringTokenizer("Nous\tfaisons  un\r \n \r deuxième \ntest");
    while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
    }
    st = new StringTokenizer("Ma;base;de;donnée", ";");
    while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
    }
    Reader lecteur = new BufferedReader(new InputStreamReader(System.in));
```

```
    StreamTokenizer streamt = new StreamTokenizer(lecteur);
    try {
      while (streamt.nextToken() != StreamTokenizer.TT_EOF) {
        switch (streamt.ttype) {
          case StreamTokenizer.TT_NUMBER:
             System.out.println("Un nombre: " + streamt.nval);
             break;
          case StreamTokenizer.TT_WORD:
             System.out.println("Un mot: " + streamt.sval);
             break;
          default:
             System.out.println("Un autre type: " + (char)streamt.ttype);
             break;
        }
      }
    }
    catch (IOException e) {
       System.out.println("IOException: " + e);
    }
  }
}
```

Les deux premiers constructeurs de `StringTokenizer` ne reçoivent pas de second paramètre : cela signifie que le défaut sera utilisé. Celui-ci correspond à l'espace, au tabulateur et aux nouvelles lignes (PC et Linux inclus). Le résultat de cette partie :

```
Nous
faisons
un
premier
test
Nous
faisons
un
deuxième
test
```

nous montre que les espaces et autres caractères de séparations multiples sont bien traités. Pour le deuxième objet, si nous avions utilisé la forme :

```
st = new StringTokenizer("Nous\tfaisons  un\r \n \r deuxième \ntest", "x\n\r\t ");
```

nous aurions obtenu le même résultat, sauf que le mot `deuxième` aurait été coupé en deux en perdant le `x` puisque nous l'aurions traité comme une marque (*token*).

Le troisième objet de `StringTokenizer` utilise cette fois-ci une séparation avec le caractère `;`. C'est le cas classique d'une base de données :

```
Ma
base
de
données
```

qui pourrait être par exemple exportée depuis Microsoft Access. Nous noterons que la classe `StringTokenizer` possède les méthodes `hasMoreTokens()` et `nextToken()`, qui nous permettent de nous promener de morceau à morceau.

Nous terminons notre exemple avec le `StreamTokenizer`, dont nous avons initialisé le `Reader` depuis la console (`System.in`). Au contraire de `StringTokenizer`, nous ne pouvons pas définir les caractères qui seront utilisés comme délimiteur. La classe `StreamTokenizer` va nous retourner le type du morceau qu'il a filtré. Après avoir réceptionné le prochain morceau avec `nextToken()`, nous vérifions son type avec `ttype`. Ici, nous avons vérifié `TT_NUMBER` et `TT_WORD` spécialement à titre d'exercice. Afin de ne pas toujours entrer les mêmes caractères, nous avons introduit le texte suivant dans le fichier `test.txt` :

```
abcd 1234 xyz
unTabSuit       aprèsLeTab
ligneVide_avant
```

et entré la ligne de commande :

```
type test.txt | java MonTokenizer
```

Nous avons obtenu pour la dernière partie du programme le résultat :

```
Un mot: abcd
Un nombre: 1234.0
Un mot: xyz
Un mot: unTabSuit
Un mot: aprèsLeTab
Un mot: ligneVide
Un autre type: _
Un mot: avant
```

avec le cas particulier du caractère _. C'est relativement puissant, mais encore très loin d'une « tokenisation » pour une chaîne de caractères telle que `"<a href="mailto:jean-bernard@boichat.ch">Email at home</a>"`.

Résumé

Nous savons à présent maîtriser la lecture et l'écriture de fichiers. Un grand nombre d'applications utilisent des données stockées sur le disque, les analysent, les transforment et peuvent aussi les sauvegarder à nouveau pour une utilisation future. Nous avons vu aussi comment former et filtrer des chaînes de caractères en mémoire, en utilisant des techniques et opérateurs similaires à la lecture et à l'écriture de fichiers sur le disque.

Exercices

Les quatre premiers exercices seront codés en Java et C++. Nous avons écrit un certain nombre d'exercices, mais avec la tentation d'en écrire beaucoup plus. Ce chapitre est non seulement l'un des plus importants, mais aussi l'un des plus intéressants en ce qui

concerne les possibilités d'écriture de petits programmes d'apprentissage. Un résultat sous forme de fichier est toujours plus concret et motivant, donnant au programmeur le sentiment d'avoir vraiment créé quelque chose.

1. Écrire un programme de copie de fichier binaire. Prendre l'exécutable compilé en C++ comme exemple et vérifier que sa copie est identique en l'exécutant ou en utilisant l'exercice 3.
2. Écrire un programme qui lit un fichier texte Linux (respectivement DOS) et le convertit en fichier texte DOS (respectivement Linux).
3. Écrire un programme de comparaison de deux fichiers binaires. Si les fichiers sont différents, donner l'index et la valeur du premier octet différent.
4. Même exercice que le 3, mais pour un fichier texte avec le numéro et la position de la première ligne différente.
5. Sauver en C++ une partie d'Othello en format texte simple, comme nous l'avons fait précédemment en Java dans la classe `WriteOthello`.
6. À partir du programme `listdir.cpp` en C++ et de la classe `ListDir` en Java, écrire dans les deux langages une classe `ListeRepertoires` qui va pouvoir présenter une liste de tous les fichiers dans le répertoire sélectionné et ses sous-répertoires. Il s'agit d'un exercice d'une certaine complexité qui pourrait même être étendu ou réutilisé pour l'écriture d'outils de maintenance ou de statistique.