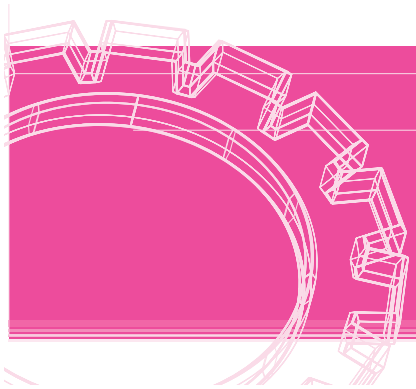
Chapitre 16

Les flots d'entrée et de sortie



Rappels

Un flot est un canal recevant (flot d'« entrée ») ou fournissant (flot de « sortie ») de l'information. Ce canal est associé à un périphérique ou à un fichier. Un flot d'entrée est un objet de type `istream` tandis qu'un flot de sortie est un objet de type `ostream`. Le flot `cout` est un flot de sortie prédéfini, connecté à la sortie standard `stdout` ; de même, le flot `cin` est un flot d'entrée prédéfini, connecté à l'entrée standard `stdin`.

La classe ostream

Elle surdéfinit l'opérateur << sous la forme d'une fonction membre :

```
ostream & operator << (expression)
```

L'expression correspondant à son deuxième opérande peut être d'un type de base quelconque, y compris `char`, `char *` (on obtient la chaîne pointée) ou un pointeur sur un type quelconque autre que `char` (on obtient la valeur du pointeur) ; pour obtenir la valeur de l'adresse d'une chaîne, on la convertit artificiellement en un pointeur de type `void *`.

Fonctions membres :

- **`ostream & put (char c)`** transmet au flot correspondant le caractère `c`.
- **`ostream & write (void * adr, int long)`** envoie `long` caractères, prélevés à partir de l'adresse `adr`.

La classe istream

Elle surdéfinit l'opérateur >> sous la forme d'une fonction membre :

```
istream & operator >> (type_de_base &)
```

Le `type_de_base` peut être quelconque, pour peu qu'il ne s'agisse pas d'un pointeur (`char *` est cependant accepté ; il correspond à l'entrée d'une chaîne de caractères, et non d'une adresse).

Les « espaces blancs » (espace, tabulation horizontale `\t` ou verticale `\v`, fin de ligne `\n` et changement de page `\f`) servent de « délimiteurs » (comme dans `scanf`), y compris pour les chaînes de caractères.

Principales fonctions membres :

- **`istream & get (char & c)`** extrait un caractère du flot d'entrée et le range dans `c`.
- **`int get ()`** extrait un caractère du flot d'entrée et en renvoie la valeur (sous forme d'un entier) ; fournit EOF en cas de fin de fichier.
- **`istream & read (void * adr, int taille`**) lit `taille caractères` sur le flot et les range à partir de l'adresse `adr`.

La classe iostream

Elle est dérivée de `istream` et `ostream`. Elle permet de réaliser des entrées sorties « conversationnelles ».

Le statut d'erreur d'un flot

À chaque flot est associé un ensemble de bits d'un entier formant le « statut d'erreur du flot ».

Les bits d'erreur

La classe `ios` (dont dérivent `istream` et `ostream`) définit les constantes suivantes :

- `eofbit` : fin de fichier (le flot n'a plus de caractères disponibles) ;
- `failbit` : la prochaine opération sur le flot ne pourra pas aboutir ;
- `badbit` : le flot est dans un état irrécupérable ;
- `goodbit` (valant, en fait `0`) : aucune des errreurs précédentes.

Une opération sur le flot a réussi lorsqu'un des bits `goodbit` ou `eofbit` est activé. La prochaine opération sur le flot ne pourra réussir que si `goodbit` est activé (mais il n'est pas certain qu'elle réussisse !).

Lorsqu'un flot est dans un état d'erreur, aucune opération ne peut aboutir tant que la condition d'erreur n'a pas été corrigée et que le bit d'erreur correspondant n'a pas été remis à zéro (à l'aide de la fonction `clear`).

Accès aux bits d'erreur

La classe `ios` contient cinq fonctions membre :

- **`eof ()`** : valeur de `eofbit` ;
- **`bad ()`** : valeur de `badbit` ;
- **`fail ()`** : valeur de `failbit` ;
- **`good ()`** : `1` si aucun bit du statut d'erreur n'est activé ;
- **`rdstate ()`** : valeur du statut d'erreur (entier).

Modification du statut d'erreur

`void clear (int i=0)` donne la valeur `i` au statut d'erreur. Pour activer un seul bit (par exemple `badbit`), on procédera ainsi (`fl` étant un flot) :

```
fl.clear (ios::badbit | fl rdstate() ) ;
```

Surdéfinition de () et de !

Si `fl` est un flot, `(fl)` est vrai si aucun des bits d'erreur n'est activé (c'est-à-dire si `good` est vrai) ; de même, `!fl` est vrai si un des bits d'erreur précédents est activé (c'est-à-dire si `good` est faux).

Surdéfinition de << et >> pour des types classe

On surdéfinira << et >> pour une classe quelconque, sous forme de fonctions amies, en utilisant ces « canevas » :

```
ostream & operator << (ostream sortie, type_classe objet[1])
{
     // Envoi sur le flot sortie des membres de objet en utilisant
     // les possibilités classiques de << pour les types de base
     // c'est-à-dire des instructions de la forme :
     //     sortie << ..... ;
   return sortie ;
}
```

1. Ici, la transmission peut se faire par valeur ou par référence.

```
istream & operator >> (istream & entree, type_de_base & objet)
{
     // Lecture des informations correspondant aux différents membres de objet
     // en utilisant les possibilités classiques de >> pour les types de base
     // c'est-à-dire des instructions de la forme :
     //     entree >> ..... ;
   return entree ;
}
```

Le mot d'état du statut de formatage

À chaque flot, est associé un « statut de formatage » constitué d'un mot d'état et de trois valeurs numériques (gabarit, précision et caractère de remplissage).

Voici les principaux bits du mot d'état :

Le mot d'état du statut de formatage (partiel)

Nom de champ	Nom du bit (s'il existe)	Signification (quand activé)
`ios::basefield`	`ios::dec`	conversion décimale
	`ios::oct`	conversion octale
	`ios::hex`	conversion hexadécimale
	`ios::showbase`	affichage indicateur de base (en sortie)
	`ios::showpoint`	affichage point décimal (en sortie)
`ios::floatfield`	`ios::scientific`	notation «þscientifiqueþ»
	`ios::fixed`	notation «þpoint fixeþ»

Action sur le statut de formatage

On peut utiliser, soit des « manipulateurs » qui peuvent être « simples » ou « paramétriques », soit des fonctions membre.

a. Les manipulateurs non paramétriques

Ils s'emploient sous la forme :

```
flot << manipulateur

flot >> manipulateur
```

Les principaux manipulateurs non paramétriques sont :

Les principaux manipulateurs non paramétriques

Manipulateur	Utilisation	ACTION
`dec`	`Entrée/Sortie`	Active le bit de conversion décimale
`hex`	`Entrée/Sortie`	Active le bit de conversion hexadécimale
`oct`	`Entrée/Sortie`	Active le bit de conversion octale
`endl`	`Sortie`	Insère un saut de ligne et vide le tampon
`ends`	`Sortie`	Insère un caractère de fin de chaîne (`\0`)

b. Les manipulateurs paramétriques

Ils s'utilisent sous la forme :

```
istream & manipulateur (argument)

ostream & manipulateur (argument)
```

Voici les principaux manipulateurs paramétriques :

Les principaux manipulateurs paramétriques

Manipulateur	Utilisation	Rôle
`setbase (int)`	`Entrée/Sortie`	Définit la base de conversion
`setprecision (int)`	`Entrée/Sortie`	Définit la précision des nombres flottants
`setw (int)`	`Entrée/Sortie`	Définit le gabarit. Il retombe à `0` après chaque opération

L'utilisation des manipulateurs paramétriques nécessite l'inclusion du fichier en-tête `iomanip`. Dans certaines implémentations, il peut encore s'agir de `iomanip.h` (associé alors à `iostream.h` et non à `iostream`, et sans l'utilisation de l'espace de noms `std`).

Association d'un flot à un fichier

La classe `ofstream`, dérivant de `ostream`, permet de créer un flot de sortie associé à un fichier :

```
ofstream flot (char * nomfich, mode_d_ouverture)
```

La fonction membre `seekp (déplacement, origine)` permet d'agir sur le pointeur de fichier.

De même, la classe `ifstream`, dérivant de `istream`, permet de créer un flot d'entrée associé à un fichier :

```
ifstream flot (char * nomfich, mode_d_ouverture)
```

La fonction membre `seekg (déplacement, origine)` permet d'agir sur le pointeur de fichier.

Dans tous les cas, la fonction `close` permet de fermer le fichier.

L'utilisation des classes `ofstream` et `ifstream` demande l'inclusion du fichier `fstream`. Dans certaines implémentations, il peut encore s'agir de `fstream.h` (associé alors à `iostream.h` et non à `iostream`, et sans l'utilisation de l'espace de noms `std`).

Modes d'ouverture d'un fichier

Les différents modes d'ouverture d'un fichier

Bit de mode d'ouverture	Action
`ios::in`	Ouverture en lecture (obligatoire pour la classe `ifstream`)
`ios::out`	Ouverture en écriture (obligatoire pour la classe `ofstream`)
`ios::app`	Ouverture en ajout de données (écriture en fin de fichier)
`ios::ate`	Se place en fin de fichier après ouverture
`ios::trunc`	Si le fichier existe, son contenu est perdu (obligatoire si `ios::out` est activé sans `ios::ate` ni `ios::app`)
`ios::binary`	(Utile dans certaines implémentations uniquement.) Le fichier est ouvert en mode dit « binaire » ou encore « non translaté »

Exercice 118

Énoncé

Écrire un programme qui lit un nombre réel et qui en affiche le carré sur un « gabarit » minimal de 12 caractères, de 22 façons différentes :

- en « point fixe », avec un nombre de décimales variant de 0 à 10,
- en notation scientifique, avec un nombre de décimales variant de 0 à 10.

Dans tous les cas, on affichera les résultats avec cette présentation :

```
précision de xx chiffres : ccccccccccc
```

Solution

Il faut donc activer d'abord le bit `fixed`, ensuite le bit `scientific` du champ `floatfield`. Nous utiliserons la fonction `setf`, membre de la classe `ios`. Notez bien qu'il faut éviter d'écrire, par exemple :

```
setf (ios::fixed) ;
```

En effet, cela activerait le bit `fixed`, sans modifier les autres donc, en particulier, sans modifier les autres bits du champ `floatfield`.

Le gabarit d'affichage est déterminé par le manipulateur `setw`. Notez qu'il faut transmettre ce manipulateur au flot concerné, juste avant d'afficher l'information voulue.

```
#include <iomanip>     // pour les "manipulateurs paramétriques"
#include <iostream>    // voir N.B. du paragraphe Nouvelles possibilités
                       // d'entrées-sorties du chapitre 2
using namespace std ;
main()
{  float val, carre ;
   cout << "donnez un nombre réel : " ;
   cin  >> val ;
   carre = val*val ;
   cout << "Voici son carré : \n" ;
   int i ;
   cout << "  en notation point fixe : \n" ;
   cout.setf (ios::fixed, ios::floatfield) ;// met à 1 le bit ios::fixed
                                            // du champ ios::floatfield
   for (i=0 ; i<10 ; i++)
      cout << "      précision de " << setw (2) << i << " chiffres : "
           << setprecision (i) << setw (12) << carre << "\n" ;
   cout << "  en notation scientifique : \n" ;
   cout.setf (ios::scientific, ios::floatfield) ;
   for (i=0 ; i<10 ; i++)
      cout << "      précision de " << setw (2) << i << " chiffres : "
           << setprecision (i) << setw (12) << carre << "\n" ;
}
```

À titre indicatif, voici un exemple d'exécution de ce programme :

```
donnez un nombre réel : 12.3456
Voici son carré :
  en notation point fixe :
      précision de  0 chiffres :          152
      précision de  1 chiffres :        152.4
      précision de  2 chiffres :       152.41
      précision de  3 chiffres :      152.414
      précision de  4 chiffres :     152.4138
      précision de  5 chiffres :    152.41385
      précision de  6 chiffres :   152.413849
      précision de  7 chiffres :  152.4138489
      précision de  8 chiffres : 152.41384888
      précision de  9 chiffres : 152.413848877
  en notation scientifique :
      précision de  0 chiffres : 1.524138e+002
      précision de  1 chiffres :     1.5e+002
```

```
précision de  2 chiffres :    1.52e+002
précision de  3 chiffres :   1.524e+002
précision de  4 chiffres :  1.5241e+002
précision de  5 chiffres : 1.52414e+002
précision de  6 chiffres : 1.524138e+002
précision de  7 chiffres : 1.5241385e+002
précision de  8 chiffres : 1.52413849e+002
précision de  9 chiffres : 1.524138489e+002
```

Exercice 119

Énoncé

Soit la classe `point` suivante :

```
class point
{   int x, y ;
   public :
    // fonctions membre
} ;
```

Surdéfinir les opérateurs << et >> de manière qu'il soit possible de lire un point sur un flot d'entrée ou d'écrire un point sur un flot de sortie. On prévoira qu'un tel point soit représenté sous la forme :

```
<entier, entier>
```

avec éventuellement des séparateurs « espaces_blancs » supplémentaires, de part et d'autre des nombres entiers.

Solution

Nous devons donc surdéfinir les opérateurs << et >> pour qu'ils puissent recevoir, en deuxième opérande, un argument de type `point`. Il ne pourra s'agir que de fonctions amies, dont les prototypes se présenteront ainsi :

```
ostream & operator << (ostream &, point) ;
istream & operator >> (istream &, point) ;
```

L'écriture de `operator <<` ne présente pas de difficultés particulières : on se contente d'écrire, sur le flot concerné, les coordonnées du point, accompagnées des symboles < et >.

En revanche, l'écriture de `operator >>` nécessite un peu plus d'attention. En effet, il faut s'assurer que l'information se présente bien sous la forme requise et, si ce n'est pas le cas, prévoir de donner au flot concerné l'état `bad`, afin que l'utilisateur puisse savoir que l'opération s'est mal déroulée (en testant « naturellement » l'état du flot).

Voici ce que pourrait être la déclaration de notre classe (nous l'avons simplement munie d'un constructeur) et la définition des deux fonctions amies voulues :

```
#include <iostream>
using namespace std ;

class point
{
   int x, y ;
  public :
   point (int abs=0, int ord=0)
     { x = abs ; y = ord ; }
   int abscisse () { return x ; }
   friend ostream & operator << (ostream &, point) ;
   friend istream & operator >> (istream &, point &) ;
} ;

ostream & operator << (ostream & sortie, point p)
{
  sortie << "<" << p.x << "," << p.y << ">" ;
  return sortie ;
}

istream & operator >> (istream & entree, point & p)
{
   char c = '\0' ;
   float x, y ;
   int ok = 1 ;
   entree >> c ;
   if (c != '<') ok = 0 ;
      else
       { entree >> x >> c ;
         if (c != ',') ok = 0 ;
            else
               { entree >> y >> c ;
                 if (c != '>') ok = 0 ;
               }
       }
   if (ok) { p.x = x ; p.y = y ; } // on n'affecte à p que si tout est OK
      else entree.clear (ios::badbit | entree.rdstate () ) ;
   return entree ;
}
```

À titre indicatif, voici un petit programme d'essai, accompagné d'un exemple d'exécution :

```
main()
{
  char ligne [121] ;
  point a(2,3), b ;
  cout << "point a : " << a << "  point b : " << b << "\n" ;
  do
   { cout << "donnez un point : " ;
     if (cin >> a)  cout << "merci pour le point : " << a << "\n" ;
             else { cout << "** information incorrecte \n" ;
                    cin.clear () ;
                    cin.getline (ligne, 120, '\n') ;
                  }
   }
  while ( a.abscisse () ) ;
}
```

```
point a : <2,3>  point b : <0,0>
donnez un point : 4,5
** information incorrecte
donnez un point : <4,5<
** information incorrecte
donnez un point : <4,5>
merci pour le point : <4,5>
donnez un point : < 8,   9     >
merci pour le point : <8,9>
donnez un point : bof
** information incorrecte
donnez un point : <0,0>
merci pour le point : <0,0>
```

Exercice 120

Énoncé

Écrire un programme qui enregistre (sous forme « binaire », et non pas formatée), dans un fichier de nom fourni par l'utilisateur, une suite de nombres entiers fournis sur l'entrée standard. On conviendra que l'utilisateur fournira la valeur `0` (qui ne sera pas enregistrée dans le fichier) pour préciser qu'il n'a plus d'entiers à entrer.

Solution

Si `nomfich` désigne une chaîne de caractères, la déclaration :

```
ofstream sortie (nomfich, ios::out) ;
```

permet de créer un flot de nom `sortie`, de l'associer au fichier dont le nom figure dans `nomfich` et d'ouvrir ce fichier en écriture.

L'écriture dans le fichier en question se fera par la fonction `write`, appliquée au flot `sortie`.

Voici le programme demandé :

```
const int LGMAX = 20 ;
#include <cstdlib>                          // pour exit
#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std ;
main()
{
   char nomfich [LGMAX+1] ;
   int n ;
   cout << "nom du fichier à créer : " ;
   cin >> setw (LGMAX) >> nomfich ;
   ofstream sortie (nomfich, ios::out) ;
   if (!sortie) { cout << "création impossible \n" ;
                  exit (1) ;
                }
   do
      { cout << "donnez un entier : " ;
        cin >> n ;
        if (n) sortie.write ((char *)&n, sizeof(int) ) ;
      }
   while (n && sortie) ;

   sortie.close () ;
}
```

Notez que `if (!sortie)` est équivalent à `if (!sortie.good())` et que `while (n && sortie)` est équivalent à `while (n && sortie.good())`.

Exercice 121

Énoncé

Écrire un programme permettant de lister (sur la sortie standard) les entiers contenus dans un fichier tel que celui créé par l'exercice précédent.

Solution

```
const int LGMAX = 20 ;
#include <cstdlib>                          // pour exit
#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std ;
main()
{
   char nomfich [LGMAX+1] ;
   int n ;
   cout << "nom du fichier à lister : " ;
   cin >> setw (LGMAX) >> nomfich ;
   ifstream entree (nomfich, ios::in) ;
   if (!entree) { cout << "ouverture impossible \n" ;
                  exit (1) ;
                }
   while ( entree.read ( (char*)&n, sizeof(int) ) )
        cout << n << "\n" ;

   entree.close () ;
}
```

Exercice 122

Énoncé

Écrire un programme permettant à un utilisateur de retrouver, dans un fichier tel que celui créé dans l'exercice 120, les entiers dont il fournit le « rang ». On conviendra qu'un rang égal à 0 signifie que l'utilisateur souhaite mettre fin au programme.

Solution

```
const int LGMAX_NOM_FICH = 20 ;
#include <cstdlib>              // pour exit
#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std ;

main()
{
  char nomfich [LGMAX_NOM_FICH + 1] ;
  int n, num ;
  cout << "nom du fichier à consulter : " ;
  cin >> setw (LGMAX_NOM_FICH) >> nomfich ;
  ifstream entree (nomfich, ios::in) ;
  if (!entree) { cout << "Ouverture impossible\n" ;
                 exit (1) ;
               }
  do
    { cout << "Numéro de l'entier recherché : " ;
      cin >> num ;
      if (num)
         { entree.seekg (sizeof(int) * (num-1) , ios::beg ) ;
           entree.read ( (char *) &n, sizeof(int) ) ;
           if (entree) cout << "-- Valeur : " << n << "\n" ;
              else {  cout << "-- Erreur\n" ;
                      entree.clear () ;
                   }
         }
    }
  while (num) ;
  entree.close () ;
}
```