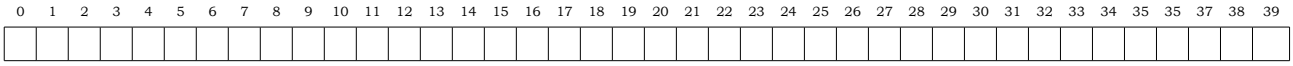
Centre Informatique pour les Lettres et les Sciences Humaines

Apprendre C++ avec Qt : Variables et constantes de types standards

1 - Notion de variable

Comme la seule réalité directement manipulable par un ordinateur est le contenu de sa mémoire, apprendre à programmer revient essentiellement à apprendre à contrôler ce contenu. Dans le contexte de ce cours, nous nous représenterons la mémoire de l'ordinateur comme une série de petites cases. Si on se limite aux quarante premières, on peut imaginer ceci :



Les numéros figurant en face des cases sont ce qu'on appelle leur adresse. Il est toujours possible de désigner une case mémoire par son adresse, mais ce n'est pas forcément très agréable pour le programmeur. Par ailleurs, ces cases étant en fait des dispositifs électriques, leur "contenu" n'est qu'une interprétation de leur état physique. Pour utiliser une information stockée en mémoire, il faut donc non seulement savoir où elle se trouve, mais aussi savoir quelles sont les conventions de codage qui doivent être utilisées pour interpréter l'état des cases concernées.

Le langage C++ propose de simplifier l'utilisation de la mémoire en attribuant un nom à certaines cases (ou à certains groupes de cases) et en convenant une fois pour toutes des conventions de codage qui s'appliqueront à la variable ainsi créé. Les différentes conventions de codage utilisées correspondent à ce qui, en C++, s'appelle les types.

2 - Création de variables

Du point de vue du programmeur, une variable sera donc caractérisée par son type, son nom et sa valeur courante (c'est à dire l'interprétation de l'état électrique actuel de la zone de mémoire concernée, selon les règles imposées par le type de la variable).

> Une fois que la variable existe, elle a évidemment une adresse en mémoire, mais cette adresse n'a normalement pas à être connue du programmeur, qui a justement créé la variable pour pouvoir ignorer ce genre de détails.

Choix d'un type et d'un nom

Lorsqu'un programmeur éprouve le besoin de stocker en mémoire une information particulière, il doit s'interroger sur le genre d'information dont il s'agit et sur les opérations qui doivent pouvoir être effectuées sur sa représentation, de façon à déterminer le type de la variable dont il a besoin. Ceci nécessite évidemment, de la part du programmeur, une certaine expérience et une bonne connaissance des types envisageables.

Il faut en outre choisir un nom. Les noms de variables sont soumis à certaines contraintes : ils ne doivent comporter ni espaces ni minuscules accentuées, mais peuvent utiliser le caractère de soulignement et les chiffres, à condition que ceux-ci n'apparaissent pas comme initiale. Le langage C++ distingue les minuscules des majuscules : `exemple` et `Exemple` seront considérées comme étant deux variables différentes. La plupart des programmeurs utilisent des minuscules pour les noms de variables, et l'interdiction des espaces oblige à séparer les mots autrement. On peut capitaliser lesInitialesDesMots ou séparer ceux-ci par des caractères_de_soulignement mais, en tous cas, la matérialisation de la limite des mots améliorenettementlalisibilitédutexte.

> Une autre contrainte pèse sur le choix des noms de variables : ceux qui correspondent à des éléments du langage ne peuvent pas être utilisés. C++ ayant une origine anglo-saxonne évidente, les conflits sont rares si vous donnez à vos variables des petits noms "bien de chez nous". La liste complète des mots du langage peut être consultée dans l'Annexe 1.

Les débutants ont généralement tendance à sous estimer gravement l'importance du choix des noms, et l'expérience prouve que le temps et les efforts consacrés à s'exprimer clairement lorsqu'on baptise les variables constituent toujours un investissement très rentable.

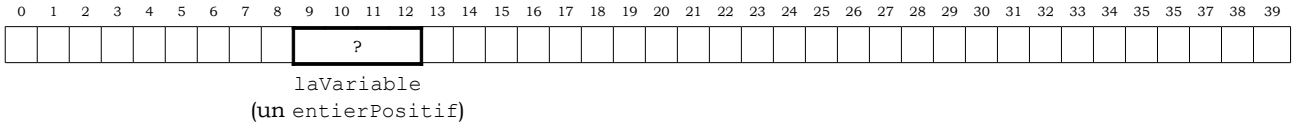
> N'hésitez pas à rebaptiser les variables dont le nom s'avère avoir été choisi maladroitement. Même si cette opération semble désagréable (elle provoque toujours une période de flottement, le temps que vous oubliiez le premier nom), rien n'est pire, en définitive, qu'une variable dont le nom suggère des idées fausses sur l'usage qui en est fait.

Définition

Une fois un type et un nom choisis, la définition d'une variable s'effectue simplement en énonçant ces deux informations. Ainsi, par exemple

```
entierPositif laVariable;
```

est la définition d'une variable de type `entierPositif` dont le nom est `laVariable`. Lorsqu'une instruction définissant une variable est exécutée, une zone de mémoire d'une taille adéquate est réservée et il devient possible d'agir sur cette zone en la désignant à l'aide du nom de la variable (de plus, les actions ainsi effectuées respecteront les règles imposées par le type choisi pour la variable). On peut imaginer ainsi les conséquences en mémoire de l'exécution de notre exemple :



La taille de la variable (4 octets) et sa position en mémoire (à l'adresse 9) ont été choisis arbitrairement pour les besoins de l'illustration. L'état électrique de cette zone est imprévisible et, s'il était interprété comme représentant une valeur de la variable, celle-ci serait sans signification.

La définition d'une variable est une instruction exécutable. Elle ne peut donc figurer que dans un bloc de code et, tant qu'elle n'a pas été exécutée, la variable correspondante n'existe pas.

Lorsque le point d'exécution du programme sort du bloc de code à l'intérieur duquel une variable est définie, la variable en question est détruite.

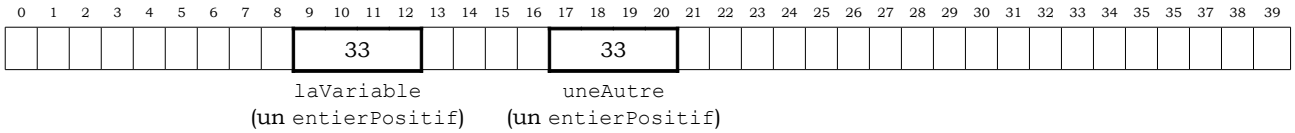
Concrètement, lorsqu'une variable est détruite, son nom ainsi que la valeur qu'elle contenait cessent d'être utilisables.

Initialisation

La définition d'une variable peut s'accompagner d'une initialisation qui, comme son nom l'indique, fixe la valeur que prendra la variable lors de sa création. Il suffit pour cela, lors de la définition d'une variable, de faire suivre son nom par des parenthèses entourant la valeur choisie.

```
1 entierPositif laVariable(33);
2 entierPositif uneAutre(laVariable); //ok : la valeur de laVariable est connue
```

Après exécution des deux lignes de code précédentes, on peut imaginer ainsi l'état de la mémoire :



L'initialisation peut également être obtenue en utilisant le signe = :

```
entierPositif laVariable = 33;
```

La valeur utilisée pour initialiser une variable doit être compatible avec le type de la variable.

```
entierPositif laVariable(-3.14);  //une plaisanterie de mauvais goût ?
```

Constance

On peut interdire toute modification de la valeur d'une variable en faisant précéder sa définition du mot `const` :

```
const entierPositif AGE_RETRAITE = 65;  //on peut rêver...
```

On utilise traditionnellement pour les constantes des noms composés de lettres majuscules.

Il faut obligatoirement initialiser les constantes.

3 - Les types standard

Il n'existe que trois types fondamentaux en C++ : deux types numériques (les types entier et décimal) et un type logique (le type booléen). Les types numériques connaissent toutefois quelques variantes, ce qui se traduit finalement par un vocabulaire assez copieux.

Le type booléen

Une variable de type `bool` peut contenir l'une des deux valeurs logiques `true` et `false`.

```
//quelques exemples de définitions de booléens
bool uneVariable;
bool uneAutre(true);
const bool VRAI(true);
const bool FAUX = false;
```

Les types décimaux

Le type le plus "naturel" pour une variable destinée à stocker des nombres décimaux est `double`. C'est celui utilisé par la plupart des fonctions mathématiques, et celui attribué aux constantes littérales décimales (si vous écrivez 3.14, le compilateur suppose qu'il s'agit d'une valeur de type `double`). Il existe une variante qui peut occuper moins de place (`float`) et une variante qui peut offrir une plage de valeurs plus étendue et permettre une plus grande précision (`long double`).

```
//quelques exemples de définition de décimaux
const double PI(3.14);
float unNombre;
long double unAutreNombre = 2;
```

Les types entiers "ordinaires"

Le type le plus "naturel" pour une variable destinée à stocker des nombres entiers est `int`. C'est, en effet, celui attribué aux constantes littérales entières (si vous écrivez 1789 dans un programme, le compilateur suppose qu'il s'agit d'une valeur de type `int`). Le type `int` possède une variante qui peut occuper moins de place en mémoire (`short`) et une variante qui peut offrir une plage de valeurs possibles plus étendue (`long`).

Ces trois types entiers peuvent être modifiés par le mot `unsigned`. Ce mot indique que le nombre doit être considéré comme dépourvu de signe (c'est à dire comme étant toujours positif), ce qui double la valeur maximale représentable par la variable.

```
//quelques exemples de définition de variables de type entier
short unEntier;
int unAutreEntier(-36);
long unTroisiemeEntier;
unsigned long unTresGrandEntierPositif = 17;
```

Un type entier "spécial" : `char`

Il s'agit, en fait, d'un type entier assez "ordinaire", à deux détails près :

1) Lorsque la valeur d'une variable de type `char` doit être présentée à l'utilisateur (dans le cas d'un affichage à l'écran, par exemple), les conventions habituelles de représentation des nombres entiers sont abandonnées au profit d'une table de correspondance arbitraire (c'est souvent le code ASCII qui est utilisé). Ainsi, si une variable de type `char` contient la valeur 65, son affichage sur l'écran d'un système ASCII ne se traduira pas par les deux caractères 6 et 5, mais par un seul : A. Du fait de cette particularité, le type `char` sert souvent de base à la représentation des textes, mais il ne faut jamais oublier qu'il s'agit d'un type entier assez normal par ailleurs.

2) Le langage C++ ne précise pas si le type `char` est ou non signé, ce qui laisse aux différents compilateurs toute latitude pour adopter l'une ou l'autre option. En cas de besoin, on peut lui appliquer les modificateurs "`signed`" ou "`unsigned`", de façon à lever l'ambiguïté.

Les constantes littérales exprimées par un caractère unique, placé entre apostrophes, sont des valeurs de type `char` (c'est à dire des nombres).

```
//quelques exemples de définitions de variables de type char
1 char uneVariable;             //cette variable n'est pas initialisée
2 unsigned char uneAutre(17);   //ok, c'est une variable de type numérique
3 char encoreUne = 'A';         //ok, puisque 'A' est un nombre
```

Conversions automatiques

Le langage assure un certain nombre de "transtypages", c'est à dire que, parfois, la représentation d'une valeur dans un certain type est transformée en représentation de la même valeur dans un autre type. Ainsi, lorsque nous écrivons

```
int unEntier('a');
```

la valeur `'a'`, qui est de type `char`, doit être représentée dans le format `int` avant de pouvoir être stockée dans la variable `unEntier`.

> Bien qu'il s'agisse de deux types entiers, les types `char` et `int` sont différents (ne serait-ce que par le nombre de cases mémoires attribuées à chaque variable) et l'état électrique correspondant à une valeur dans l'un des types est différent de celui représentant la même valeur dans l'autre type.

Les règles exactes qui gouvernent ces conversions sont trop complexes pour être présentées ici, mais, dans la plupart des cas, le bon sens permet de deviner ce qui va se passer : la conversion d'une valeur dans un type "plus puissant" que son type initial ne pose pas de problème, alors qu'une conversion inverse risque de se traduire par une perte d'information.

> On peut dire qu'un type est plus puissant qu'un autre s'il permet de représenter de façons distinctes toutes les valeurs possibles pour une variable de cet autre type.

Nous utiliserons donc sans arrière-pensées les conversions de valeurs de type `char` vers les autres types numériques ordinaires, ou de valeurs de types entiers vers le type `double`.

```
1 double unDouble = 5; //l'int 5 est automatiquement converti en double
2 int unEntier = 8;
3 double unAutre = unEntier; //l'int 8 est automatiquement converti en double
```

> La conversion ne concerne évidemment que la valeur utilisée au cours de l'exécution de l'instruction (en C++, les variables ne changent jamais de type). Ainsi, après exécution de la ligne *3*, `unEntier` est toujours de type `int` et contient toujours la valeur 8.

Il existe en outre une conversion automatique dont il est important d'avoir connaissance :

Toute valeur numérique non nulle sera, en cas de besoin, convertie en valeur booléenne `true`, alors qu'une valeur numérique nulle sera convertie en valeur booléenne `false`. Réciproquement, les valeurs `true` et `false` peuvent être converties respectivement en 1 et 0.

Conversions explicites

Il arrive que (pour les besoins d'un calcul, par exemple), on souhaite disposer d'une représentation d'une valeur dans un type particulier, alors que la valeur en question est stockée dans une variable d'un autre type. Le langage C++ permet d'exprimer clairement ce désir : si `unEntier` est une variable de type `int`, l'expression

```
static_cast <double> (unEntier)
```

a la même valeur que `unEntier`, mais est de type `double`.

> Pour des raisons historiques (compatibilité avec des versions antérieures du langage), il est également possible d'obtenir le même effet avec les deux notations suivantes :
>
> ```
> double(unEntier)
> (double) unEntier
> ```