Flots

Les mécanisme d'entrées/sorties

1. Hiérarchie de classes
2. Manipulateurs
3. Fichiers
4. Formatage des sorties
5. Entrées
6. Manipulateurs, le retour

Hiérarchie de classes

Les entrées et sorties sont gérées dans C++ à travers des objets particuliers appelés `streams` ou `flots`.

Ils **ne font pas** partie de la spécification de base de C++, et leur implémentation peut varier d'un compilateur à un autre.

La classe de base de la hiérarchie de classes est:

```
class ios
```

Les classes

```
class istream : public virtual ios
class ostream : public virtual ios
```

gèrent les entrées et les sorties. La classe

```
class iostream : public istream, public ostream
```

permet l'insertion et l'extraction.

Des classes particulières existent pour la gestion des fichiers:

```
class ifstream : public istream
class ofstream : public ostream
class fstream : public iostream
```

et d'autres classes pour la gestion de tableaux de caractères en mémoire (chaînes du C).

```
class istrstream : public istream
class ostrstream : public ostream
class strstream : public iostream
```

Éléments prédéfinis

Deux opérateurs sont surchargées de manière appropriée pour les flots:
* l'opérateur d'insertion << (écriture)
* l'opérateur d'extraction >> (lecture)

Les flots prédéfinis sont
* `cout` (de la classe `ostream_withassign`) associé à la sortie standard;
* `cerr` associé à la sortie erreur standard;
* `clog` associé à la sortie erreur standard, mais les données sont bufférisées;
* `cin` (de la classe `istream_withassign`) associé à l'entrée standard.

Les opérateurs d'insertion et d'extraction sont prédéfinis dans les classes `istream` et `ostream` pour les types de base, par exemple:

L'opérateur retourne une référence d'un `stream`, et on peut donc enchaîner les ordres de lecture resp. d'écriture.

Toute nouvelle classe X peut surdéfinir les opérateurs d'insertion et d'extraction en fonctions amies selon le canevas:

```
istream& operator>>(istream& in, X& objet) {
  /* Lecture de l'objet */
  return in;
}
ostream& operator<<(ostream& out, const X& objet) {
  /* Ecriture de l'objet */
  return out;
}
```

Etat d'un flot

Chaque flot possède un état dont la valeur reflète la situation du stream. L'état du flot peut être examiné par:

```
class ios {
...
public :
  int eof() const;  // fin de fichier : 1=EOF 0 sinon
  int fail() const; // opération a échoué : 2=fail, 0=ok
  int bad() const;  // flot est inutilisable : 4=bad, 0=ok
  int good() const; // flot est en bon etat :1=ok, 0 sinon
```

Dans l'état `fail()` on sait que le flot n'est pas altéré, alors que dans `bad()` on ne peut faire cette hypothèse.

Dans une écriture comme

```
while (cin >> i) {..};
```

l'état du flot est testé, et le test est réussi si et seulement si l'état est good(). Dans les autres cas, on retourne 0.

Une conversion est définie pour la classe ios:

```
ios::operator void * ();
```

qui retourne une adresse non nulle si l'état du fichier est `good()`.

Manipulateurs

Les manipulateurs sont une façon agréable de formater. Inclure

#include <iomanip.h>

* Un manipulateur s'insère dans le flot comme une donnée ou une référence
* Il s'applique à partir de son insertion.

Exemple

```
cout << x << flush << y << flush;
```

Les manipulateurs prédéfinis sont:

`oct`	(io) notation octale
`hex`	(io) notation hexadécimale
`endl`	(o) ajoute \n ou _cr\lf
`ends`	(o) ajouter \0 (fin de cahine)
`ws`	(i) ignore les blancs en tête pour la sasie
`setprecision(int)`	(o) chiffres décimaux
`setw(int)`	(o) gabarit d'écriture
`setfill(char)`	(o) craactères de remplissage

On retrouve des opérateurs de scanf et printf.

`setprecision(int n)` fixe à n le nombre de chiffres d'affichage.

`setw(int n)` fixe à n le nombre minimum de caractères utilisés pour le prochain affichage.

Fichiers

Les fichiers sont associés à des flots des classes ifstream, ofstream et fstream.

```
int main(int argc, char* argv[]) {
  if (argc != 3) error("arguments");
  ifstream depuis(argv[1]);
  ofstream vers(argv[2]);
  char c;
  while (depuis.get(c)) vers.put(c);
  if (!depuis.eof() || vers.bad())
    error("entree sortie");
}
```

Les constructeurs peuvent prendre un deuxième argument optionnel pour des modes de lecture ou écriture que sont

```
class ios { ...
public :
  enum open_mode {
    in = 1;
    out = 2;
    ate = 4; //ouvre a la fin
    app = 010; //append
    trunc = 020; // tronque a longueur 0
    nocreate = 040; // echoue si fichier n'existe pas
    noreplace = 0100; // echoue si fichier existe
  } ...
};
```

On peut alors écrire:

```
ofstream fichier(nom, ios::out | ios::nocreate);
```

Formatage des sorties

Formatage d'un flot par modification de l'état de formatage (un entier long de la classe ios).

* Chaque bit de l'état code un paramètre.

* Chaque paramètre a un nom symbolique
* setf() active le paramètre, unsetf() le désactive. Deux formes de setf() existent.
* Un paramètre activé le reste jusqu'à désactivation explicite.
* On peut obtenir la valeur de l'état par la méthode flags()

```
cout << hex << cout.flags() << endl;
```

Quelques paramètres

`ios::skipws`	0x0001	saute blancs en tête (entrée)
`ios::left`	0x0002	justification à gauche
`ios::dec`	0x0010	sortie en format décimal
`ios::oct`	0x0020	sortie en format octal
`ios::hex`	0x0040	sortie en format hexadécimal
`ios::showbase`	0x0080	montre 0 resp 0x
`ios::showpoint`	0x0100	complète les décimales
`ios::fixed`	0x1000	nombre fixe de décimales

Emploi des formats

Activation par `setf`, désactivation par `unsetf`. L'activation d'un attribut l'ajoute à l'état de formatage.
Première forme. de setf

```
cout.setf(ios::showpos); // effet: etat |= ios::showpos;
cout.unsetf(ios::showpos); // effet: etat &= !ios::showpos;
```

Plusieurs paramètres peuvent être activés ou désactivés simultanément. Par exemple,

```
cout.setf(ios::showpos | ios::showpoint);
cout << 65.32;
-> +65.3200
```

et

```
cout.setf(ios::hex | ios::uppercase |ios::showbase);
cout << 65;
-> 0X41
```

Rien (sauf le bon sens) n'empêche d'écrire

```
  cout.setf(ios::dec|ios::oct|ios::hex);
```

avec des résultats imprévisibles.

La deuxième forme de la méthode `setf()` prend deux arguments

```
setf(parametres, masque);
```

avec pour effet

```
unsetf(masque);
setf(parametres & masque); // attention
```

La deuxième forme de setf() est utilisée avec les raccourcis suivants:

`ios::basefield = ios::dec|ios::oct|ios::hex` pour la base de numération;

```
   cout.setf(ios::showbase);
   cout.setf(ios::hex, ios::basefield);
   cout << 65;
-> 0x41
```

`ios::floatfield = ios::fixed|ios::scientific` pour les nombres flottants;

```
   cout.setf(ios::fixed, ios::floatfield);
   cout << 65.2 << endl;
   cout.setf(ios::scientific, ios::floatfield);
   cout << 65.2 << endl;
-> 65.200000
   6.5200000e+01
```

`ios::adjustfield = ios::left|ios::right|ios::internal` pour la justification.

```
   cout.width(5); cout.fill(`*');
   cout.setf(ios::showpos);
   cout.setf(ios::internal,ios::adjustfield);
   cout << 65;
-> +**65
```

d'Autres fonctions d' IO

`put(char)` affiche son argument comme caractère:

```
   cout.put(65).put(67) << endl;
-> AC
```

`width(int)` fixe la longueur minimale de l'affichage (par défaut 0). Ne s'applique qu'à la donnée suivante

```
   cout.width(8); cout.fill(`*');
   cout << 65 << 66 << endl;
-> ******6566
```

`fill(char)` fixe le caractère de remplissage. S'applique à toutes les sorties subséquentes.
`precision(int)` fixe le nombre de chiffres .

Considérons les instructions suivantes:
cout << 1.23456789 << endl << 4.00 << endl << 5.6789E2 << endl << 0.0 << endl;
L'affichage par défaut est
1.23457
4
567.89
0
Précédé de `cout.precision(3);`, devient
1.23
4
568
0
Précédé de `cout.precision(3);cout.setf(ios::showpoint);`
1.23
4.00
568.
0.00
Par `cout.setf(ios::scientific, ios::floatfield);`
1.2345678900e+00
4.0000000000e+00
5.6789000000e+02
0.0000000000e+00
et avec `cout.setf(ios::fixed, ios::floatfield);`
1.2345678900
4.0000000000
567.8900000000
0.0000000000

Les fonctions `tellp` et `seekp` de la classe `ostream` positionnent l'écriture.

Servent pour l'écriture dans les fichiers (classe ofstream qui dérive de ostream).

* `tellp`() fixe la position courante dans le flot.
* `seekp` a deux formes
  - o `ostream& seekp(streampos pos)`. Fixe la position absolue à pos.
  - o `ostream& seekp(streampos pos, ios::seekdir base)`. Fixe la position, en comptant pos relativement à base. L'argument base peut prendre les valeurs `ios::beg, ios::cur ou ios::end.` (on retrouve l'interface de l'appel système).

Exemple:

```
ofstream in("toto"); // ouvre "toto" en ecriture
in << "12345";       // ecrit dans toto
cout << in.tellp();  // affiche 5
in.seekp(3);         // positionne apres le 3ième caractère
in << " soleil";
```

Le fichier contient 123 soleil.

Mêmes comportements pour les fonctions `seekg`() et `tellg`() en lecture. Sur un fichier ouvert en lecture écriture le curseur de fichier est le même pour `seekg` et `seekp`.

Entrées Lecture simple

Lecture simple
cin >> entier >> car >> flottant;
donne

23A1.2 23 A 1.2
23 A 1.2 23 A 1.2 (blancs en tête sont sautés)
23 A 1 .2 23 A 1

La méthode cin.get() vient en plusieurs variantes.
(1) cin.get(char&) honore les blancs: cin.get(a).get(b); pour l'entrée A B retourne a= et b=A.
(2) int iostream::get() sans argument retourne
* le dernier caractère lu, s'il y en a un;
* ou EOF si fin de fichier.

D'où une boucle comme

```
while((c = cin.get()) !=EOF) {...}
```

Lecture de textes

```
char txt[80]; cin >> txt;
```

donne, pour

```
Bonjour Monde
```

le résultat

```
Bonjour
```

car par défaut, les blancs en tête sont sautés et la lecture s'arrête au premier blanc.
De même

```
cin.get(txt,11);
```

donne

```
Bonjour
```

car le nombre demandé de caractères est lu.

Le manipulateur ws saute les blancs en tête. Ainsi
`cin >> ws; cin.get(txt,11);`
donne
Bonjour Mon

(5) La forme générale est
`get(char* txt, int len, delim = `\n');`
et lit au maximum `len` caractères, et arrête la lecture au premier `delim` rencontré. Dans Gnu, `getline`() a le même comportement, à ceci près que le délimiteur est enlevé du flot.

Autres fonctions de lecture
`gcount` retourne le nombre de caractères lus par la dernière opération de lecture.
`putback(char)` et `unget` replacent le dernier caractère lu dans le flot d'entrée.
`int peek()` a le même comportement que `get()` sauf que le caractère lu n'est pas retiré du flot d'entrée.

Etat de lecture
L'état de lecture décrit la situation du flot d'entrée. Quatre fonctions en retournent les détails. Par exemple

	eof()	***good()***	***fail()***	***bad()***	***(cin)***
65	F	T	F	F	T
Rien	F	F	T	F	F
Ctrl-D	T	F	T	F	F

En cas d'erreur, on passe en fail, et la lecture est bloquée. La méthode clear() remet l'état à good.
L'état d'un flot peut être testé par la méthode rdstate().

Lecture et écriture en mémoire

Il s'agit d'analogues des `sscanf` et `sprintf` de C.
On inclut `strstream.h`.
La classe `istrstream` a deux constructeurs

* `istrstream(const char*)` qui associe un flot à la chaîne passée en paramètre.
* `istrstream(const char*, int)` qui associe le flot à la chaîne avec taille.

Exemple:

```
char txt = "ABC 1.23 45";
char t[100]; float f; int i;

istrstream in(txt); // construction

in >> t >> f >> i;
cout << t <<","<< f  <<"," << i;
```

Le résultat est ABC,1.23,45.

La syntaxe est la même pour la sortie avec `ostrstream`

Lecture et écriture simultanée

Unstrstream peut servir comme file (FIFO) pour ranger des données.

Exemple:

```
strstream es;      // cree un strstream
es << "ABC" << " " << 1.234 << " " << 5 << endl;

cout << es.rdbuf() << endl; // tout

char t[100]; float f; int i;
es.seekg(0);      // rewind
es >> t >> f >> i;
cout << t << "," << f << "," << i << endl;
```

donne

ABC 1.234 5
ABC,1.234,5

Biensur n'a pas le comportement bloquant et circulaire d'un tube.

Manipulateurs, le retour

Pourquoi peut-on écrire

```
  cout << x << flush << y << flush; ?
```

La fonction (globale) flush() est déclarée

```
ostream& flush(ostream&);
```

Posons (pour simplifier ?)

```
typedef ostream& (*Manip)(ostream&);
```

Alors flush est du type `Manip`.

L'opérateur d'insertion de la classe ostream est déjà surchargé:

```
ostream& operator<<(Manip f) {
  return f(*this);
}
```

Noter l'apparition du schéma conceptuel visiteur.

Ainsi, `cout << endl` est l'équivalent de `endl(cout)` ! Par exemple, avec

```
ostream& h(ostream& out) {
  return out << " --- ";
}
```

La ligne

```
cout << "Bonjour" << h << "Monde";
```

affiche

```
Bonjour --- Monde
```

On transforme ainsi une fonction en manipulateur.

Une fonction avec argument dans un flot de sortie, comme

```
cout << setprecision(4) << x;
```

s'écrit en deux temps:

* la fonction `setprecision()` retourne un objet d'une nouvelle classe, contenant la fonction et l'argument;
* l'opérateur d'insertion de la nouvelle classe est surchargé pour appliquer la fonction à son argument.

Les fonctions considérées sont de type

```
typedef ostream& (*FManip)(ostream&, int);
```

La nouvelle classe est

```
class Omanip {
private:
  FManip f;
  int i;
public:
  Omanip(FManip f, int _i) : f(f), i(i) {}
  friend ostream& operator<<(ostream& os, Omanip& m)
    { return m.f(os, m.i);}
}
```

Exemple : la fonction `setprecision()` retourne un objet de la classe `Omanip` à partir d'une fonction de type `FManip`.

```
Omanip setprecision(int i) {
  return Omanip(&precision, i);
}
```

La fonction `ostream& precision(ostream&, int i)` est définie par

```
ostream& precision(ostream& os, int i) {
  os.precision(i);
  return os;
}
```

Ainsi, l'exécution de

```
cout << setprecision(4)
```

débute par l'évaluation de `setprecision(4)` qui retourne un objet temporaire de la classe `Omanip`, soit

```
temp =  Omanip(&precision, 4)
```

L'évaluation de

```
cout << temp
```

se traduit par l'appel de

```
precision(cout, 4)
```

Une définition générique est contenue dans le fichier `iomanip.h`