

Flux C++ (entrées-sorties)

La STL gère de nombreux aspects des entrées-sorties. Elle inaugure une façon de programmer pour rendre persistants les nouveaux types définis à l'aide du langage C++.

L'équipe qui l'a conçue à la fin des années 80 a eu le souci d'être conforme aux techniques en vigueur et de produire un travail qui résisterait au fil des ans.

Il faut reconnaître que la gestion des fichiers a énormément évolué depuis l'introduction de la bibliothèque standard : les bases de données relationnelles ont remplacé les fichiers structurés, et les interfaces graphiques se sont imposées face aux consoles orientées caractères.

Toutefois, les axes pris pour le développement de la STL étaient les bons. Si l'utilisation des flux est un peu tombée en désuétude, leur étude permet d'y voir plus clair pour produire une nouvelle génération d'entrées-sorties. Aussi, le terminal en mode caractères continue son existence, la vitalité des systèmes Linux en est la preuve.

1. Généralités

Pour bien commencer l'apprentissage des entrées-sorties, il faut faire la différence entre fichier et flux. Un fichier est caractérisé par un nom, un emplacement, des droits d'accès et parfois aussi un périphérique. Un flux (*stream* en anglais) est un contenu, une information qui est lue ou écrite par le programme. Cette information peut être de plus ou moins haut niveau. À la base, on trouve naturellement l'octet, puis celui-ci se spécialise en donnée de type entier, décimal, booléen, chaîne... Enfin, on peut créer des enregistrements composés d'informations très diverses. Il est tout à fait logique de considérer que la forme de ces enregistrements correspond à la formation d'une classe, c'est-à-dire d'un type au sens C++.

Les flux C++ (que l'on appelle parfois flots en langue française) sont organisés en trois niveaux ; le premier, le plus abstrait, regroupe les `ios_base`, format d'entrée-sortie indépendant de l'état et du formatage. Puis on trouve le niveau `basic_ios`, version intégrant la notion de paramètre régional (*locale* en anglais).

Enfin, nous trouvons le niveau `basic_istream`, groupe de modèles de classes

destinées à supporter le formatage de tous les types de base connus par le langage. C'est à ce niveau que nous travaillerons.

La circulation des informations se fait dans le cadre d'un système de mémoire tampon (*buffer*), supporté par la classe `basic_streambuf`.

Le programmeur-utilisateur de la STL connaît généralement les flux standards, `cout`, `cin` et `cerr`, ainsi que les classes `istream` et `ostream`.

2. Flux intégrés

Les flux intégrés, `cout`, `cin` et `cerr`, offrent de nombreuses façons d'échanger des informations. Les objets `cout` et `cerr` sont des instances de la classe `ostream` tandis que `cin` est une instance de la classe `istream`.

Ces classes proposent les opérateurs `<<` et `>>` pour lire ou écrire les types primitifs :

```
cout << "Bonjour"; // char*
int age;
cin >> age;
```

Nous avons vu qu'il était tout à fait possible d'enrichir le registre des types supportés en surchargeant cet opérateur à l'aide d'une fonction amie :

```
friend ostream& operator << (ostream& out, Type & t);
friend istream& operator >> (istream& in, Type & t);
```

3. État d'un flux

Les flux `istream` et `ostream` disposent de méthodes pour caractériser leur état.

<code>setstate(iostate)</code>	Ajoute un indicateur d'état.
<code>clear(iostate)</code>	Définit les indicateurs d'état.

4. Mise en forme

La classe `ios_base` propose un certain nombre de contrôles de formatage applicables par la suite. Ces contrôles s'utilisent comme des interrupteurs. On applique un formatage qui reste actif jusqu'à nouvel ordre.

<code>skipws</code>	Saute l'espace dans l'entrée.
<code>left</code>	Ajuste le champ en le remplissant après la valeur.
<code>right</code>	Remplit avant la valeur.
<code>internal</code>	Remplit entre le signe et la valeur.
<code>boolalpha</code>	Utilise une représentation symbolique pour true et false.
<code>dec</code>	Base décimale.
<code>hex</code>	Base hexadécimale.
<code>oct</code>	Base octale.
<code>scientific</code>	Notation avec virgule flottante.

<code>fixed</code>	Format à virgule fixe dddd.dd.
<code>showbase</code>	Ajoute un préfixe indiquant la base.
<code>showpoint</code>	Imprime les zéros à droite.
<code>showpos</code>	Indique + pour les nombres positifs.
<code>uppercase</code>	Affiche E plutôt que e.
<code>adjustfield</code>	Lié à l'ajustement du champ : internal, left ou right.
<code>basefield</code>	Lié à la base : dec, hex ou oct.
<code>floatfield</code>	Lié à la sortie en flottant : fixed ou scientific.
<code>flags()</code>	Lit les indicateurs.
<code>flags(fmtflags)</code>	Définit les indicateurs.
<code>setf(fmtflags)</code>	Ajoute un indicateur.
<code>unsetf(fmtflags)</code>	Annule un indicateur.

À l'aide des formateurs, nous pouvons afficher temporairement en hexadécimal une valeur entière :

```
cout << hex << 4096 << "\n";
```

```
cout << 8192 << "\n"; // toujours en hexa
cout << dec << 4096 << "\n"; // repasse en décimal
```

La fonction `width()` peut spécifier l'espace destiné à la prochaine opération d'entrée-sortie, ce qui est utile pour l'affichage de nombres. La méthode `fill()` fournit le caractère de remplissage :

```
cout.width(8);
cout.fill('#');
cout << 3.1415;
```

La STL propose également un certain nombre de manipulateurs de flux, comme `flush`, qui assure la purge du flux, c'est-à-dire le vidage du tampon à destination du périphérique de sortie :

```
cout << 3.1415 << flush;
```

5. Flux de fichiers

Pour travailler avec les fichiers d'une autre manière que le ferait le langage C, la STL propose les classes `ofstream` et `ifstream`. Il s'agit, pour les fichiers, d'adaptation par héritage des classes `ostream` et `istream`.

Voici pour commencer un programme de copie de fichier utilisant ces classes :

```
#include <fstream>

using namespace std;

int main(int argc, char* argv[])
{
    if(argc<3)
```

```

    return 1;

// ouverture des flux
ifstream src(argv[1]);
if(!src)
{
    cerr << "Impossible d'ouvrir " << argv[1] << endl;
    return 2;
}

ofstream dst(argv[2]);
if(!dst)
{
    cerr << "Impossible d'ouvrir " << argv[2] << endl;
    return 2;
}

// copie
char c;
while(src.get(c))
    dst.put(c);

// fermeture
src.close();
dst.close();

return 0;
}

```

Les classes possèdent la même logique que leurs ancêtres `ostream` et `istream`. La boucle de copie aurait donc pu être écrite de la manière suivante :

```

// copie
char c;
while(! src.eof())
{
    src >> c;
    dst << c;
}

```

```
}
```

Il existe une différence importante avec l'API proposée par le langage C : les quantités numériques sont traitées comme des chaînes de caractères. L'ouverture en mode binaire ne changerait rien à ce comportement, `ofstream` étant spécialisé dans le traitement des caractères (*char*).

```
sortie.open("fichier.bin",ios_base::binary);
sortie << x; // toujours la chaîne "43"
sortie.close();
```

6. Flux de chaînes

Un flux peut être attaché à une chaîne de caractères (*string*) plutôt qu'à un périphérique. Cette opération rend le programme plus générique et permet d'utiliser les séquences de formatage pour des messages avant une impression dans un journal (*log*), un fichier, un écran...

Pour utiliser les flux de chaînes, il faut inclure le fichier `<sstream>`, ainsi que l'expose le programme suivant :

```
#include <math.h>
#include <sstream>
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
    ostream out;
    double pi = 3.14159265358;
    out.width(8);
    out.setf(ios_base::scientific);
    out << "cos(pi/4)= " << cos(pi/4) << endl;
```

```

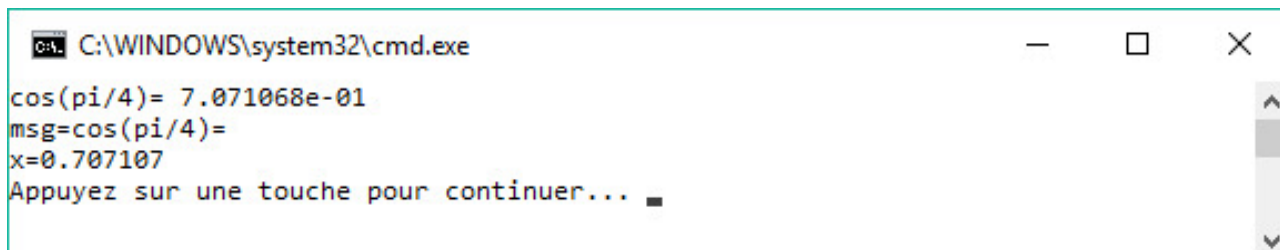
// sortie du flux sur une string
string res = out.str();

// finalement affichage
cout << res;

// relecture
istringstream in(res);
double x;
string msg;
in >> msg >> x;
cout << "msg=" << msg << endl << "x=" << x;
return 0;
}

```

L'exécution de ce programme est assez intéressante. À la relecture, le scanner stoppe la chaîne `msg` sur le caractère espace inséré après l'espace. D'autre part, nous retrouvons bien la valeur de la variable `x`, soit racine de 2 sur 2 (0.707) :



```

C:\WINDOWS\system32\cmd.exe
cos(pi/4)= 7.071068e-01
msg=cos(pi/4)=
x=0.707107
Appuyez sur une touche pour continuer...

```

7. Paramètres locaux

La classe `locale` définit un certain nombre de facettes destinées à être prises en compte lors du formatage par les flux de la STL. Chaque facette se charge d'un type de formatage pour une culture particulière. Imaginons pour l'exemple le formatage d'un nombre en monnaie pour un pays de la zone Euro, ou bien le formatage de la date en langue allemande.

Cette classe sert à choisir un certain nombre de facettes en vue de leur application sur un

flux par l'intermédiaire de la méthode `imbue()`. Elle contient aussi un certain nombre de méthodes statiques pour comparer deux classes locales entre elles.

Pour illustrer le fonctionnement subtil des classes locales définies par la STL, nous allons concevoir une facette pour afficher des nombres en euros. L'exemple pourra ensuite être aménagé pour supporter différents symboles monétaires.

Pour commencer, nous allons inclure les fichiers correspondant à l'emploi de classe locale, ainsi que l'entête `fstream`, car notre système utilise le flux `cout` pour l'affichage :

```
#include <fstream>
#include <locale>
using namespace std;
```

À présent, nous définissons une structure (classe) `Monnaie` pour que `cout` fasse la distinction entre l'affichage d'un `double` - sans unité - et l'affichage d'une donnée de type `Monnaie` :

```
struct Monnaie
{
    Monnaie (const double montant)
        : m(montant) {}

    double m;
};
```

Vous aurez sans doute remarqué le style d'initialisation de la variable `m`, très courant en C++ (c'est comme si l'on utilisait le constructeur de la variable `m`).

Nous poursuivons par l'écriture d'une facette destinée à formater la donnée pour l'affichage. Il est facile de sous-classer cette facette en vue de la rendre paramétrable.

```
class monnaie_put: public locale::facet
{
public:
    static locale::id id;
```

```

monnaie_put (std::size_t refs = 0)
: std::locale::facet (refs) { }

string put (const double &m) const
{
    char buf[50];
    sprintf(buf,"%g Euro",m);
    return string(buf);
}
};

locale::id monnaie_put::id;

```

Suivant l'organisation de votre programme, la définition du champ `monnaie_put::id` devra se faire dans un fichier `.cpp` séparé.

Nous allons maintenant surcharger l'opérateur `<<` pour supporter l'insertion d'un objet `Monnaie` dans un flux `ostream`. S'agissant d'une structure dont tous les champs sont publics, la déclaration d'amitié n'est pas nécessaire.

Nous obtenons directement :

```

ostream& operator<< (ostream& os, const Monnaie& mon)
{
    std::locale loc = os.getloc ();
    const monnaie_put& ppFacet
        = std::use_facet<monnaie_put> (loc);
    os << ppFacet.put(mon.m);
    return (os);
}

```

Il ne nous reste plus qu'à appliquer cette construction et à tester le programme.

```

int main(int argc, char* argv[])
{
    cout.imbue (locale (locale::classic (), new monnaie_put));
}

```

```
Monnaie m(30);  
cout << m; // affiche 30 Euro  
return 0;  
}
```