Centre Informatique pour les Lettres et les Sciences Humaines

C++ : Leçon 2
Variables

La Leçon 1 s'est achevée sur l'idée que les programmes C++ étaient segmentés en fonctions et manipulaient généralement la mémoire sous la forme de variables. Il nous faut maintenant examiner comment ces variables sont créées. La seule véritable difficulté de cette Leçon est que les différents types de variables ne prennent vraiment de sens que lorsqu'on les utilise. Mais, d'un autre côté, pour pouvoir utiliser des variables, il faut d'abord les créer... Patience, donc. Les choses commencent à devenir un peu moins frustrantes à partir de la Leçon 3.

1 - Définition de variables

Lorsqu'un programmeur éprouve le besoin de stocker en mémoire une information particulière, il doit examiner trois types de questions.

Une première interrogation concerne la façon dont va être codée cette information. En réfléchissant au genre d'information dont il s'agit et aux opérations qui doivent pouvoir être effectuées sur sa représentation, le programmeur parvient à une conclusion sur le type de variable dont il a besoin. Ceci nécessite évidemment, de la part du programmeur, une certaine expérience et une bonne connaissance des types envisageables.

La deuxième question à résoudre est le choix d'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 premier caractère du nom. Le langage C++ distingue les minuscules des majuscules, ce qui signifie que `exemple` et `EXEMPLE` seront considérés comme étant les noms de deux variables différentes. La plupart des programmeurs ont l'habitude d'utiliser pour les variables des noms écrits en minuscules. L'interdiction d'utiliser des espaces conduit à adopter un autre moyen pour séparer les mots. Il est possible de capitaliserLesInitialesDesMots ou de séparer ceux-ci par des_caractères_de_soulignement. Dans un cas comme dans l'autre, la matérialisation de la limite des mots améliorenettementlalisibilitédutexte.

> Un autre type de contrainte pèse sur le choix des noms : ceux qui correspondent à des éléments prédéfinis du langage ne peuvent être utilisés pour définir des variables. Les éléments du langage ayant une origine anglo-saxonne évidente, les conflits sont rares si vous donnez à vos variables des petits noms "bien de chez nous". Notez toutefois que vous ne pourrez pas gérer votre basse-cour à l'aide d'une variable baptisée `volatile`, ni même créer un programme de jeux comportant une variable nommée `case`. La liste complète des mots du langage peut être consultée dans l'Annexe 1 : Petit inventaire lexical.

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.

Le troisième type de décisions que le programmeur doit prendre concerne le statut de la variable et fait intervenir des considérations sur la "durée de vie" de celle-ci, sa disponibilité pour les différentes fonctions composant le programme ou la prévisibilité de la quantité d'information à stocker. Cette décision se traduit tout d'abord par le choix de la position dans le programme à laquelle l'instruction créant la variable doit être insérée, et nous pouvons donc laisser cette question en suspens pour l'instant.

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 maVariable; //exemple de définition de variable
```

est la définition d'une variable de type "`entierPositif`" dont le nom est "`maVariable`".

Lorsqu'un programme comporte une telle définition, le compilateur génère le code convenable pour réserver le nombre de cases mémoire nécessaires et fait en sorte que, dorénavant, toute instruction concernant `maVariable` se traduise par du code affectant ces cases mémoire.

Pour pouvoir remplir ce contrat, le compilateur doit donc disposer de certaines informations concernant le type concerné : Quelle place occupe-t-il en mémoire ? Comment doit-on modifier la mémoire lorsqu'un ordre concernant `maVariable` doit être exécuté ? Deux cas peuvent alors se présenter : soit il s'agit de l'un des types standard[1] du langage (et, dans ce cas, le compilateur dispose, par construction, de toute l'information nécessaire), soit il s'agit d'un type défini par l'utilisateur (et, dans ce cas, la définition en question doit être disponible, faute de quoi il est impossible de compiler le programme).

2 - Initialisations

La définition d'une variable peut s'accompagner d'une initialisation qui, comme son nom l'indique, indique la valeur que doit prendre la variable dès sa création. Il suffit pour cela, lors de la définition d'une variable, de faire suivre son nom par le signe = et la valeur choisie.

```
//exemples de définitions de variables comportant une initialisation
1 entierPositif maVariable = 33;
2 entierPositif uneAutre = maVariable;//ok: maVariable a déjà une valeur connue
```

> Les numéros verts figurant dans la marge en face des fragments de code comportant plusieurs lignes ne sont qu'un artifice facilitant parfois la discussion et n'ont aucun équivalent dans un texte source réel.

[1] On les dit aussi parfois "natifs", "fondamentaux", "prédéfinis" ou "de base".

L'initialisation peut également être obtenue en plaçant la valeur choisie entre parenthèses :

```
//autre exemple de définition de variable comportant une initialisation
entierPositif maVariable (33);
```

> Lorsqu'une variable est définie sans être initialisée, on emploie souvent le mot "initialisation" pour faire allusion à la première opération ayant pour effet d'attribuer une valeur déterminée à cette variable. Si l'usage du terme semble justifié par le changement d'état de la variable concernée, il faut tout de même signaler que, techniquement, il ne s'agit pas dans ce cas d'une initialisation. La nuance a une importance réelle, car, nous le verrons, certains effets ne peuvent être obtenus que dans le cadre d'une authentique initialisation (c'est à dire au moment de la définition).

Il va sans dire que la valeur utilisée pour initialiser une variable doit être compatible avec le type de la variable.

```
entierPositif maVariable = - 3.14;  //une plaisanterie de mauvais goût ?
```

3 - Les types standard

A bien y regarder, il n'existe en fait que trois types fondamentaux en C++ : deux types numériques (le type entier et le type décimal) et un type logique (le type booléen). Les types numériques donnent toutefois lieu à quelques variations (avec, parfois, des synonymies), ce qui se traduit finalement par un vocabulaire que l'on peut sans doute juger disproportionné.

Le type booléen

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

```
//quelques exemples de définitions de variables booléennes
1 bool uneVariable;
2 bool uneAutre = true;
3 bool encoreUne(false);
```

Les types décimaux

Le type le plus "naturel" pour une variable destinée à stocker des nombres décimaux est `double`. C'est en effet celui utilisé par la plupart des fonctions mathématiques, et en tous cas, celui attribué aux constantes littérales décimales (en d'autres termes, si vous écrivez `3.14` dans un programme, le compilateur suppose qu'il s'agit d'une valeur de type `double`).

Il existe une variante qui peut occuper moins de place en mémoire (`float`) et une variante qui peut permettre une plus grande précision (`long double`), mais la taille exacte des types `float`, `double` et `long double` n'est pas définie par le langage : elle dépend du compilateur.

> C++ a hérité du langage C cette façon très précise de... ne pas préciser certains de ses aspects ! Il s'agit, en l'occurrence, de laisser les concepteurs de compilateurs tirer le parti maximum des performances du processeur visé. Dans le cas de Visual C++, le type `float` est représenté sur 4 octets et peut représenter des quantités dont la valeur absolue est comprises entre 1.175494351 E – 38 et 3.402823466 E + 38, avec une précision d'au moins 6 chiffres. Les types `double` et `long double` sont identiques et occupent 8 octets. Ils peuvent représenter, avec une précision d'au moins 15 chiffres, des quantités dont la valeur absolue est comprises entre 2.2250738585072014 E – 308 et 1.7976931348623158 E + 308.

```
//quelques exemples de définition de variables de type décimal
1 float unNombre = 3.14;
2 double unAutreNombre;
3 long double unTroisieme(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 (en l'absence de spécification contraire), aux constantes littérales entières (en d'autres termes, si vous écrivez `1789` dans un programme, le compilateur suppose qu'il s'agit d'une valeur de type `int`).

Il existe une variante qui peut occuper moins de place en mémoire (`short`) et une variante qui peut offrir une plage de valeurs possibles plus étendues (`long`), mais les caractéristiques exactes de `short`, `int` et `long` ne sont pas parfaitement définies par le langage : elles dépendent en partie du compilateur. (La norme exige d'une part que le type `short` corresponde à au moins 2 octets et le type `long` à au moins 4, et, d'autre part, que l'ordre intuitif des tailles soit respecté : `short` ≤ `int` ≤ `long`.)

> Dans le cas de Visual C++, le type `short` est sur 2 octets et peut donc représenter des valeurs allant de –32 768 à 32 767, alors que les types `int` et `long` (qui sont identiques) occupent 4 octets, ce qui leur permet de stocker des valeurs allant de –2 147 483 648 à 2 147 483 647.

```
//quelques exemples de définition de variables de type entier
short unEntier;
int unAutreEntier = -36;
long unTroisiemeEntier;
```

Ces trois types entiers peuvent être modifiés par le mot `unsigned`. Ce mot signifie simplement que le nombre doit être considéré comme dépourvu de signe (c'est à dire, en fait, comme étant toujours positif), ce qui a pour effet de doubler la quantité maximale représentable (puisque, comme nous l'avons vu dans la Leçon 1, les valeurs négatives occupent la moitié des patterns disponibles et réduisent d'autant le nombre de valeurs positives représentables).

```
//quelques exemples de définition de variables de type entier non signé
unsigned short unEntierPositif(17);
unsigned int unAutreEntierPositif;
unsigned long encoreUn;
```

Un type entier "spécial" : `char`

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

1) Lorsque le contenu d'une variable de type `char` doit être présenté 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 en général le code ASCII (ou une de ses variantes adaptées aux langues européennes) qui est utilisé, mais ce n'est pas nécessairement le cas.

   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 objets de type `char` (c'est à dire des nombres).

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

D'autres types entiers "spéciaux" : les pointeurs

Les pointeurs sont des objets destinés à contenir des adresses.

> Nous avons vu, au cours de la première Leçon, qu'une adresse est simplement un numéro affecté à chacune des cases de mémoire et servant à la désigner. Le fait que le langage C++ permet de désigner certaines cases mémoires à l'aide du nom des variables auxquelles elles correspondent ne signifie pas que nous devions renoncer à accéder à la mémoire en utilisant les adresses. Après tout, c'est finalement de cette façon que les choses se dérouleront (c'est la seule méthode possible pour le processeur), et il y a des cas où utiliser explicitement les adresses s'avère plus simple que d'essayer de les masquer derrière des noms de variables.

Puisque ces fameuses adresses ne sont jamais que de bêtes numéros, pourquoi ne pas les stocker dans des variables numériques "ordinaires", de type `int`, par exemple ? C'est tout à fait possible, et nous serons effectivement amenés à le faire dans certains cas. En général, toutefois, cette façon de procéder n'est pas vraiment satisfaisante, car elle sacrifie inutilement les avantages que nous avons obtenus en introduisant la notion de type.

La notion de type, rappelons-le, permet au compilateur d'une part de vérifier la légitimité des opérations que nous entreprenons, et, d'autre part, de nous décharger du détail de la mise en œuvre des opérations en question. Souvenez-vous : nous n'avons aucune envie d'avoir à préciser (et donc à savoir…) ce qu'il faut faire exactement pour modifier la représentation d'un nombre décimal de façon à ce qu'elle devienne la représentation de ce nombre décimal augmenté d'une unité.

> Si vous ne vous souvenez pas, relancez le programme Visuram, choisissez une zone de quatre octets, notez sa valeur en tant que nombre décimal et essayez de trouver l'état de la mémoire qui correspond à cette valeur plus un. Bonne chance, et à l'année prochaine…

Nous sommes donc très contents que le compilateur se charge de traduire notre ordre "ajoute 1" en tenant compte de la nature exacte de ce à quoi nous voulons "ajouter 1". Si une adresse est simplement stockée dans une variable de type `int`, le compilateur n'a plus aucune trace de la nature de l'objet qui est représenté dans la zone de mémoire située à cette adresse. Il ne peut donc plus nous aider à travailler correctement sur l'information qu'elle contient, et, de fait, nous ne sommes guère plus avancés que si nous n'avions pas de compilateur et étions obligés de programmer directement en langage machine.

Fort heureusement, C++ nous offre un moyen de manipuler les adresses sans que pour autant le compilateur perdre la trace du type des objets qui sont représentés dans les zones mémoires désignées : il s'agit précisément des pointeurs.
La définition d'une variable de type "pointeur sur…" n'est guère différente de la définition d'une variable d'un autre type. Il faut simplement comprendre qu'il n'existe pas UN type "pointeur", mais un nombre potentiellement infini de types "pointeur sur…". Il n'est donc pas possible de leur donner à chacun un nom différent, et la syntaxe utilisée consiste simplement à faire suivre d'une étoile le type des objets dont le pointeur est destiné à stocker l'adresse. Le nom de la variable définie doit, comme d'habitude, suivre l'énoncé de son type. Ainsi,

```
double * unPointeur;
```

ne définit pas une variable de type `double` (qui serait capable de stocker une quantité décimale), mais une variable de type "pointeur sur `double`" (c'est à dire une sorte d'entier).

Outre la possibilité qu'ils offrent de conserver le type de l'objet désigné, les pointeurs présentent un autre avantage par rapport au stockage des adresses dans des variables entières ordinaires. En effet, étant donné que les pointeurs ne sont destinés qu'à contenir des adresses, le compilateur est en mesure de s'opposer à certaines opérations qui ne peuvent être légitimement effectuées sur une adresse. Si, par exemple, vous essayez de multiplier une adresse par 2, c'est certainement par erreur car vous ne pouvez en aucun cas avoir la moindre idée de ce qui se trouve à l'adresse ainsi obtenue, et vous courrez au désastre si vous essayez de l'utiliser de quelque façon que ce soit. Si l'adresse est contenue dans une variable entière ordinaire, le compilateur ne "sait" pas qu'il s'agit d'une adresse, et n'a donc aucune raison de s'opposer à la multiplication. Le résultat que vous obtenez est une adresse dangereuse, et la catastrophe n'est certainement pas loin. Si l'adresse est contenue dans un pointeur, en revanche, le compilateur refusera de traduire en langage machine votre demande de multiplication par 2 et vous préviendra (plus ou moins aimablement) de votre inconséquence.

> Arrivé à ce point, une question vous brûle sans doute les lèvres : Quel intérêt ? Est-il bien raisonnable de s'encombrer de ces mystérieux pointeurs, alors qu'on peut se contenter d'appeler les choses (et les variables en particulier) par leur nom ?
>
> Sans trop anticiper sur la suite du cours, on peut déjà souligner une différence essentielle entre l'accès direct à une variable (à l'aide de son nom) et l'accès indirect offert par un pointeur. Imaginez qu'il y ait un grand nombre de données de type `double`, et qu'il s'avère soudain nécessaire d'ajouter 1 à 9 697 d'entre elles. Si vous ne pouvez accéder à une de ces données qu'en mentionnant le nom de la variable qui la stocke, il ne vous reste plus qu'a recopier 9 697 fois l'ordre "ajoute 1" dans votre programme, en l'appliquant chaque fois à une variable différente. Un nom, en effet, désigne une variable et une seule, et toujours la même. Un pointeur, en revanche, est lui-même une variable, ce qui signifie que son contenu peut changer. Que se passe-t-il lorsque l'adresse contenue dans un pointeur change ? La zone de

mémoire qui se trouve désignée par le pointeur n'est plus la même. Ainsi, le même ordre "ajoute 1", s'il est exécuté sur une zone de mémoire désignée par un pointeur dont on fait varier le contenu, pourra servir 9 697 fois (grâce à l'un des dispositifs de répétition que nous étudierons dans la Leçon 4), ce qui est tout de même plus élégant (et moins fatigant) que des dizaines de pages de code répétant la même instruction ! Plus sérieusement, si le nombre de données à traiter est inconnu au moment de l'écriture du programme (parce qu'il dépend des circonstances de l'exécution), la technique utilisant le nom des variables est tout simplement inapplicable, même si les données en question s'avèrent finalement peu nombreuses.

Maintenant, si vous avez déjà un peu pratiqué la programmation à l'aide d'un autre langage, vous pensez sans doute que le cas des 9 697 données de type `double` peut être traité très facilement, à l'aide d'un tableau que l'on parcourt en faisant varier un index. Deux remarques s'imposent alors. La première est que le tableau n'est qu'un cas particulier : il arrive que l'organisation "naturelle" des données à traiter ne soit, justement, pas un tableau. Il peut s'agir d'une liste chaînée (Leçon 10) ou d'un arbre (Leçon 19), par exemple. L'usage de pointeurs offre une réponse générale à ce problème, alors qu'un index ne permet de traiter qu'une seule structure de données, le tableau. La deuxième remarque est que, lorsqu'une organisation de type tableau convient, l'usage d'un pointeur ne donne pas lieu à une écriture très différente de celle adoptée lorsqu'on utilise un tableau (nous verrons dès la Leçon 9 que, lorsqu'on lit un programme, l'examen des instructions manipulant les données ne permet pas de déterminer s'il y a un véritable tableau ou s'il s'agit d'un simple pointeur).

Conversions automatiques

Le langage assure, sans formalités particulières, un certain nombre de "transtypages", c'est à dire que, dans certains cas, 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`.

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[2] que le sien ne pose pas de problème, alors qu'une conversion inverse risque de se traduire par une perte d'information.

Nous utiliserons donc sans arrière pensées les conversions de valeurs de type `char` vers les autres types numériques ordinaires, ainsi que les conversions de valeurs de types entiers vers les types décimaux. Les conversions entre types de pointeurs, ainsi que celles aboutissant à un type qui n'est pas capable de représenter toutes les valeurs possibles pour le type initial, exigent en revanche des mesures spéciales, sur lesquelles nous aurons l'occasion de revenir.

```
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
```

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`. A l'inverse, les booléens `true` et `false` peuvent être convertis respectivement en 1 et en 0.

Le type `bool` a été introduit assez récemment dans le langage C++, et vous aurez sans doute l'occasion d'être confronté à des pratiques établies antérieurement. Le principe fondamental en est l'utilisation de la valeur 0 pour tenir lieu de valeur logique fausse. Toute valeur non nulle est, dans ce contexte, équivalente à la valeur logique vraie. Pour rendre les choses un peu plus claires, de nombreux programmeurs avaient pris l'habitude de définir et d'utiliser des constantes entières nommées `TRUE` et `FALSE`. La seule différence entre `TRUE` et `true`, d'une part, et `FALSE` et `false`, d'autre part, est donc que les uns sont des entiers alors que les autres sont des `bool`, ce qui n'a généralement pas de conséquence pratique majeure : les conversions automatiques décrites ci-dessus permettent à des portions de code écrites avant l'introduction du type `bool` de fonctionner harmonieusement avec du code plus récent.

[2] Informellement, on peut dire qu'un type est "plus puissant" qu'un autre s'il permet de représenter toutes les valeurs possibles pour une variable de cet autre type.

4 - Définition de nouveaux types

La plupart des programmes sont amenés à manipuler des réalités plus complexes que de simples valeurs logiques ou numériques. Une des caractéristiques du langage C++ est qu'il encourage le programmeur à traiter tout problème comme un problème de définition de types de données. Les moyens disponibles pour créer de nouveaux types constituent donc le véritable cœur du langage, reléguant au rang de détails subalternes des questions telles que les différentes façons de répéter l'exécution d'un groupe d'instructions, ou de décider s'il faut ou non exécuter telle ou telle instruction. Inutile, donc, de préciser que la création de nouveaux types est un sujet qu'il n'est pas question d'épuiser dès la Leçon 2 !

Les types énumérés

Une première façon, très simple et pourtant fort utile, de créer un nouveau type de données est l'énumération. Cette technique convient dans les cas où l'information qui doit être stockée dans une variable correspond à un choix dans une liste assez brève.

Un doigt d'une main humaine normale, par exemple, est nécessairement un pouce, un index, un majeur, un annulaire ou un auriculaire. Si vous écrivez un programme destiné à votre manucure, il est possible que le type DOIGT présente un intérêt. Ce qu'il vous faut, en fait, c'est un type permettant de créer des variables pouvant prendre les cinq valeurs envisageables, et uniquement celles-là. Le langage C++ vous permet de créer ce type de la façon suivante :

```
//exemple de définition d'un nouveau type de variables par énumération
enum DOIGT {POUCE, INDEX, MAJEUR, ANNULAIRE, AURICULAIRE };
```

Il faut bien comprendre que la ligne de code ci-dessus *ne définit aucune variable* (les noms correspondant aux différentes valeurs possibles ne sont, bien entendu, pas des noms de variables, ce qu'on peut souligner en les écrivant en majuscules). Aucune zone mémoire n'est réservée pour permettre à votre programme de stocker quelque information que ce soit. En fait, lorsque le compilateur rencontre la ligne de code en question, il ne génère aucune instruction destinée au processeur. Il se contente de "prendre note" de ce que vous entendez, à partir de cet instant, par un `DOIGT`. Le type `DOIGT` devient alors disponible, et vous pouvez l'utiliser pour définir des variables, exactement comme vous utilisez les types standard :

```
1 DOIGT unDoigt;                //Définition d'une variable de type DOIGT
2 DOIGT unAutreDoigt = MAJEUR;  //Définition avec initialisation
3 DOIGT encoreUn(POUCE);        //Définition avec initialisation
```

Toute tentative de donner à une variable de type `DOIGT` une valeur ne faisant pas partie de l'énumération définissant le type sera impitoyablement rejetée par le compilateur.

```
  //Exemples d'erreurs d'utilisation du type DOIGT
1 DOIGT unDoigt = 4;                //interdit en C++
2 DOIGT unAutreDoigt = Majeur;      //impossible (à cause des minuscules)
```

Les seules choses qu'on puisse faire avec une variable d'un type énuméré sont lui attribuer l'une des valeurs figurant dans l'énumération et vérifier si sa valeur est ou non égale à une autre valeur. Paradoxalement, c'est cette restriction extrême des opérations possibles qui rend les types énumérés intéressants : si une variable de type `DOIGT` a été initialisée, il est certain que son contenu sera toujours l'une des cinq valeurs possibles.

> Si vous êtes en train de découvrir le processus de création de types, cette présentation des types énumérés devrait suffire. Si vous êtes en train de relire cette Leçon parce que vous avez un problème particulier à ce propos, voyez Horton, page 54.

Les classes

Si les types énumérés sont très faciles à créer et s'avèrent souvent très pratiques, il faut reconnaître que leur usage ne convient qu'à certains cas assez particuliers. Le langage C++ offre d'autres moyens, qui permettent de donner naissance à des types dont l'usage est moins restreint. Le principal de ces moyens est la création de classes. Bien que le système des classes soit considérablement plus complexe que celui des types énumérés, il partage avec celui-ci un point essentiel : il s'agit d'abord de définir un type, et c'est seulement dans un second temps que des variables de ce type peuvent effectivement être créées.

Les classes sont une variété particulière de types. Lorsque le type d'une variable est une classe, on a l'habitude de dire que la variable est une instance de cette classe.

Lorsqu'une variable est une instance d'une classe, on a aussi l'habitude de dire que c'est un objet. De cette habitude sont nées les expressions "langage à objets" et "langage orienté objets", que vous avez peut-être déjà rencontrées.

> L'orientation objet a connu, il y a quelques années, une vogue un peu ridicule (qui s'est maintenant un peu calmée). On a ainsi pu parler de "bases de données orientées objets", de "systèmes d'exploitation orientés objets", et même de "romans policiers orientés objets". Ne me demandez pas de quoi il s'agit, je ne m'intéresse plus qu'aux cyber e-médi@s virtuels...

Il faut toutefois savoir que l'usage du mot "objet" en programmation a largement précédé l'apparition des "langages à objets", et qu'il est toujours légitime de parler d'objet à propos d'une entité quelconque manipulée par un programme. Il est parfaitement correct, par exemple, de dire qu'une simple variable de type `int` est un objet, bien que le type `int` ne soit pas une classe. Mieux encore, on utilise parfois le mot objet en pensant à des choses qui ne sont pas nécessairement des variables, comme par exemple des fonctions, ou même des types (y compris les classes elles-mêmes !). N'investissez donc pas le mot objet d'une signification technique trop stricte.

Une des premières choses que l'on peut dire pour expliquer ce qu'est une classe est que ce système est bien adapté à une description multidimensionnelle de la réalité. En d'autres termes, si plusieurs variables peuvent "collaborer" pour décrire ensemble le phénomène qui vous intéresse, la définition d'une classe est sans doute une méthode que vous devriez envisager. Nous avons déjà vu un exemple de ce genre : dans la Leçon 1, nous avons évoqué le problème de la représentation des couleurs, et nous avons vu qu'une solution possible était de les décrire selon trois dimensions (rouge, vert et bleu).

Dans sa version la plus simple, une classe permettant de définir des variables représentant des couleurs pourrait elle-même être définie ainsi :

```
//exemple de définition d'une classe très rudimentaire
1 class ClasseCouleur
2 {
3 public:
4    int quantiteDeRouge;
5    int quantiteDeVert;
6    int quantiteDeBleu;
7 };
```

Si on analyse cette définition, on peut y reconnaître certains éléments familiers.

Tout d'abord, tout comme notre exemple de type énuméré, la classe définie ici porte un nom, qui sera utilisé pour définir les variables de ce type. En l'occurrence, il ne s'agit plus de `DOIGT`, mais de `ClasseCouleur`.

Ensuite, il semble assez clair que le type `ClasseCouleur` implique en fait plusieurs variables, de type `int`, qui sont manifestement chargées de stocker respectivement les quantités de rouge, de vert et de bleu.

Cette apparence familière ne doit cependant pas vous abuser : les lignes *4* à *6* de la définition précédente ne créent aucune variable. En effet, nous sommes ici en train de définir un *type*. Le compilateur ne va pas générer des instructions destinées au processeur (et permettant, par exemple, de réserver des zones de mémoire pour y stocker des données), mais il va simplement "prendre note" de ce que nous entendons, à partir de maintenant, par `ClasseCouleur`. C'est

seulement lorsque nous définissons une *variable* de ce type que le compilateur génère du code dont l'exécution se traduira effectivement par une réservation de mémoire.

```
ClasseCouleur uneCouleur;      //création d'une instance de la ClasseCouleur
```

Comment se présente alors la variable `uneCouleur` ? Elle a trois "sous-variables", et on désigne chacune d'entre elles à l'aide d'un nom formé en combinant ceux de la variable et de la "sous-variable" considérée. En l'occurrence, après la définition de la variable `uneCouleur`, nous disposons en fait de trois variables nommées respectivement `uneCouleur.quantiteDeRouge`, `uneCouleur.quantiteDeVert` et `uneCouleur.quantiteDeBleu`.

S'il ne s'agissait que de cela, le concept de classe serait déjà d'une certaine utilité. Il correspondrait, grosso-modo, à ce que le langage C appelle une *structure*[3], le langage Pascal un *record* et le langage Visual Basic un *type membre*, et il permettrait une certaine organisation des données. Une des caractéristiques majeures des types prédéfinis lui manquerait cependant. En effet, un type prédéfini ne permet pas simplement au compilateur de savoir comment coder l'information, il lui permet aussi de savoir comment opérer sur cette information en respectant la signification qu'elle a.

> Dire, par exemple, qu'une variable est de type `double` n'indique pas seulement qu'il faut lui réserver huit cases de mémoire et que l'information y est codée selon les conventions en vigueur pour la représentation des nombres décimaux. Le type `double` implique aussi que l'ordre "ajouter 1" doit se traduire par une manipulation de la mémoire particulière, propre à ce type (totalement différente de ce qui se passe, par exemple, lorsqu'on "ajoute 1" à un `int`).

Telle que nous l'avons définie ci-dessus, la classe `ClasseCouleur` indique effectivement combien d'octets doivent être réservés pour une variable de ce type, et comment l'information y est codée.

> En l'occurrence, la place nécessaire est de trois fois les quatre octets nécessaires pour chacune des "sous-variables", soit au total 12 octets. L'ordre dans lequel elles sont mentionnées dans la définition de la classe et le fait que ces sous-variables sont de type `int` suffisent à définir parfaitement le codage utilisé.

En revanche, rien n'indique au compilateur comment il doit opérer sur une variable de type `ClasseCouleur`. Dans son état présent, le type `ClasseCouleur` ne peut servir qu'à définir des variables, variables sur lesquelles le programme ne pourra pratiquement effectuer de traitements qu'en accédant aux sous-variables.

> Remarquez qu'accéder aux sous-variables ne permet d'effectuer des traitements que dans la mesure où les sous-variables sont d'un type prédéfini, ce qui permet au compilateur de savoir comment traduire les ordres les concernant. Si les sous-variables sont elles-mêmes des instances d'une classe, le programme doit accéder aux sous-variables des sous-variables pour pouvoir effectuer des traitements (à condition que ces sous-sous-variables ne soient pas, elles-mêmes...). Inutile de préciser que ce genre de système atteint rapidement les limites au-delà desquelles il perd tout intérêt pratique.

Pour que le compilateur accepte de traiter des ordres appliqués directement à des variables de type `ClasseCouleur`, il faut que la définition de la classe comporte une description de la façon dont ces ordres doivent être exécutés. Nous sommes donc conduits à conclure que la classe doit comporter du code source décrivant les traitements qui peuvent être appliqués aux données qu'elle représente. Comme nous savons que C++ organise le code en segments nommés des fonctions, il est assez naturel d'envisager la présence de fonctions dans la définition de la classe.

Avant de passer à la Leçon 3, qui concerne évidemment la définition de fonctions, il nous faut adopter une dernière convention terminologique.

Les variables et fonctions qui constituent une classe sont appelés des membres de cette classe. Une classe comporte donc des variables membre et des fonctions membre.

Nous abandonnerons donc ici l'expression "sous-variable" au profit de "variable membre", qui possède incontestablement plus de cachet, et donnera à notre discours une allure plus techniquement respectable.

[3] Le mot `struct` (qui sert à créer des structures en C) existe encore en C++, où il est un quasi-synonyme de `class`.

5 - Bon, c'est gentil tout ça, mais ça fait déjà 8 pages. Qu'est-ce que je dois vraiment en retenir ?

1) Quand on a besoin d'une variable, il faut la définir, c'est à dire annoncer son type et son nom.
2) C++ connaît les types logique (`bool`) et numériques (`int` pour les entiers, `double` pour les décimaux et "pointeur sur ..." pour les adresses).
3) C++ ne connaît les caractères alphabétiques que comme une façon exotique d'afficher (ou de spécifier) certaines valeurs entières.
4) Les types que C++ ne connaît pas, il ne demande qu'à les apprendre.
5) Pour apprendre un nouveau type à C++, on définit en général une classe (ou, parfois, un simple type énuméré).
6) Un objet dont le type est une classe est appelé une instance de cette classe.
7) Une classe comporte des variables membre et des fonctions membre.

6 - J'ai rien compris, est-ce que quelqu'un d'autre pourrait m'expliquer ça un peu plus clairement ?

Alors là, ce ne sont pas les candidats pédagogues qui manquent ! Et c'est une bonne nouvelle, car si vous avez *vraiment* l'impression d'être passé complètement à côté de la Leçon 2, il est à craindre que ma façon d'aborder la question ne vous convienne pas du tout. Il va donc vous falloir un autre guide, et la liste qui suit vous en propose quelques-uns.

Ivor Horton : Visual C++ 6.0, Eyrolles, Paris, 1999. (ISBN 2-212-094043-9)

Horton n'aborde pas du tout sa présentation du langage de la même façon que moi. La notion de fonction et les types de données sont traités dans le chapitre 2, ainsi que la création de types énumérés. La notion de classe, pour sa part, n'est abordée qu'au chapitre 8 (page 289), tout le reste du langage ayant été préalablement décrit dans ses détails les plus obscurs et les moins usités. A mon humble avis, ce choix rend le manuel aussi précieux comme outil de référence (lorsque vous voulez savoir toute la vérité, ou presque, sur une question particulière) qu'inutilisable en tant que guide pour apprendre le langage. Ceci dit, vous avez le droit de ne pas être d'accord avec moi (en fait, si mon approche vous avait convaincu, vous ne seriez sans doute pas en train de lire ce paragraphe).

Steve Heller : Who's Afraid of C++ ?, Academic Press, 1996 (ISBN 0-12-339097-4)

Cet ouvrage, déjà recommandé en fin de Leçon 1, adopte une démarche exactement inverse de celle de Horton, ce qui en fait un médiocre ouvrage de référence mais lui donne une place de tout premier plan dans mon panthéon didactique personnel. Il a cependant toujours les mêmes défauts : il est en anglais et les raisons mêmes qui me le font aimer risquent de vous le faire détester, si vous n'aimez pas mon cours.

Nino Silverio : Langage C++, Eyrolles, 1996 (ISBN 2-212-08858-2)

L'opposé du Heller : il est en français, et je le trouve d'une médiocrité absolue (présentation pédante, goût très journalistique pour les pseudo-paradoxes, aucune vision claire de ce qui est essentiel et de ce qui relève de l'approfondissement, manque général de rigueur technique). Mais, encore une fois, chacun ses goûts. Peut-être est-ce exactement le livre qu'il vous faut, après tout !

Bjarne Stroustrup : Le langage C++, International Thomson Publishing, 1996 (ISBN 2-84180-079-2)

Stroustrup est le créateur du langage C++, son livre fait donc figure de bible en la matière. Je le trouve personnellement très difficile à lire et ennuyeux. A vous de voir si c'est un indice en sa faveur.

Oleg Yaroshenko : The beginner's guide to C++, Wrox Press, 1994 (ISBN 1-874416-26-5)
Un des nombreux cours complets proposés sur le marché américain. Une fois de plus, je déteste l'hyper-structuration complètement artificielle imposée par les normes pédagogiques US (tous les chapitres bien de la même longueur, avec les mêmes titres de paragraphes rencontrés dans le même ordre, etc.) Je n'adhère pas non plus au rejet de la notion de classe dans le dernier tiers du livre et au ton exagérément pragmatique de l'ensemble. Un côté sympathique : le compilateur utilisé est Borland C++ (mais il n'est pas fourni avec le livre). A vous de voir, mais vous avez bien compris que c'est en anglais, n'est-ce pas ?

Randy Davis : Programmation Windows 95 pour les nuls. Sybex, 1995 (ISBN 2-7361-1667-4)
La saveur inimitable de la collection "pour les nuls", même si celui-ci suppose explicitement que vous ayez déjà une expérience minimum de la programmation (un soupçon de Visual Basic l'année dernière, peut être ?) Pourquoi pas, si vous aimez ça !

Gerhard Willms : Le langage C++. Micro Application, 1998 (ISBN 2-7429-0750-5)
Une approche traditionnelle : on n'aborde les classes qu'en page 637. La présentation qui en est alors faite est une de mes préférées parmi celles disponibles en français. Je ne l'ai pas adopté comme manuel pour trois raisons : le traitement des pointeurs est complètement incompréhensible (en raison de problèmes techniques de préparation du manuscrit), il ne se situe pas dans un contexte Windows (ce qui pourrait être considéré comme une qualité, mais ne correspond pas à l'approche que j'ai choisie) et, pour quelques dizaines de francs de plus, Eyrolles offre un compilateur avec le livre de Horton !

Robert Lafore : C++ Interactive Course. Waite Group Press, 1996 (ISBN 1-57169-063-8)
Ce tutoriel n'est pas un simple livre : il s'accompagne d'une offre de prise en charge personnalisée, basée sur Internet, et propose un examen validé par l'Université Marquette (je n'en avais jamais entendu parler, mais l'idée même est provocante, donc sympathique). La notion de classe est introduite là où elle doit l'être (dès la 11ème des 850 pages), et l'ensemble a l'air plutôt sérieux (mais j'avoue que je n'ai pas encore eu le temps d'examiner la chose en détails). Si je voulais apprendre C++ et que je ne supporte pas l'enseignement qu'on me propose à l'Université, je pense que c'est celui-ci que je choisirais (il est beaucoup plus ambitieux que le Heller, qui est vraiment destiné à des débutants complets et timides). Mais c'est en anglais, évidemment.

Stanley Lippman & Josée Lajoie : C++ primer. Addison Wesley, 1998 (ISBN 0-201-82470-1)
Une présentation très "moderne" du langage, qui met en avant l'utilisation de la librairie standard. Complet et sérieux. Evitez, si possible, la traduction française, dont les nombreuses déficiences et approximations conduisent à un texte ayant perdu beaucoup de la rigueur de l'original.

7 - Pré-requis de la Leçon 3

La Leçon 2 doit avoir été raisonnablement bien comprise.

Il n'est pas nécessaire que les TD 1 et 2 ("Premier dialogue") aient été faits.

Notez quand même que le TD 02 (qui nécessite d'avoir d'abord fait le TD 01) devrait vous aider à bien comprendre la Leçon 2, ce qui sera sans aucun doute un gros avantage pour suivre les Leçons suivantes...