Chapitre 2 : variables, entrée/sortie dans un fichier, nombres aléatoires

1 Quelques mots sur les déclarations de variables

1.1 Organisation de la mémoire

La mémoire de l'ordinateur peut être vue comme un long ruban constitué de cases mémoire, c'est-à-dire d'emplacement dans lequel on peut enregistrer une suite de bits de longueur fixée. Les cases mémoire ont quasi-systématiquement une taille de 8 bits, ce qui s'appelle également un *octet*.

À chaque déclaration de variable, on réserve un certain nombre de cases mémoire, correspondant à la place nécessaire pour stocker la variable en question. Par exemple, sur une machine stockant les `int` sur 32 bits et les `double` sur 64 bits, la déclaration d'un `int` va réserver 4 cases mémoire, et la réservation d'un `double` va en réserver 8.

On peut obtenir l'adresse mémoire d'une variable `x` par la syntaxe `&x`. On verra plus tard que `&x` est un objet de type *pointeur*. On peut afficher l'adresse d'une variable (sous forme hexadécimale) en l'envoyant vers un flux de sortie.

1.2 Visibilité des variables

On appelle *visibilité d'une variable* le fait qu'une variable définie dans un programme ne puisse pas être utilisée à n'importe quel moment. La visibilité d'une variable va dépendre des conditions dans laquelle elle a été déclarée.

Il existe trois types de variables :

— Les variables globales : il s'agit de variables qui ont été déclarées hors de toute fonction. Elles sont accessibles à tout moment du programme. Attention : ce type de variable peut assez facilement être modifié sans que l'on s'en aperçoive. En général, on va donc déclarer les variables globales comme étant constantes (voir partie 1.3). De manière générale, on conseille de limiter l'usage des variables globales. Typiquement, on se servira de variables globales pour entrer les paramètres d'un problème qui ne seront pas modifiés pendant l'exécution.

— Les arguments de fonction : ils ne sont accessible qu'à l'intérieur de la fonction correspondante. Il est important de comprendre qu'à chaque appel de la fonction, on définit une nouvelle variable, visible uniquement dans cette fonction, dans laquelle on recopie la valeur passée en argument de la fonction. Cela a notamment pour effet qu'*une fonction ne peut pas modifier ses arguments*. On verra plus tard qu'une modification des arguments nécessitera de passer par des pointeurs ou des références.

— Les variables locales : on appelle *bloc* tout groupe d'instruction situé entre deux accolades. Toute variable définie à l'intérieur d'un bloc n'est visible que dans ce bloc.

Dans un même programme, plusieurs variables peuvent avoir le même nom. Le principe est que si deux variables ont le même nom, la "plus locale" des deux va masquer la "plus globale". Par exemple, si on dispose d'une variable globale `x`, et qu'une fonction possède un argument également noté `x`, alors le nom `x` désignera, dans la fonction, l'argument formel. La variable globale ne sera donc pas utilisable

dans cette fonction. De même, si une fonction possédant un argument `x` comprend un bloc dans lequel est déclarée une variable `x`, alors, hors du bloc, `x` désignera l'argument de la fonction, et dans le bloc `x` désignera la variable locale définie dans le bloc.

Bien entendu, multiplier les variables différentes ayant le même nom n'est pas à conseiller pour avoir un code lisible.

1.3 Qualificateurs `const` et `static`

Une variable peut être déclarée *constante* par la syntaxe "`type const nom;`". Cela a pour effet de créer une erreur à la compilation si la variable est modifiée dans le programme. On conseille de déclarer constante toutes les variables dont on sait qu'elles ne doivent pas être modifiée, afin de détecter toute modification accidentelle.

Une variable locale peut être déclarée *statique*, par la syntaxe "`static type nom;`". Cela ne modifie pas la visibilité de la variable, mais lui donne une durée de vie permanente. Plus précisément, la variable ne va pas être réinitialisée à chaque passage dans le bloc où elle est déclarée. Cela permet par exemple de compter le nombre d'appel à une fonction, ou encore de garder la valeur courante d'un générateur de nombres aléatoires (voir partie 3).

2 Écriture et lecture dans un fichier

Il est possible de lire et d'écrire dans un fichier, avec une syntaxe proche de la lecture et l'écriture vers la sortie standard (les opérateurs `>>` et `<<`). Les fonctions nécessaires sont présentes dans la bibliothèque `fstream`. Dans cette bibliothèque sont définis les deux types `std::ofstream` (sortie (out) vers un fichier) et `std::ifstream` (entrée (in) depuis un fichier).

On peut donc définir un flux de sortie vers un fichier par la syntaxe "`std::ofstream mon_flux;`". On ouvre alors un fichier, vers lequel le flux pointera, avec l'instruction "`mon_flux.open("mon_fichier")`". De même, on referme le fichier à l'aide de l'instruction "`mon_flux.close()`". Il est également possible d'ouvrir un fichier dès la déclaration avec la syntaxe "`std::ofstream mon_flux("mon_fichier");`"

Il n'est pas indispensable de refermer explicitement le fichier (il sera de toutes façons fermé à la fin de l'exécution du programme), mais c'est un bon réflexe : si un programme manipule de nombreux fichiers, il est préférable de refermer les fichier dont on n'a plus l'utilité.

On verra plus tard que `std::ofstream` et `std::ifstream` sont en fait des *classes*, que la syntaxe `std::ofstream flux("mon_fichier")` est en fait un appel de constructeur, et que les instructions `flux.open("fichier")` et `flux.close()` sont des appels de méthodes.

3 Génération de nombres aléatoires

3.1 La théorie

Un ordinateur ne peut pas générer des nombres au hasard. Toutefois, il y a certaines situations (méthode de Monte Carlo, cryptographie,...) où l'on voudrait obtenir une suite de nombres qui "ressemble" à une suite de nombres aléatoires. On a donc mis au points des algorithmes *totalement déterministes*, mais qui génèrent des suites vérifiant certaines propriétés vérifiées par des suites de variables aléatoires indépendantes. Ces suites sont appelées des suites de nombres *pseudo-aléatoires*.

En pratique, tous ces algorithmes génèrent une suite d'éléments de $\{1, \dots, N\}$ où N est très grand, qui "ressemble" à une suite de variables indépendantes et de loi uniforme sur $\{1, \dots, N\}$. À partir de là, il est possible de simuler approximativement une variable uniforme sur $[0, 1]$: si X est une variable de loi uniforme sur $\{1, \dots, N\}$, alors X/N est approximativement une variable de loi uniforme sur $[0, 1]$.

Voici un exemple [1]. On définit une suite d'éléments de $\{1, \dots, 2^{31} - 1\}$ par :

$$\begin{cases} x_0 & \text{donné,} \\ x_{n+1} & = (2^{16} + 3) \times x_n \text{ modulo } 2^{31}. \end{cases}$$

On peut montrer que la suite $(x_n)_{n\in\mathbb{N}}$ est périodique de période $2^{29} \simeq 5 \times 10^8$. En fait, toute suite de nombres pseudo-aléatoire est périodique, puisqu'il s'agit de l'itération d'une fonction donnée d'un ensemble fini dans lui-même. Pour être un "bon" générateur, un générateur doit (entre autres) avoir une longue période. Ici, la période 2^{29} peut être suffisante pour bon nombre de problèmes.

Toutefois une autre condition est que les corrélations doivent être faibles : une petite modification de x_n (et de quelques valeurs précédentes) doit pouvoir amener une grande modification de x_{n+1}. Ce n'est pas le cas du générateur présenter ici : en effet, la suite $(x_n)_{n\in\mathbb{N}}$ vérifie la relation $x_{n+2} = 6x_{n+1} - 9x_n$, de sorte qu'une petite modification de x_n et de x_{n+1} ne peut entraîner qu'une petite modification de x_{n+2}. Par conséquent, la suite des (x_n, x_{n+1}, x_{n+2}) ne va pas du tout être uniformément répartie dans l'espace à trois dimensions.

Enfin, il est à noter que la suite $(x_n)_{n\in\mathbb{N}}$ est en fait indexée par le paramètre x_0, appelé la *graine* du générateur. On obtient plusieurs suites différentes en partant de plusieurs graines différentes. Toutefois, deux suites partant de deux graines différentes *ne doivent surtout pas être considérées commme des suites indépendantes.*

3.2 Les implémentations

3.2.1 `rand`

La méthode la plus classique pour générer des nombres aléatoires est d'utiliser la fonction `rand` de la bibliothèque `cstdlib`.

Toutefois, ce générateur peut être insuffisant pour certaines applications (méthode de Monte Carlo nécéssitant une longue période). Notamment, suivant les implémentations, sa période peut varier. Par exemple, dans certaines implémentations, `rand` a une période de $2^{15} \simeq 32.000$, ce qui est *beaucoup trop peu.*

On peut choisir la graine du générateur de `rand` avec la fonction `srand`. Le résultat de `rand` est un entier compris entre `0` et `RAND_MAX` inclus.

3.2.2 Mersenne twister

Depuis le standard 2011 [2], le générateur dit "Mersenne twister" est implémenté pour le `C++`. Ce générateur est très utilisé pour les calculs de Monte Carlo, notamment car sa période est extrèmement grande ($2^{19937} - 1 \simeq 10^{6000}$). Il n'est toutefois pas sûr pour un usage cryptographique.

Pour utiliser ce générateur, on va se servir du type `std::mt19937`, défini dans la bibliothèque `random`. À la déclaration, on peut écrire `std::mt19937 gen(1234)` pour déclarer une variable `gen` correspondant

1. Ce générateur, appelé RANDU, est réputé être *très mauvais.* À ne pas utiliser.
2. Pour compiler un programme avec `g++` en utilisant le standard 2011 du `C++`, il faut ajouter l'option "`-std=c++11`" lors de la compilation.

à un générateur de graine 1234. On obtiendra la suite pseudo-aléatoire par des appels successifs de `gen()`. La valeur maximale rendue par `gen` peut être obtenue par `gen.max()`. Pour conserver l'état courant du générateur, il se peut que la variable `gen` doive être déclarée `static`.

On verra plus tard que `std::mt19937` est en fait une classe, dont `std::mt19937 x(int)` est un constructeur, et que `x()` correspond à l'appel de l'opérateur `()`.