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

Master 2 IFMA – Programmation en C++

Université Pierre et Marie Curie

Automne 2016

Plan

Organisation de la mémoire

- Mémoire de l'ordinateur : long "ruban" de cases mémoire= suites de 8 bits (=1 *octet*).

Déclaration de variable : réservation d'un certain nombre de cases mémoire (place nécessaire pour stocker la variable).
Exemples :

- `bool` = 1 bit = 1 case mémoire ;
- `char` = 8 bits = 1 case mémoire ;
- `int` = 32 bits = 4 cases mémoire ;
- `double` = 64 bits = 8 cases mémoire.

- adresse mémoire d'une variable `x` accessible par `&x` ;
- `&x` est un objet de type *pointeur* ;
- `std::cout << &x` : affiche l'adresse de la variable.

Visibilité des variables

Visibilité d'une variable : le fait qu'une variable ne puisse pas être utilisée à n'importe quel moment. La visibilité dépend des conditions dans laquelle la variable a été déclarée.

Trois types de variables :

Variables locales

- Une variable définie à l'intérieur d'un bloc n'est visible que dans ce bloc ;
- *Bloc* = groupe d'instructions situé entre deux accolades ;
- Utilisation typique : paramètre d'une boucle.

Visibilité des variables

Visibilité d'une variable : le fait qu'une variable ne puisse pas être utilisée à n'importe quel moment. La visibilité dépend des conditions dans laquelle la variable a été déclarée.

Trois types de variables :

Arguments de fonction

- Accessible seulement à l'intérieur de la fonction correspondante ;
- Chaque appel de la fonction définit une nouvelle variable dans laquelle on recopie la valeur passée en argument de la fonction ;
- Notamment, 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.

Visibilité des variables

Visibilité d'une variable : le fait qu'une variable ne puisse pas être utilisée à n'importe quel moment. La visibilité dépend des conditions dans laquelle la variable a été déclarée.

Trois types de variables :

Variables globales

- Déclarées hors de toute fonction ;
- Accessibles à tout moment ;
- Attention : ce type de variable peut être modifié sans que l'on s'en aperçoive (par un appel de fonction).
- De préférence, déclarer les variables globales comme étant constantes.
- Usage typique : paramètres d'un problème à ne pas modifier.

Visibilité des variables

- Dans un même programme, plusieurs variables peuvent avoir le même nom.
- La “plus locale” des deux va masquer la “plus globale”.

Trop multiplier les variables différentes ayant le même nom est une mauvaise idée pour avoir un code lisible.

Qualificateur `const`

Une variable peut être déclarée *constante* par la syntaxe

```
type const nom;
```

Cela provoque une erreur à la compilation si la variable est modifiée dans le programme : c'est une aide pour l'écriture du programme.

Utilisation

On conseille de déclarer constante toutes les variables dont on sait qu'elles ne doivent pas être modifiée, afin de détecter toute modification accidentelle.

Qualificateur `static`

Une variable locale peut être déclarée *statique*, par la syntaxe

```
static type nom;
```

Donne une durée de vie permanente à la variable, sans changer sa visiblilité : la variable ne va pas être réinitialisée à chaque passage dans le bloc où elle est déclarée.

Exemples

Compter le nombre d'appel à une fonction ; garder la valeur courante d'un générateur de nombres aléatoires.

Plan

Écriture et lecture dans un fichier

- bibliothèque `fstream`.
- deux *types* `std::ofstream` (sortie (out) vers un fichier) et `std::ifstream` (entrée (in) depuis un fichier).

```
std::ofstream mon_flux; //On déclare un flux de sortie
mon_flux.open("mon_fichier") // On ouvre un fichier
mon_flux << "Bonjour\n"; // On écrit dans le fichier
mon_flux.close(); // On referme le flux
```

On peut aussi ouvrir le fichier dès la déclaration :

```
std::ofstream mon_flux("mon_fichier");
```

Refermer le fichier n'est pas indispensable,, mais c'est un bon réflexe, par exemple si on manipule de nombreux fichiers.

Écriture et lecture dans un fichier

On verra plus tard que `std::ofstream` et `std::ifstream` sont en fait des *classes*, que

```
std::ofstream mon_flux("mon_fichier")
```

est un appel de constructeur, et que

```
mon_flux.open("fichier")
```

et

```
mon_flux.close()
```

sont des appels de méthodes.

Plan

Nombres aléatoires : la théorie

- Ordinateur : aléa impossible.
- *Mais* : il existe des algorithmes *totalement déterministes*, générant des suites vérifiant certaines propriétés statistiques.
- On parle de suites de nombres *pseudo-aléatoires*.

En pratique

- On calcule $x_n \in \{1, \ldots, N\}$, avec N grand, "ressemblant" à une suite i.i.d. de loi uniforme sur $\{1, \ldots, N\}$.
- Si X est uniforme sur $\{1, \ldots, N\}$, alors X/N est approximativement uniforme sur $[0, 1]$.

Un (*mauvais*) exemple : RANDU

On définit $x_n \in \{1, \ldots, 2^{31} - 1\}$ par :

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

$(x_n)_{n \in \mathbb{N}}$ est $2^{29} \simeq 5 \times 10^8$-périodique. 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.

Un (*mauvais*) exemple : RANDU

- Autre condition : corrélations faibles
- Ici : pour tout n $x_{n+2} = 6x_{n+1} - 9x_n$. La suite des (x_n, x_{n+1}, x_{n+2}) n'est pas du tout uniformément répartie dans $[0,1]^2$.

Remarque

La suite $(x_n)_{n\in\mathbb{N}}$ est indexée par le paramètre x_0, appelé *graine* du générateur.
On obtient plusieurs suites différentes en partant de plusieurs graines différentes.
Deux suites partant de deux graines différentes *ne doivent par contre pas être considérées commme des suites indépendantes*.

Les implémentations : `rand`

La méthode simple

- Fonction `rand` de la bibliothèque `cstdlib`.
- Insuffisant pour certaines applications (méthode de Monte Carlo nécéssitant une longue période).
- Dans certaines implémentations, `rand` a une période de $2^{15} \simeq 32.000$: *beaucoup trop peu*.
- Choix de la graine avec `srand`. Le résultat de `rand` est un entier compris 0 et `RAND_MAX` inclus.

Les implémentations : Mersenne twister

La méthode efficace

- Depuis le standard 2011 (avec g++, option "`-std=c++11`"), le générateur dit "Mersenne twister" est implémenté ;
- Très utilisé pour les calculs de Monte Carlo (période extrêmement grande : $2^{19937} - 1 \simeq 10^{6000}$) ;
- Pas sûr pour un usage cryptographique.

Les implémentations : Mersenne twister

La méthode efficace

- Type `std::mt19937`, défini dans la bibliothèque `random` ;
- `std::mt19937 gen(1234)` déclare une variable `gen` correspondant à un générateur de graine 1234. Pour conserver l'état courant du générateur, il se peut que la variable `gen` doive être déclarée `static` ;
- Suite pseudo-aléatoire obtenue par des appels successifs de `gen()` ;
- La valeur maximale rendue est `gen.max()` ;
- 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 `()`.