



Centre Informatique pour les **L**ettres  
et les **S**ciences **H**umaines

## Apprendre C++ avec Qt : Leçon 2 Variables, constantes et références

1 - Création de variables .....	2
Réflexion préalable .....	2
Définition .....	2
Initialisation .....	3
Constance .....	3
2 - Création de références.....	3
3 - Les types standard.....	4
Le type booléen.....	4
Les types décimaux .....	4
Les types entiers "ordinaires".....	4
Un type entier "spécial" : char .....	5
D'autres types entiers "spéciaux" : les pointeurs.....	5
Définition .....	5
Initialisation .....	6
Utilisation .....	6
Conversions automatiques .....	6
Conversions explicites .....	7
4 - Définition de nouveaux types .....	7
Les types énumérés .....	7
Les classes .....	8
5 - Bon, c'est gentil tout ça, mais ça fait déjà 9 pages. Qu'est-ce que je dois vraiment en retenir ? .....	10

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 - Création de variables

Comme nous l'avons vu, la logique d'un programme repose sur la façon dont le problème qu'il traite est représenté en mémoire. Créer les variables est donc un acte capital, et commettre une erreur grave à ce niveau ne laisse que peu d'espoir d'arriver à un programme correct.

### Réflexion préalable

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 initiale. Le langage C++ distingue les minuscules des majuscules : exemple et Exemple seront considérés comme étant deux variables différentes. La plupart des programmeurs utilisent des noms de variables écrits en minuscules, et l'interdiction d'utiliser des espaces conduit à adopter un autre moyen pour séparer les mots. On peut capitaliser lesInitialesDesMots ou séparer ceux-ci par des caractères\_de\_soulignement mais, en tous cas, la matérialisation de la limite des mots améliore nettement la lisibilité du texte.

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.

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.

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

## 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 le signe = et la valeur choisie.

Lorsqu'une variable est définie sans être initialisée, on emploie souvent le mot "initialisation" pour faire allusion à la première opération ayant pour effet de lui attribuer une valeur déterminée. Si le terme semble justifié par le changement d'état de la variable concernée, il reste néanmoins techniquement incorrect. La nuance a une réelle importance, car certains effets ne peuvent être obtenus que dans le cadre d'une véritable initialisation.

```
//exemples de définitions de variables comportant une initialisation
```

```
1 entierPositif laVariable = 33;
```

```
2 entierPositif uneAutre = laVariable; //ok : la valeur de laVariable est connue
```

Les numéros figurant en marge des fragments de code comportant plusieurs lignes ne sont qu'un artifice facilitant parfois la discussion. Ils n'ont aucun équivalent dans un texte source réel. Les commentaires (parties de lignes figurant à droite d'une double barre oblique) sont en revanche une partie importante des textes sources. Ils sont ignorés par le compilateur, mais permettent au programmeur de fournir des explications à l'intention des êtres humains qui seront amenés à relire le texte source (pour le corriger ou le modifier, par exemple).

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

```
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

Il arrive que la valeur placée dans une variable lors de sa création ne soit pas destinée à être modifiée au cours de l'exécution du programme. On peut prévenir tout risque de modification accidentelle de la valeur d'une telle variable en faisant de celle-ci une constante. Il suffit pour cela de faire précéder la définition du mot const :

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

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

Etant donné que la valeur de ce genre de "variables" ne peut plus changer après leur définition,

Il faut obligatoirement initialiser les constantes.

## 2 - Création de références

Il arrive qu'il soit utile d'attribuer un nouveau nom à un objet (une variable, par exemple) qui existe déjà. La syntaxe utilisée pour ce faire est très proche de celle d'une déclaration de variable initialisée : le type et le nouveau nom sont annoncés, et la "valeur d'initialisation" est remplacée par une expression désignant l'objet qui reçoit le nouveau nom. Le fait qu'il ne s'agit pas d'une définition de variable est signalé par l'introduction, entre le type et le nouveau nom, du signal de création de référence :

```
1 entierPositif nabuchodonosor = 4; //création d'une variable
```

```
2 entierPositif & toto = nabuchodonosor; //toto est un "surnom" de nabuchodonosor
```

L'attribution d'un nouveau nom à un objet ne rend pas inutilisable le nom que possédait précédemment cet objet. En général, la création de références ne sert pas, comme le suggère cet exemple, à fournir un nom plus court pour un objet dont la désignation initiale a été mal choisie, mais plutôt à baptiser des objets qui n'ont pas encore de noms.

L'objet désigné par une référence doit être spécifié lors de la création de celle-ci et ne pourra pas changer au cours de l'exécution du programme.

Si l'objet qui reçoit un nouveau nom est une constante, la référence créée doit être une référence à un objet constant, de façon à ce que le compilateur puisse continuer à garantir l'immutabilité de la constante, même lorsque celle-ci est manipulée à l'aide de son "surnom" :

```
1 const decimal TVA = 1.206; //Ne varie pas pendant l'exécution du programme.
2 const decimal & VAT = TVA; //Synonyme de TVA pour les anglophones.
3 decimal & taxe = TVA;      //ERREUR ! une référence à un objet variable
4                            //ne peut pas désigner un objet constant.
```

Une "référence à un objet constant" peut, en revanche, être associée à un objet non constant : l'usage de cette référence ne permettra alors pas de modifier la valeur de l'objet en question.

```
1 decimal lePrix = 1247.38;
2 const decimal & ETIQUETTE = lePrix;
```

La création d'une référence ne peut donc en aucun cas procurer des privilèges dont on ne disposait pas avant. Elle peut en revanche produire une désignation d'un objet offrant *moins* de possibilités que la désignation initiale.

### 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 booléens
1 bool uneVariable;
2 bool uneAutre = true;
3 bool encoreUne(false);
4 const bool VRAI = true;
```

#### 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 offrir une plage de valeurs possibles plus étendue et permettre une plus grande précision (`long double`).

```
//quelques exemples de définition de décimaux
1 const double PI = 3.14;
2 float unNombre;
3 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 (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`). 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
1 short unEntier;
2 int unAutreEntier = -36;
3 long unTroisiemeEntier;
4 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 objets 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
```

### D'autres types entiers "spéciaux" : les pointeurs

Les pointeurs sont des objets destinés à contenir des adresses.

Nous avons vu que les adresses sont un système de numérotation des cases mémoire servant à les désigner. Ce n'est pas parce que le langage C++ permet aussi de désigner certaines cases mémoire à l'aide du nom de la variable à laquelle elles correspondent que nous devons renoncer à utiliser les adresses. C'est de toute façon ainsi 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 adresses ne sont que des numéros, pourquoi ne pas les stocker dans des variables "ordinaires", de type int, par exemple ? C'est tout à fait possible, et il arrive qu'on soit amené à le faire. En général, toutefois, cette façon de procéder n'est pas satisfaisante, car elle sacrifie inutilement les avantages offerts par la notion de type. Les types, rappelons-le, permettent au compilateur d'une part de vérifier la légitimité des opérations que nous entreprenons, et, d'autre part, de nous décharger du détail des opérations en question.

Souvenez-vous : nous n'avons aucune envie d'avoir à préciser (et donc à savoir...) ce qu'il faut faire pour modifier la représentation d'un nombre décimal de façon à ce qu'elle devienne, par exemple, 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...

Les pointeurs offrent précisément un moyen de manipuler des adresses sans perdre la trace du type des objets qui sont représentés dans les zones mémoires que ces adresses désignent.

#### Définition

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 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 du **type**. Ainsi,

```
double * unPointeur;
```

ne crée pas une variable de type double (qui permettrait de stocker une quantité décimale), mais une variable de type "pointeur sur double" (capable de stocker l'adresse d'un double, c'est à dire une valeur entière).

## Initialisation

Une variable de type pointeur doit donc recevoir une valeur correspondant à l'adresse d'un objet du type "promis" par le type du pointeur. On obtient l'adresse d'une variable à l'aide d'un opérateur spécialement prévu à cet effet, l'opérateur "adresse de", noté &.

```
1 double unNombre = 36.2;
2 double * ptr = & unNombre;
//la variable ptr contient l'adresse de la variable unNombre, et non 36.2
```

Le symbole & a donc deux significations : lorsqu'il apparaît entre un type et un nom en cours de déclaration, il indique que c'est une simple référence qui est créée, et non une variable. Dans tous les autres cas, il s'agit de l'opérateur "adresse de".

## Utilisation

Lorsqu'un pointeur contient une adresse, on accède à l'objet habitant à cette adresse en **déréférencant** le pointeur à l'aide d'un opérateur spécialement prévu à cet effet, noté \*.

```
3 double unAutre = *ptr;
//unAutre contient 36.2, et non l'adresse de unNombre contenue dans ptr
```

Le langage C++ utilise le même symbole pour l'opérateur de déréférencement et pour signaler la définition d'un pointeur. Lorsque l'étoile apparaît entre le **type** et le **nom** d'un objet en cours de déclaration, elle indique que c'est un pointeur qui est créé.

Si la valeur contenue dans un pointeur n'est pas l'adresse d'un objet du type correspondant à celui du pointeur, on dit que ce pointeur est **invalide**. Déréférencer un tel pointeur conduirait à accéder à une zone de mémoire ne contenant pas le type d'information attendue, ce qui a peu de chances de produire un résultat intéressant. Lorsqu'un pointeur est invalide, il est préférable de lui donner la valeur nulle, qui indique explicitement que le pointeur ne doit pas être déréférencé (par convention, aucun objet C++ n'habitera jamais à l'adresse 0).

## 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.

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 entier, l'expression

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

sera évaluée comme ayant 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
```

## 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 l'une des cinq valeurs envisageables, et aucune autre. C++ vous permet de créer ce type de la façon suivante :

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

Il faut bien comprendre que la ligne de code précédente *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
```

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 :

```
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.

## 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.

Lorsqu'une variable a pour type une classe, on dit qu'elle est une **instance** de cette classe.

Lorsqu'une instance est créée, on dit que la classe concernée est *instanciée*. Le nombre de variables adoptant un même type n'étant évidemment pas limité, une classe peut être instanciée un grand nombre de fois dans le même programme.

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. 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 impliquée dans un programme. N'investissez donc pas le mot objet d'une signification technique trop stricte.

L'orientation objet a connu, il y a quelques années, une vogue assez 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...

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 :

```
1 //définition du type ClasseCouleur
2 class ClasseCouleur
3 {
4 public:
5     int quantiteDeRouge;
6     int quantiteDeVert;
7     int quantiteDeBleu;
8 };

```

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. 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.

La définition (création) d'une classe ne définit (crée) aucune variable membre. Les variables membres sont simplement *déclarées* (au compilateur) lors de la création de la classe.

La création d'une classe est analogue à la mise au point d'un plan de maison : tant qu'on en reste là, il n'existe aucun objet dans lequel habiter (ou stocker des valeurs...)



Ce n'est que 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
```

Instancier une classe, c'est un peu comme construire une maison en suivant le plan.

Comment se présente alors la variable `uneCouleur` ? Elle a trois "sous-variables", désignées par des noms formés en combinant celui de la *variable* avec celui de la "*sous-variable*" considérée. Cette combinaison est faite à l'aide d'un *opérateur de sélection*, le point. En l'occurrence, après la définition de la variable `uneCouleur`, nous disposons de trois variables nommées respectivement `uneCouleur.quantiteDeRouge`, `uneCouleur.quantiteDeVert` et `uneCouleur.quantiteDeBleu`.

C'est lors de l'instanciation d'une classe que des variables sont effectivement créées.

Lorsqu'une instance n'est pas désignée par son nom, mais par un pointeur contenant son adresse, l'accès aux "sous variables" est fourni par l'opérateur de *sélection indirecte*, noté `->`, qui déréférence le pointeur et sélectionne l'un des "sous objets" de l'objet pointé :

```
ClasseCouleur * ptr = & uneCouleur;  
//ptr->quantiteDeRouge équivaut à uneCouleur.quantiteDeRouge
```

S'il ne s'agissait que de cela, les classes seraient déjà utiles, puisqu'elles permettraient une certaine organisation des données. Une des caractéristiques majeures des types prédéfinis leur 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 qui lui a été attribuée.

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, la `ClasseCouleur` indique effectivement combien d'octets doivent être réservés pour une variable de ce type, et comment l'information y est codée. Cependant, rien n'indique encore 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, qui sont, elles, d'un type prédéfini (int).

Pour que des ordres puissent être appliqués directement à des variables dont le type est une classe, il faut que la définition de cette classe indique comment ces ordres doivent être exécutés. Nous sommes donc conduits à conclure que la classe doit comporter du code décrivant les traitements qui peuvent être appliqués aux données que ses instances représentent. Comme nous savons que le code C++ se présente sous la forme de 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.

## 5 - Bon, c'est gentil tout ça, mais ça fait déjà 9 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) Pour créer une variable, il faut la définir, c'est à dire annoncer son type et son nom.
- 2) Si une variable est `const`, elle n'est pas variable.
- 3) Pour donner un deuxième nom à une variable, on crée une référence initialisée avec la variable en question.
- 4) C++ connaît les types logique (`bool`) et numériques (`char` ou `int` pour les entiers, `double` pour les décimaux et "pointeur sur ..." pour les adresses).
- 5) C++ ne connaît les caractères alphabétiques que comme une façon exotique d'afficher (ou de spécifier) les valeurs entières de type `char`.
- 6) Les types que C++ ne connaît pas, il ne demande qu'à les apprendre.
- 7) Pour apprendre un nouveau type à C++, on définit en général une classe (ou, parfois, un simple type énuméré).
- 8) Définir une classe, c'est déclarer ses membres.
- 9) Une classe comporte des variables membre et des fonctions membre.
- 10) Un objet dont le type est une classe est appelé une instance de cette classe.
- 11) Lorsqu'une classe est instanciée, un jeu de variables membre (qui constitue, précisément, la nouvelle instance) est créé.
- 12) On accède aux variables membres composant une instance en liant la désignation de l'instance et le nom du membre à l'aide d'un opérateur de sélection (un point si l'instance est désignée par son nom ou par une référence qui lui est associée, une flèche si elle est désignée par un pointeur contenant son adresse).