Déclarations, types et identificateurs

Les variables

Comme la plupart des langages de programmation, le C++ utilise la notion de variable. Une variable peut être vue comme une zone de la mémoire qui comprend une certaine valeur.

Les types en C++ et les systèmes de représentation

Le langage C++ impose un mécanisme de type pour indiquer la nature des données contenues dans une variable. Ainsi un `double` permettra de stocker un réel et un `int` permettra de stocker un entier. Par contre, il ne définit pas de système de représentation pour représenter ces variables. Ainsi, le standard ne spécifie pas comment on représente un double sous la forme d'une suite de bits. Le système de représentation utilisé peut donc varier entre deux ordinateurs ou en fonction du compilateur utilisé. Cette particularité peut parfois poser de graves problèmes de portabilité d'un programme.

Les déclarations

Toute variable en C++ doit être déclarée : la déclaration indique l'identificateur de la variable (son nom) et sa nature (son type).

Syntaxe :

```
type identificateur;
```

Exemple :

```
int a;
```

Cette déclaration définit une variable d'identificateur a qui contient un entier de type int.

Identificateurs valides

Un identificateur est une suite de caractères (pouvant être majuscules ou minuscules), de chiffres ou d'underscores (underscore ou "tiret bas" est le caractère _). Cette suite ne peut pas commencer par un chiffre. Un identificateur ne peut contenir ni espace, ni tiret - (utilisé pour l'opération de soustraction).

Les types entiers

Le langage C++ possède plusieurs types de base pour désigner un entier.

- **int** : contient un entier de taille normale, positif ou négatif.
- **short int** : contient un entier de petite taille, positif ou négatif.
- **long int** : contient un entier de grande taille (32 bits), positif ou négatif.
- **long long int** : contient un entier de plus grande taille (64 bits), positif ou négatif.
- **unsigned int** : contient un entier de taille normale, positif ou nul.

- **unsigned short int** : contient un entier de petite taille, positif ou nul.
- **unsigned long int** : contient un entier de grande taille (32 bits), positif ou nul.
- **unsigned long long int** : contient un entier de plus grande taille (64 bits), positif ou nul.

La longueur d'un long int , d'un int et d'un short int n'est pas spécifié par le langage. Plus un entier est représenté sur un grand nombre de bits, plus il pourra être grand. Ainsi, il est usuel de représenter un int sur 32 bits : il peut alors représenter n'importe quel entier entre -2^{31} et $2^{31}-1$. Le langage impose juste que la taille d'un long int doit être supérieure ou égale à celle d'un int et que la taille d'un int doit être supérieure ou égale à celle d'un short int !

Le système de représentation utilisé non plus. En général, la base 2 est utilisée pour les types unsigned int, unsigned long int et unsigned short int et le complément à 2 est utilisé pour les types int, long int et short int. Ce n'est toutefois nullement obligatoire.

Interprétation des constantes entières

- si une constante commence par 0x, elle sera interprétée comme une valeur en hexadécimal (base 16).
- si une constante commence par 0 suivi d'un chiffre, elle sera interprétée comme valeur en octal (base 8).
- dans le cas contraire, elle sera interprétée comme étant en base 10.

Exemples :
98 représente 98 en base 10.
0x62 représente 98 en hexadécimal (6*16+2).
0142 représente 98 en octal (1*64+4*8+2).

- si une constante entière se termine par un U, elle sera interprétée comme étant un unsigned.
- si une constante entière se termine par un L, elle sera interprétée comme un long.
- si une constante entière se termine par LL, elle sera interprétée comme un long long.

Exemples

- 78U représente le unsigned int valant 78.
- 78L représente le long int valant 78.
- 78LL représente le long long int valant 78.
- 78ULL représente le unsigned long long int valant 78.

Les types réels

Pour représenter un réel, il existe 3 types de base :

- float (simple précision)
- double (double précision)
- long double (précision étendue)

Le langage ne précise pas ni le système de représentation, ni la précision de ces différents formats. Le type long double est juste censé être plus précis que le double, lui-même plus précis que le float.
Il est toutefois usuel (mais non obligatoire) de représenter le float sur 32 bits dans le format IEEE 754 simple précision et le double sur 64 bits dans le format IEEE 754 double précision.

Interprétation des constantes réelles

- On peut écrire un réel sous la forme 1.87 ou .56 ou 8. ou 7.6e10 ou 5.4e-3 ou encore 10.3E+2.
- Une constante qui se termine par un f ou un F sera interprétée comme un float.
- Une constante qui se termine par un l ou un L sera interprétée comme un long double.

Exemples :

- 3.65L représente le long double valant 3.65.
- 3.65F représente le float valant 3.65.

Les caractères

Le caractère est l'élément de base de tout texte et donc sans doute de toute pensée. Les changements de l'informatique ont conduit à différentes approches pour représenter un caractère sur cinq, sept, huit, seize, dix-sept ou trente-deux bits.

Aujourd'hui, les deux types incontournables sont le type char hérité du C, et ceux relatifs à Unicode.

Ce chapitre traite des types de bases utiles pour représenter des caractères. Toutefois les caractères sont rarement utilisés seuls, surtout dans les applications Unicode. A ce sujet, vous trouverez dans la suite de cet ouvrage les chapitres suivants :

- Programmation C++/La_librairie_standard#Les chaînes de caractères
- Programmation C++/Les_tableaux#Les tableaux de caractères

Le type char

Il s'agit du type historique pour représenter un caractère. Bien que le type char fasse penser à un caractère (*character* en anglais), il désigne souvent de facto un octet, qui peut être signé ou non signé suivant le compilateur. C'est l'un des concepts que le C++ a repris du langage C. Aucun système de représentation n'est imposé pour les caractères et on utilise en général des dérivés (8 bits) du code ASCII (qui est un code sept bits). [1] Usuellement, le type char est exactement 8 bits, ce qui fait que c'est le seul type utilisé pour représenter un octet, créant ainsi une confusion entre caractère et octet. Le chapitre suivant s'intéresse au huitième bit.

Exemple :

```
char c;
c = 'E';
```

Le caractère E est transféré dans la variable c de type char .
On peut transférer un char dans un int pour récupérer son codage.
Ainsi, on peut écrire :

```
int a;
char b;
b = 'W';
a = b;
```

On récupère alors dans a le codage du caractère 'W'. On récupérera donc en général le code ASCII de 'W' dans la variable a.

Aujourd'hui, il est désuet de considérer que l'on code tous caractères sur un seul char. Le type char reste cependant incontournable car il est souvent utilisé pour désigner un octet.

Transformation de majuscule en minuscule

```
#include<iostream>
using namespace std;

int main()
{
  char a, b;
  cout<<"Tapez un caractere : "; cin>>a;
  if (a>='A' && a<='Z') {
    cout<<"Vous avez tapé une majuscule."<<endl;
```

```
    b = a + ('a'-'A');
    cout<<"La minuscule correspondante est "<< b <<endl;
  }
  else if (a>='a' && a<='z') {
    cout<<"Vous avez tapé une minuscule."<<endl;
    b = a + ('A'-'a');
    cout<<"La majuscule correspondante est "<< b <<endl;
  }
  else cout<<"Vous n'avez pas tapé une lettre."<<endl;
  return 0;
}
```

- **Explications**
  - On demande à l'utilisateur de taper un caractère dans une variable a.
  - Si l'utilisateur a tapé une majuscule, on affiche la minuscule correspondante.
  - Si l'utilisateur a tapé une minuscule, on affiche la majuscule correspondante.
- **Exécution 1**

Tapez un caractère : **H**
Vous avez tapé une majuscule.
La minuscule correspondante est h.

- **Exécution 2**

Tapez un caractère : **w**
Vous avez tapé une minuscule.
La majuscule correspondante est W.

- **Exécution 3**

Tapez un caractère : **9**
Vous n'avez pas tapé une lettre.

Les types signed char et unsigned char

Lorsqu'on transfère un char dans un int, peut-on récupérer une valeur négative ? La réponse est oui si on utilise le type **signed char** et non si on utilise le type **unsigned char**. Ces types peuvent être utile lorsqu'on manipule des caractères non ASCII.

Pour les données de type char, lorsque ni signed ni unsigned ne sont précisés, le choix entre les deux est fait par le compilateur. Dans tous les cas à l'époque où seuls les codages ASCII et autres codages ISO-646 étaient utilisés cela n'avait pas d'importance.

Aujourd'hui cependant, quasiment tous les codages de caractères utilisent a minima huit bits. C'est notamment le cas d'UTF-8. L'éventuel bit de signe doit donc être considéré pour permettre la portabilité du logiciel.

Les types `char16_t` and `char32_t`

À partir de C++11 (C++ norme de 2011) trois types de chaînes de caractères sont prise en charge: UTF-8, UTF-16, et UTF-32. Le type `char` conserve ses unités de codage de huit bits pour le codage des caractères Unicode via UTF-8, les nouveaux types `char16_t` et `char32_t` sont des unités de codage de seize ou trente-deux bits pour le codage des caractères Unicode via UTF-16 ou UTF-32.

Ces types sont standard à partir de C++2011 mais n'existent pas sur des compilateurs plus anciens, ni même sur les compilateurs C-2011.

Le type wchar_t

Ce type de caractère n'existe qu'avec les compilateurs supportant l'Unicode (jeu de caractère internationnal standard couvrant les langues du monde entier). Ces caractères sont stockés sur 2 octets ou 4. Les valeurs constantes de caractère (entre simple quote) ou de chaîne de caractères (entre double quote) doivent alors être précédées du caractère `L`.

Exemple :

```
wchar_t a = L'é'; // caractère 'é' unicode(16 ou 32 bits);
wchar_t[] chaine = L"Bonjour, monde !"; // chaîne de caractère unicode
```

Ce type présente le problème de ne pas être standard: certaines implémentations n'offrent que 16 bits soit une portion limitée des caractères Unicode.

API exceptionnellement Unicode

Certaines API C++ telle que Visual C++ sous Windows sont paramètrables par une option dite *unicode ou non*. Pour cela elles se basent sur le type de caractère`TCHAR` que le compilateur interprète (en fait remplace) par `char` ou `wchar_t` selon l'option Unicode. Dans ce cas, les valeurs constantes de chaînes et de caractères doivent être encadrées par la macro `_T`. Cette macro peut alors faire précéder les constantes d'un caractère `L` ou non.

Exemple :

```
TCHAR a = _T('é'); // caractère 'é' unicode ou huit bits;
TCHAR[] chaine = _T("Bonjour, monde !"); // chaîne de caractère unicode ou ascii (huit bits)
```

Les booléens

Le C++ utilise le type **bool** pour représenter une variable booléenne. Un bool ne peut prendre que 2 valeurs : `true` ou `false`. On peut affecter à un bool le résultat d'une condition, ou l'une des deux constantes citées précédemment.

Exemple :

```
bool a;
int c;
c = 89;
a = (c > 87);
// a reçoit alors la valeur true.
```

L'opérateur const

L'opérateur `const` placé devant une déclaration de variable décrit celle-ci comme constante; elle ne peut être changée.

Exemple :

```
int x = 2;
const int y = 2;
x += 1; // x recevra la valeur 3
y += 1; // Cette opération est interdite par un compilateur C++ conforme
```

Dans cet exemple, nous déclarons deux variables x et y, chacune avec la même valeur. La première variable (x) peut être modifiée, ce qui n'est pas le cas de la seconde (y) qui est déclarée const. Le compilateur refusera toute opération qui tenterait d'en modifier son contenu (opérations d'assignation et d'incrémentation =, +=, -=, *=, /=, >>=, <<=, ++).

Il en va de même pour les objets. Un objet déclaré const ne pourra pas être modifié, c'est-à-dire que le compilateur refusera l'invocation d'une méthode non const sur cet objet.

Exemple :

```
class X {
public:
    X() : valeur_(0) {}
    explicit X(int valeur) : valeur_(valeur) {}
    void annule() { this->valeur_ = 0; }
    void init(int valeur) { this->valeur_ = valeur; }
    int valeur() const { return this->valeur_; }
private:
    int valeur_;
};

X x(2)
const X y(5);
x.annule(); // Ok
y.annule(); // Erreur de compilation.
```

Dans cet exemple, deux objets x et y ont été déclarés. Il est possible d'invoquer n'importe quelle méthode (const ou non) sur l'objet x, par contre seules les méthodes const peuvent être invoquées sur l'objet y puisqu'il est déclaré const.

L'opérateur sizeof

L'opérateur `sizeof` permet de savoir le nombre d'octets qu'occupe en RAM une certaine variable. On peut écrire *sizeof(a)* pour savoir le nombre d'octets occupé par la variable a. On peut aussi écrire, *sizeof(int)* pour connaître le nombre d'octets occupés par une variable de type int. Cet opérateur est très utile lorsqu'on veut résoudre des problèmes de portabilité d'un programme.

Plus précisément `sizeof(char)` vaut toujours 1, par définition. Sur la plupart des architectures un `char` est codé par huit bits, soit un octet. Programmation C/Types de base#Caractères

Définir un alias de type

L'instruction `typedef` permet de définir un alias pour un type de données. Ceci permet dans certains cas de raccourcir le code, et dans tous les cas c'est l'occasion de donner un nom plus explicite à un type. Ceci favorise une meilleur lecture du code.

La syntaxe de `typedef` est exactement la même que celle de la déclaration d'une variable, excepté que l'instruction commence par `typedef` et qu'aucune variable n'est réservée en mémoire, mais un alias du type est créé.

Exemple:

```
typedef unsigned long data_size;

data_size readData(char* buffer, data_size buffer_size);
```

On s'aperçoit plus facilement que la fonction retourne le nombre d'octets lus que dans la déclaration sans `typedef` :

```
unsigned long readData(char* buffer, unsigned long buffer_size);
```

Voir aussi

- Exercices

- name=Un caractère doit juste être codé sur au moins 8 bits.
