Mémento pour débuter en C++

Algorithmique effective

2005–2006

1 Édition, compilation, exécution

Le *code source* est l'ensemble des instructions que l'on écrit dans un fichier ayant l'extension `.cpp`. Il faut ensuite le *compiler* pour obtenir un programme exécutable par la machine. À chaque fois que l'on veut exécuter le programme, s'il y a eu des modifications, on enregistre le fichier source, on compile, puis on exécute.

Pour éditer le code source :
`emacs mon_fichier.cpp &`

Pour compiler (si l'alias `G++='g++-3.4 -Wall -W -pedantic'` a été créé) :
`G++ -o mon_programme mon_fichier.cpp`
Si l'alias n'a pas été créé, la ligne précédente ne marche pas et il faut alors taper
`g++-3.4 -Wall -W -pedantic -o mon_programme mon_fichier.cpp`

Un programme exécutable appelé `mon_programme` est créé. Pour l'exécuter :
`./mon_programme`

Si le programme lit l'entrée standard (`std::cin`), il va alors attendre que l'on rentre des données au clavier. S'il écrit sur la sortie standard (`std::cout`), il va afficher des données à l'écran. Pour éviter d'avoir toujours à taper au clavier, on peut lui fournir un fichier texte en entrée : il s'agit d'une redirection, car `std::cin` lit alors le fichier texte plutôt que le clavier. Exemple : lecture dans le fichier `texte.in`
`./mon_programme < texte.in`
On peut aussi remplacer la sortie `std::cout` par un fichier (ici `texte.out`), notamment lorsque trop de choses seraient écrites en sortie :
`./mon_programme > texte.out`
Enfin, on peut combiner les deux :
`./mon_programme < texte.in > texte.out`

2 Structure d'un programme C++

Un programme C++ commence par des *en-têtes* : ce sont des fichiers qui contiennent les instructions de bases et qu'il convient d'inclure pour pouvoir utiliser ces instructions. Les en-têtes les plus usuels sont

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
```

Viennent ensuite la déclaration des *fonctions* :

```
type_de_retour f(type_1 argument_1, ..., type_n argument_n)
{
  le code de la fonction ici
}
```

La fonction `f` prend alors n arguments (éventuellement $n = 0$), et renvoie un élément de type `type_de_retour`. Les principaux types sont : `int` pour les entiers, `float` pour les nombres flottants, `char` pour les caractères, `bool` pour les booléens (`true` et `false`), et enfin `void` pour ne rien renvoyer. Une fonction renvoie une valeur par l'instruction `return`. Par exemple, pour sortir d'une fonction de type `int` en renvoyant la valeur 10, on écrit

```
return 10;
```

La fonction principale s'appelle `main`. C'est elle qui est appelée au début du programme. Son type de retour est `int`, mais elle fait exception et on peut considérer qu'elle ne prend aucun argument et qu'elle ne renvoie rien.

```
int main()
{
  le code de main ici
}
```

Pour que `main` puisse appeler la fonction `f` (par l'instruction `f(1, 2);` par exemple, si `f` prend deux arguments de type `int`), cette dernière doit être définie avant. Remarquer que le code d'une fonction se trouve toujours entre l'accolade ouvrante de début de fonction et l'accolade fermante de fin de fonction.

Les *blocs* sont ainsi délimités par des accolades (par exemple, la définition d'une fonction, le corps d'une boucle, ou d'une instruction `if`). Les lignes d'instructions, quant à elles, se terminent par un point-virgule.

3 Variables

Pour définir une *variable*, la syntaxe est

```
type nom;
```

On peut aussi initialiser la variable lors de sa définition, et même définir plusieurs variables du même type d'un coup, en les séparant par des virgules : par exemple

```
int i = 0, j = 1, k;
```

Si une variable n'est pas initialisée (par exemple, `k` ici), elle a une valeur quelconque (ce qu'il y a en mémoire à ce moment-là).

On peut facilement définir des tableaux *de taille fixe* (la taille doit être explicitement entrée à la main, et ne doit pas être une variable), contenant des éléments de même type :

```
type tableau[taille];
```

Attention, les indices des tableaux commencent à zéro : la zone valide est donc de `tableau[0]` à `tableau[taille-1]`.

Un exemple de définition d'un tableau `t` de 100 entiers :

```
int t[100];
```

On peut accéder à l'élément `i` du tableau (`i` est compris entre 0 et `taille-1`) par l'instruction `t[i]` ; par exemple, pour initialiser le premier élément à 1 :

```
t[0] = 1;
```

Une *chaîne de caractères* de base (il existe aussi des chaînes plus sophistiquées, de type `string`) est simplement un tableau de caractères. La fin d'une chaîne est marquée par le caractère `'\0'`, il faut donc penser à allouer une case supplémentaire pour stocker ce caractère. À part pour la taille du tableau, on ne se soucie en général pas de ce dernier caractère, car les instructions qu'on utilise sur les chaînes s'en chargent automatiquement. On peut par exemple définir une chaîne `"cpp"` par

```
char chaine[4];
chaine[0] = 'c'; chaine[1] = 'p';
chaine[2] = 'p'; chaine[3] = '\0';
```

ou alors, de manière équivalente mais plus simple, par
`char chaine[4] = "cpp";`

4 Conditions, boucles

4.1 Branchement conditionnel

En C++, on peut faire des *tests* : ils renvoient un booléen. Pour tester l'égalité, on utilise l'opérateur `==` (il s'agit ici d'une *comparaison*, à ne pas confondre avec l'affectation `=`). Par exemple, `i == 3` vaut `true` si la variable `i` a la valeur 3, et `false` sinon.

Il y a aussi les opérateurs `!=` (différent, par exemple `i != 3` est faux ssi `i` vaut 3), `<`, `<=`, `>` et `>=`. La négation d'un test s'obtient par l'opérateur `!`. La conjonction de deux tests s'obtient par `&&` et la disjonction par `||`. Par exemple, le test suivant signifie que `i` est dans l'ensemble $]-\infty, 3[\cup[5, 8[\cup]8, +\infty[$:
`(i < 3) || ((i >= 5) && !(i == 8))`

L'instruction `if` s'utilise de la manière suivante :

```
if (condition)
  {
    le code si la condition est vraie
  }
else
  {
    le code si elle est fausse
  }
```

4.2 Boucles

Il y a principalement deux types de *boucles* : `for` et `while`. `for` s'utilise surtout lorsqu'il s'agit de boucler un nombre déterminé de fois, en incrémentant un compteur. Cependant, en C++ elle a la puissance de `while`. Pour du code propre, éviter d'utiliser `for` à la place de `while`. La syntaxe est

```
for (initialisation; condition; incrementation)
  {
    le code a repeter
  }
```

`condition` est évaluée à chaque passage en début de boucle, et si elle est vraie, `incrementation` est exécutée puis on commence le code de la boucle. Si elle est fausse, on quitte la boucle sans rien exécuter. L'utilisation typique est
`for (int i = 0; i < n; ++i)`
qui boucle `n` fois (pourvu que `i` et `n` ne soient pas modifiés dans la boucle). L'instruction `++i` incrémente `i` d'une unité. Il y a aussi `--i` pour le décrémenter. Plus généralement, on peut utiliser `i = i+1` ou `i = i-1` de manière (presque) équivalente.

Une boucle de `n-1` à 0 s'obtient donc logiquement par
`for (int i = n-1; i >= 0; --i)`

Passons maintenant aux boucles `while`. Leur syntaxe est

```
while (condition)
  {
    le code a repeter
  }
```

Une telle boucle évalue `condition`, et si elle est vraie exécute le code à répéter. Sinon, on sort de la boucle sans exécuter le code. C'est donc une boucle qui répète les instructions *tant que* la condition est vraie.

5 Entrées et sorties

5.1 Entrée standard

Comme expliqué à la section 1, lorsqu'on lit des entrées sur l'entrée standard (`std::cin`), par défaut le programme attend des caractères tapés au clavier. Si on redirige l'entrée standard, il peut aussi prendre les entrées dans un fichier. Il existe des commandes simples et puissantes en C++ pour lire l'entrée standard.

Les séparateurs (espaces, tabulations, retours à la ligne) ont un traitement particulier par rapport aux autres caractères. Supposons que l'entrée comporte une chaîne de caractères représentant un entier. On peut lire ce caractère grâce à l'instruction
`std::cin >> i;`
Alors on a récupéré l'entier dans la variable `i`, si celle-ci est de type `int`. L'instruction `std::cin >> i;` saute tous les séparateurs éventuels du début de l'entrée, lit et stocke l'entier `i`, et se place juste après celui-ci.

Si toutefois il existe des caractères en début d'entrée qui ne sont pas des séparateurs, ni ne représentent un entier, alors l'instruction ci-dessus se bloque, car elle attend un entier.

De même, on peut lire d'autres types de données, par exemple un nombre flottant (si la variable a pour type `float`), un caractère (`char`), ou un mot quelconque (si l'on a un tableau de caractères). On peut aussi lire plusieurs entités en une commande : par exemple, on peut lire deux entiers puis un nombre flottant par

```
int i, j;
float f;
std::cin >> i >> j >> f;
```

Ce qu'il faut retenir : la commande `std::cin >> i;` saute les espaces et les retours à la ligne, lit seulement une instance du type de `i` (elle se bloque s'il n'y en a pas), et se positionne juste après l'instance lue.

5.2 Sortie standard

Pour afficher quelque chose en sortie (écran par défaut, ou fichier si redirection), il faut cette fois gérer soi-même les espaces, retours à la ligne, etc. La sortie standard s'appelle `std::cout`, et la syntaxe ressemble à la lecture de l'entrée standard (sauf que la direction de la "flèche" est inversée). Pour afficher un entier `i`, l'instruction est
`std::cout << i;`
Seul l'entier est affiché, aucun espace ni retour à la ligne n'est ajouté. On peut faire de même pour une chaîne de caractères, un caractère, un nombre flottant, etc. Un retour à la ligne s'obtient par `std::endl`. Par exemple,

```
float f = -.0520;
std::cout << "Voici la valeur de f : " << f << "." << std::endl;
```

affiche `Voici la valeur de f : -0.052.` et passe à la ligne.

6 Remarques finales

1. Une bonne habitude pour relire facilement le code est d'inclure des commentaires dans celui-ci. Il y a deux façon d'écrire des commentaires dans le code source :

```
/* Methode 1 :
   Voici un commentaire
   sur plusieurs lignes */
int i; // Methode 2 : commentaire sur la fin d'une ligne
```

2. Au début du fichier source, après les `#include`, on peut définir des constantes par
`#define N 100`
On utilise ensuite `N` plutôt que `100`. Cela donne un code plus lisible, et on peut facilement modifier la valeur de `N`. Dès que le compilateur lit `N` dans le code source, il le remplace alors par `100`. `N` n'est pas traité comme une variable : tout se passe vraiment comme si l'on avait écrit `100` à la place de `N`. En particulier, on peut s'en servir pour définir la taille d'un tableau.
3. Une référence à lire sans hésiter : *Le langage C*, de Kernighan et Richie.

7 Un exemple simple

Voici un programme pour l'exercice 1.1 du TD 1. Il s'agit de lire en entrée des nombres flottants, et de les écrire en sortie dans l'ordre inverse.

```
#include <iostream> // 3 en-tetes a inclure
#include <stdlib.h>
#include <stdio.h>

#define N 1100 // constante : taille du tableau

int main()
{
  int n, m; // nombre de lignes, nombre de flottants par ligne
  float tableau[N]; // pour stocker la ligne en cours

  std::cin >> n; // lecture du nombre de lignes
  std::cout << n << std::endl; // on affiche n

  for (int i=0; i<n; ++i) // pour chaque ligne
    {
      std::cin >> m; // nombre d'entiers sur la ligne

      for (int j=0; j<m; ++j) // les stocker dans tableau
        {std::cin >> tableau[j];}

      for (int j=m-1; j>0; --j) // les afficher a l'envers
        {std::cout << tableau[j] << ' ';}

      std::cout << tableau[0] << std::endl; // dernier cas separe
                                            // pour eviter un espace
                                            // a la fin de la ligne
    }
}
```