



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

Apprendre C++ avec Qt : Leçon 10 Constructeurs et destructeurs

1 - La surcharge des noms de fonctions.....	2
2 - Constructeurs.....	4
Des fonctions invisibles ?	4
Déclaration, définition et appel explicites d'un constructeur.....	4
Constructeurs par défaut	5
Constructeurs par copie	6
Constructeurs de transtypage	7
Constructeurs à arguments multiples	8
De quels constructeurs disposons-nous ?.....	9
Echec d'un constructeur	9
3 - Listes d'initialisations.....	10
4 - Destructeurs.....	11
5 - Bon, c'est gentil tout ça, mais ça fait déjà 10 pages. Qu'est-ce que je dois vraiment en retenir ?	12

Notre utilisation du mécanisme des classes reste encore assez rudimentaire, et il nous manque en particulier une méthode permettant d'initialiser les variables membre lorsqu'un objet est créé. Le propos de cette Leçon est donc d'examiner plus en détail le processus d'instanciation.

Deux autres thèmes seront abordés : la destruction des objets et la surcharge des noms de fonctions. Le thème de la destruction est logiquement lié à celui de l'instanciation, puisqu'il s'agit du processus inverse. Comme il s'agit en outre d'un phénomène qui ne nécessite qu'un exposé très bref, il est naturel de traiter ces deux sujets dans la même Leçon. La surcharge n'a, en revanche, pas de rapport logique avec l'instanciation ou la destruction, et aborder ce thème ici peut sembler tout à fait arbitraire, voire contestable. Il se trouve simplement que surcharger les noms de fonctions n'est ni une opération primordiale (ce qui explique que nous n'ayons pas encore abordé le sujet), ni une technique complexe (ce qui explique qu'elle ne fasse pas l'objet d'une Leçon spécifique), ni un procédé que l'on peut ignorer lorsqu'on s'intéresse au processus d'instanciation (ce qui explique que nous allons commencer par explorer cette question).

1 - La surcharge des noms de fonctions

Le langage C++ accepte, sous certaines conditions, l'existence de fonctions portant le même nom. On dit alors que le nom en question est surchargé (puisque'il sert à désigner plusieurs fonctions différentes). L'usage courant a cependant imposé une expression plus simple :

On dit qu'une fonction est surchargée lorsqu'il existe (au moins) une autre fonction portant exactement le même nom.

Attention, en dépit du raccourci verbal généralement employé, ce n'est pas la fonction qui se trouve surchargée, mais bien son nom. Cette "surcharge" n'implique rien de spécial pour la fonction, dont la déclaration et la définition restent parfaitement inchangées.

L'existence de fonctions portant le même nom pose évidemment un problème : lorsque ce nom est utilisé dans une expression, quel est le code qui doit être exécuté pour évaluer celle-ci ? Le compilateur doit disposer d'un critère de décision clair, et le langage impose donc des conditions pour que l'homonymie entre fonctions soit acceptable :

Deux fonctions ne peuvent porter le même nom que si elles diffèrent par leur caractère constant ou par le type d'au moins un de leurs paramètres.

Il est donc possible de définir deux fonctions différentes dont les déclarations seraient

```
1 void f( int n, double x);  
2 void f(char n, double x);
```

puisque le premier paramètre de la première est de type `int` alors que le premier paramètre de la seconde est de type `char`. L'ordre des paramètres est bien entendu significatif, et

```
3 void f(int n, double x); //déclaration d'une fonction  
4 void f(double x, int);  //déclaration d'une fonction homonyme de la précédente
```

sont également des déclarations correspondant à deux fonctions différentes.

Les noms attribués aux paramètres sont en revanche sans importance, et les lignes

```
1 void f(int n, double x);           //déclaration d'une fonction  
2 void f(int machin, double truc);  //redéclaration de la fonction précédente
```

constituent en fait deux déclarations de la même fonction¹ et non les déclarations de deux fonctions homonymes. Comme une fonction ne peut être définie qu'une seule fois, toute tentative pour définir la pseudo seconde fonction se soldera par un message indiquant que cette définition est inacceptable, car l'unique fonction `f()` existe déjà.

Dans le cas de fonctions membre, le fait que seule l'une d'entre elles soit privée du droit de modifier les variables membre de l'instance au titre de laquelle elle est invoquée suffit à les distinguer, et l'on peut donc avoir :

```
1 class CExemple  
2 {public:  
3     void f(int x, double y);           //une première fonction membre  
4     void f(int x, double y) const;    //une seconde fonction membre  
5 };
```

¹ Rappels : Un objet peut être déclaré plusieurs fois, à condition que toutes ces déclarations soient identiques. Les noms des paramètres ne figurent dans la déclaration qu'à titre documentaire, et sont ignorés par le compilateur.

La constance et la liste (ordonnée) des types de ses arguments constituent ce qu'on appelle la **signature** d'une fonction.

La condition pour que deux fonctions puissent porter le même nom est donc que leurs signatures diffèrent. C'est grâce à ces signatures que le compilateur détermine quel code doit être exécuté lorsque l'évaluation d'une expression implique l'appel d'une fonction surchargée.

Si nous disposons de deux fonctions déclarées ainsi

```
1 void g(bool x); //déclaration d'une fonction
2 void g(char * s); //déclaration d'une fonction homonyme de la précédente
```

il est en effet clair que des lignes telles que

```
1 bool test = true;
2 g(test);
3 g(false);
```

se traduiront par l'exécution de la première fonction `g()` (celle qui dispose d'un paramètre de type `bool` pour recevoir la valeur booléenne transmise), alors que les lignes

```
4 char texte[] = "Bravo";
5 char c = 'a';
6 g(texte);
7 g(&c);
```

déclencheront l'exécution de la seconde (celle qui dispose d'un paramètre de type "pointeur sur char" pour recevoir l'adresse transmise).

Dans le cas de fonctions membre dont les signatures ne diffèrent que parce que l'une des deux est constante, la règle appliquée est simple : la fonction "non constante" est exécutée, sauf lorsque l'appel est effectué au titre d'une instance constante (seule la fonction qui ne peut pas modifier les variables membre est alors légitime).

L'utilisation des signatures pour choisir entre des fonctions homonymes interagit parfois avec le mécanisme de conversion automatique. Ainsi, si nous disposons d'une fonction déclarée

```
void h(int a); //première fonction h()
```

l'appel suivant est tout à fait possible :

```
h('x');
```

En effet, bien que 'x' soit de type `char`, le compilateur est capable de convertir automatiquement sa valeur en un `int` convenant à l'initialisation du paramètre de la fonction `h()`. Imaginons maintenant que l'évolution du programme conduise à introduire une fonction déclarée ainsi :

```
void h(char a); //seconde fonction h()
```

Les signatures diffèrent, l'homonymie est donc acceptée. Les lignes de code qui utilisaient une expression de type `char` pour spécifier la valeur transmise à la première fonction `h()` déclenchent donc désormais l'exécution de la seconde. Ceci est généralement souhaitable, mais il faut éviter d'utiliser la surcharge dans les cas où ce phénomène serait inopportun.

Par ailleurs, l'utilisation de valeurs par défaut conduit certaines fonctions à avoir plusieurs signatures. Valeurs par défaut et surcharge peuvent donc s'avérer incompatibles. Imaginons, par exemple, que nous disposions de deux fonctions déclarées ainsi :

```
void k(int a, char b = 'x');
void k(int a);
```

Ces deux fonctions présentent effectivement des signatures différentes : `(int, char)` dans un cas, et `(int)` dans l'autre. Du fait de la valeur par défaut dont dispose son second paramètre, la première fonction dispose toutefois d'une signature "alternative", réduite, elle aussi, à un simple `int`. Il n'y a donc plus de moyen de déterminer si

```
k(12);
```

doit se traduire par l'appel de la première fonction `k()`, avec utilisation de la valeur par défaut du second paramètre, ou par l'appel de la seconde fonction `k()`.

La surcharge est indispensable dans certains cas, et elle peut être utile lorsqu'elle permet aux utilisateurs d'une collection de fonctions d'utiliser un même nom pour appliquer un "même" traitement à des données de types différents. Un exemple très simple et très familier est celui de l'addition : il vous semble naturel d'utiliser le même symbole pour additionner deux entiers et pour additionner deux décimaux, même si vous savez (Leçon 1) que les opérations nécessaires pour traiter ces deux cas diffèrent profondément. La surcharge vous permet de doter les fonctions que vous écrivez de la même souplesse : selon le type des arguments, le détail des opérations peut varier considérablement, mais l'utilisateur ne connaît qu'un seul nom et n'a pas à prendre lui-même en compte le type des arguments.

2 - Constructeurs

Les classes que nous créons sont généralement destinées à nous permettre de stocker des données. La définition d'une classe n'est alors qu'une première étape : il nous faut ensuite l'instancier. Ce sont ces instances qui nous permettent finalement de stocker et de manipuler efficacement nos données. L'intérêt du mécanisme des classes repose donc en grande partie sur le processus d'instanciation, processus que nous avons utilisé jusqu'à présent sans nous inquiéter outre mesure des opérations qu'il implique. La plupart de ces opérations restent sous la responsabilité exclusive du compilateur, et nous n'aurons jamais à nous y intéresser. Le langage C++ nous offre cependant un moyen d'influer, lorsque nous le jugeons nécessaire, sur le processus d'instanciation des classes que nous avons créées :

La création d'une instance d'une classe s'accompagne toujours de l'exécution d'une fonction particulière, que l'on appelle un constructeur de la classe.

Des fonctions invisibles ?

Les constructeurs présentent de nombreuses caractéristiques qui les distinguent des fonctions "ordinaires". Une des caractéristiques les plus déroutantes pour le programmeur novice est sans doute que ces fonctions sont très souvent **appelées implicitement**, c'est à dire sans que la ligne de code qui déclenche leur exécution présente les caractéristiques qui permettent habituellement de reconnaître un appel de fonction : le nom de la fonction, suivi d'un couple de parenthèses encadrant la (ou les) expression(s) déterminant la (ou les) valeur(s) transmise(s) comme argument(s). L'étrangeté de la situation est en outre amplifiée par le fait que **le compilateur crée automatiquement certains constructeurs**, qui n'apparaissent alors pas dans le code définissant la classe.

Imaginons, par exemple, une classe définie de la façon suivante :

```
1 class CTresSimple
2 {
3 public :
4     int leMembre;
5 };
```

A première vue, cette classe ne comporte aucune fonction, et il est donc assez difficile d'imaginer que la création d'une variable locale dans une fonction quelconque, à l'aide d'une ligne aussi anodine que

```
CTresSimple uneInstance; //instanciation innocente de notre classe
```

se traduit en fait par l'appel (invisible) d'une fonction membre qui n'est ni déclarée ni définie par notre code. C'est pourtant très exactement ce qui se produit : le compilateur ajoute automatiquement un constructeur à la classe, et ce constructeur est (implicitement) appelé lors de la création d'une instance.

Le rôle du compilateur est de produire un exécutable à partir du texte source, et certainement pas de modifier ou de compléter le texte source. Un constructeur ajouté automatiquement à une classe par le compilateur n'existe donc à aucun moment sous forme d'un texte en C++ que nous pourrions consulter.

Ce mécanisme quelque peu opaque s'éclaircit rapidement dès lors que les constructeurs sont explicitement déclarés, définis et appelés.

Déclaration, définition et appel explicites d'un constructeur

Un constructeur est une **fonction membre** qui porte le **même nom que la classe** et est **dépourvue de type**.

Attention : c'est bien de type que les constructeurs sont dépourvus, et pas simplement de valeur de retour. En d'autres termes, les constructeurs ne sont même pas de type void.

Pour reprendre l'exemple précédent, la classe CTresSimple peut donc être définie ainsi :

```
1 class CTresSimple
2 {
3 public :
4     int leMembre;
5     CTresSimple();    //déclaration d'un constructeur :- pas de type
                        // - même nom que la classe
6 };
```

La définition du constructeur en question ne présente pas de particularités par rapport à celle de n'importe quelle autre fonction membre : on reprend la déclaration (en prenant toutefois soin d'utiliser le **nom complet** de la fonction), mais on remplace le point virgule final par un couple d'accolades contenant le code de la fonction.

```
1 CTresSimple::CTresSimple()
2 {
3     //le bloc de code définissant ce constructeur est vide
4 }
```

Il peut sembler choquant que le bloc de code définissant le constructeur reste vide. Contrairement aux apparences, cela ne signifie pas que l'exécution du constructeur soit sans intérêt. Une des caractéristiques très particulières des constructeurs est que leur exécution provoque la création d'une instance avant que le bloc de code qui les définit ne commence à être exécuté. Le véritable processus de création d'une instance est du ressort exclusif du compilateur, il ne peut pas réellement être décrit en C++ "normal".

La déclaration et la définition explicites de ce constructeur ne changent rien aux possibilités d'utilisation de la classe CTresSimple, qui peut toujours être instanciée normalement :

```
CTresSimple uneInstance; //appel implicite du constructeur
```

Notons également qu'il est aussi possible d'appeler explicitement le constructeur lors de l'instanciation de la classe :

```
CTresSimple uneInstance = CTresSimple(); //appel explicite du constructeur
```

Un tel appel explicite est toujours possible, que le constructeur ait ou non été défini explicitement. Il reste toutefois d'un usage assez rare dans le cas de la définition d'une variable, car il en alourdit l'écriture sans présenter d'avantages vraiment significatifs.

Constructeurs par défaut

Les constructeurs que nous avons rencontrés jusqu'à présent sont dépourvus d'arguments, et permettent donc d'obtenir l'instanciation de la classe sans avoir à fournir de données.

Un constructeur dépourvu d'arguments est appelé un constructeur par défaut.

Cette absence d'argument restreint la souplesse d'usage du constructeur, et nous serons rapidement amenés à envisager des constructeurs plus élaborés. Le constructeur par défaut assure cependant un "service minimum" qui permet d'instancier la classe. C'est pourquoi

Lorsque le code définissant une classe ne comporte aucun constructeur, le compilateur rend la classe instanciable en lui ajoutant automatiquement un constructeur par défaut.

Le constructeur par défaut fourni par le compilateur correspond à un constructeur dont le bloc d'instructions serait vide. Lorsque le constructeur est défini explicitement, il est bien entendu possible d'y placer des instructions, et celles-ci seront alors exécutées comme si le constructeur avait été invoqué au titre de l'instance qui vient d'être créée.

Une des missions qui échoient naturellement à un constructeur est l'initialisation des variables membre, et le constructeur par défaut de la classe que nous avons imaginée dans les exemples précédents pourrait être défini ainsi :

```
1 CTresSimple::CTresSimple()
2 {
3     leMembre = 0;
4 }
```

L'intérêt des constructeurs devient alors évident : étant donné que l'instanciation de la classe CTresSimple s'accompagne automatiquement de l'appel du constructeur, il devient

rigoureusement impossible qu'une variable de ce type soit créée sans que sa variable membre ne reçoive une valeur initiale. Une source d'erreurs de programmation se trouve ainsi éliminée.

Constructeurs par copie

La définition d'une variable n'est pas le seul cas où une classe se trouve instanciée. Lors de l'appel d'une fonction utilisant comme paramètre une instance de la classe, il est également nécessaire de créer une instance.

En effet, comme nous le savons depuis la Leçon 5, la fonction ne va pas travailler sur l'objet utilisé pour spécifier la valeur de son paramètre, mais va simplement utiliser cette valeur pour initialiser son propre objet (le paramètre lui-même). Ceci suppose donc une opération d'initialisation de l'objet nouvellement créé (le paramètre) à l'aide des valeurs contenues dans l'objet utilisé par la fonction appelante pour spécifier la valeur de ce paramètre. Chaque fois que nous écrivons quelque chose comme :

```
1 CTresSimple uneInstance; //le constructeur par défaut initialise leMembre à 0
2 uneInstance.leMembre = 17; //on affecte une nouvelle valeur au membre
3 maFonction(uneInstance);
```

la fonction appelée se trouve en position de devoir *initialiser* son paramètre avec la *valeur transmise*. Une situation analogue se présente si nous écrivons :

```
1 CTresSimple uneInstance;
2 uneInstance.leMembre = 18;
3 CTresSimple uneAutre = uneInstance; //initialisation d'une instance
```

Dans un cas comme dans l'autre, le constructeur ne doit pas initialiser `leMembre` de l'instance en cours de création² avec la valeur 0, mais avec la valeur contenue dans `leMembre` de l'instance qui sert de "modèle", c'est à dire 17 dans le premier cas, et 18 dans le second. Quelle que soit la façon dont il est défini, le constructeur par défaut est parfaitement incapable de réaliser cette opération car, étant lui-même dépourvu de paramètre, il est condamné à utiliser toujours la même valeur pour initialiser `leMembre`. Lorsque ce genre d'initialisation est nécessaire, il nous faut donc disposer d'un autre type de constructeur, capable d'utiliser une instance existant déjà comme "modèle" de l'instance à créer. Ce "modèle" sera communiqué au constructeur par le biais d'un paramètre de type "référence à une instance constante".

Il n'est pas envisageable que le paramètre soit de type "instance", puisque nous venons de voir que l'utilisation ce type de paramètre est précisément l'un des cas qui exige la mise en œuvre du constructeur qu'il s'agit ici de définir ! L'usage d'une référence évite ce cercle vicieux et, comme le propos du constructeur par copie n'est certainement pas de modifier l'instance qui lui sert de modèle, il est préférable que son paramètre désigne l'objet en question comme étant constant.

On appelle constructeur par copie un constructeur qui reçoit comme unique paramètre une référence à une instance de la classe.

Un constructeur par copie a normalement vocation à s'inspirer de l'objet qui lui est communiqué pour initialiser les variables membre de l'instance en cours de création. Dans de nombreux cas, cette "inspiration" consiste purement et simplement à donner aux variables membre de la nouvelle instance des valeurs identiques à celles rencontrées dans l'instance qui sert de modèle, ensemble d'opérations que l'on qualifie souvent de "copie membre à membre".

Nous rencontrerons bientôt des cas où le constructeur par copie ne peut se contenter d'une approche aussi rudimentaire. Un des cas classiques est celui des variables membre de type pointeur : un constructeur par copie "membre à membre" produit une instance qui pointe au même endroit que le modèle utilisé, ce qui n'est pas toujours l'effet souhaité, et peut même s'avérer fort dangereux.

La présence d'un constructeur par copie est indispensable à l'utilisation d'une instance de la classe pour en initialiser une autre. Etant donné qu'une telle initialisation est implicitement effectuée dès qu'une fonction utilise un paramètre de type "instance de la classe", un constructeur par copie est une fonction membre très souvent requise, ce qui justifie que

Le compilateur génère automatiquement un constructeur par copie pour toute classe qui en est dépourvue alors que l'usage qui en est fait le nécessite.

² Le paramètre dans le premier cas, la variable `uneAutre` dans le second.

Le constructeur par copie généré par le compilateur se contente de l'approche rudimentaire évoquée ci-dessus : chaque variable membre du nouvel objet adopte pour valeur celle du membre correspondant de l'instance qui sert de modèle. Dans le cas de notre classe, s'il était visible, le constructeur généré automatiquement ressemblerait donc à

```

1 CTresSimple::CTresSimple(const & CTresSimple leModele)
2 {
3     leMembre = leModele.leMembre;
4 }

```

Les constructeurs par défaut et par copie d'une même classe portent évidemment le même nom (celui de la classe) et ne se distinguent que par la liste des types de leurs arguments. Comme des fonctions de ce type sont fréquemment générées automatiquement par le compilateur, il est fort probable que vous ayez déjà créé et utilisé (sans le savoir) des fonctions surchargées !

Constructeurs de transtypage

Les constructeurs par défaut et par copie ne sont pas les seuls types de constructeurs envisageables. Selon la nature de la classe, il peut en effet être assez naturel d'utiliser, pour spécifier l'état initial d'une instance, autre chose qu'une autre instance de cette même classe. Dans le cas de notre classe CTresSimple, il serait assez tentant de pouvoir initialiser directement l'unique variable membre à l'aide d'une valeur entière. Ceci devient possible dès lors que la classe dispose d'un constructeur admettant pour paramètre un int :

```

1 class CTresSimple
2 {public :
3     int leMembre;
4     CTresSimple();           //constructeur par défaut
5     CTresSimple(int valeur); //constructeur de transtypage à partir d'int
6 };

```

Le constructeur en question peut être défini ainsi :

```

1 CTresSimple::CTresSimple(int valeur)
2 {
3     leMembre = valeur;
4 }

```

Il est alors possible d'appeler explicitement ce constructeur

```
CTresSimple uneInstance = CTresSimple(3);
```

On pourrait croire que la ligne précédente provoque d'abord l'appel du constructeur de transtypage pour créer une instance temporaire et anonyme qui serait ensuite utilisée par un constructeur par copie pour créer uneInstance. Il n'en est rien, et cette ligne ne génère qu'un unique appel, au constructeur de transtypage.

Toutefois, on préfère généralement la simplicité syntaxique d'un appel implicite

```
CTresSimple uneInstance(3); //appel implicite du constructeur de transtypage
```

qui peut aussi être obtenu en utilisant la notation "traditionnelle" de l'initialisation

```
CTresSimple uneInstance = 3; //appel implicite du constructeur de transtypage
```

Il faut aussi remarquer que la disponibilité d'un constructeur de transtypage permet au compilateur d'effectuer automatiquement certaines conversions, ce qui autorise par exemple l'affectation suivante :

```

1 CTresSimple uneInstance;           //appel implicite du constructeur par défaut
2 uneInstance = 15;                  //affectation avec transtypage automatique !

```

Cette affectation peut sembler un peu "magique", mais le déroulement des opérations est finalement assez simple :

- L'expression placée à droite de l'opérateur d'affectation est évaluée. Il s'agit d'une constante littérale, dont la valeur est 15 et le type int.
- L'expression placée à gauche de l'opérateur d'affectation est évaluée. Il s'agit du nom d'une variable dont le type est CTresSimple, et l'expression désigne donc une zone de mémoire susceptible de stocker le résultat obtenu à droite, sous réserve que les types soient

"compatibles" (deux types sont compatibles lorsqu'ils sont identiques ou lorsque l'on sait comment produire une valeur du type de gauche à partir d'une valeur du type de droite).

- Les types `int` et `CTresSimple` ne sont pas identiques, mais le compilateur sait comment produire une valeur de type `CTresSimple` à partir d'une valeur de type `int` : il suffit de transmettre cette dernière au constructeur de `CTresSimple` qui attend ce type de paramètre ! Une fois ceci fait, il ne reste plus qu'à appliquer l'opérateur d'affectation entre deux objets de types `CTresSimple` : la variable `uneInstance` et l'instance (temporaire et anonyme) produite par le constructeur à partir de l'entier 15.

Cette apparente possibilité d'affectation à une instance de la classe d'une valeur d'un type différent explique la terminologie employée :

On appelle constructeur de transtypage un constructeur qui permet la création d'une instance à partir d'une valeur d'un autre type que la classe elle-même.

Il arrive parfois qu'un constructeur de transtypage permette au compilateur d'effectuer automatiquement des conversions qui ne sont pas souhaitables. On peut alors interdire l'usage automatique du constructeur concerné en faisant précéder sa déclaration du mot `explicit`. Si notre classe est définie ainsi

```
class CTresSimple
{
public :
    int leMembre;
    CTresSimple();           //constructeur par défaut
    explicit CTresSimple(int valeur); //constructeur de transtypage
};
```

la conversion d'un `int` en `CTresSimple` n'est plus effectuée que si elle explicitement demandée

```
CTresSimple uneInstance;           //appel implicite du constructeur par défaut
uneInstance = 15;                  //ERREUR : transtypage automatique interdit !
uneInstance = CTresSimple(15);     //OK : appel explicite du constructeur
```

ou si elle intervient lors d'une initialisation

```
CTresSimple incroyable(3);         //appel implicite d'un constructeur explicit !
```

La possibilité d'effectuer l'initialisation d'une instance à partir d'une valeur d'un autre type dépend bien entendu de la nature de la classe considérée et de l'usage qui en est fait. Il n'est donc pas possible pour le compilateur d'inventer des constructeurs de transtypage, et ceux-ci n'existent que dans la mesure où l'auteur de la classe les a définis. Il est, par ailleurs, tout à fait possible de définir des constructeurs exigeant plusieurs arguments (et qui ne sont donc ni "par défaut", ni "par copie", ni "de transtypage").

Constructeurs à arguments multiples

Imaginons que nous disposions d'une classe comportant plusieurs variables membre. Si nous souhaitons pouvoir initialiser plusieurs de ces variables avec des valeurs quelconques, il nous faut bien entendu un constructeur disposant d'autant de paramètres.

```
1 class CDuo
2 {
3 public:
4     double m_double;
5     int     m_int;
6     CDuo (double unDouble, int unInt); //constructeur à deux arguments
7 };
```

La définition d'un constructeur de ce genre est sans surprise :

```
1 CDuo::CDuo (double unDouble, int unInt)
2 {
3     m_double = unDouble;
4     m_int = unInt;
5 }
```

Définie ainsi, notre classe se prête à l'instanciation grâce à l'une des syntaxes suivantes :

```

1 CDuo uneInstance = CDuo(0.0, 0);    //appel explicite du constructeur
2 CDuo uneAutre(3.14, 18);           //appel implicite du constructeur

```

Il faut toutefois noter que, étant donné qu'un constructeur comportant plusieurs arguments n'est pas un constructeur de transtypage, il n'est pas possible d'en utiliser un appel implicite pour effectuer l'affectation d'une liste de valeurs à une instance de notre classe :

```
uneInstance = {2.7, 36};    // IMPOSSIBLE !
```

Une affectation n'est ici possible qu'entre deux objets de même type, ce qui exige qu'une instance (temporaire et anonyme) soit créée par **appel explicite du constructeur** :

```
uneInstance = CDuo(2.7, 36); // OK
```

L'appel explicite d'un constructeur muni de paramètres permet de créer "à la volée" une valeur ayant pour type la classe mentionnée.

De quels constructeurs disposons-nous ?

Le fait que certains constructeurs puissent parfois être générés automatiquement par le compilateur simplifie considérablement la mise en œuvre des classes les plus simples. Ce phénomène risque, en revanche, de générer une certaine confusion dans l'esprit du programmeur débutant, qui connaît l'existence de ces constructeurs, mais maîtrise encore mal les règles qui les gouvernent. Le tableau ci-dessous résume les caractéristiques essentielles des quatre types de constructeurs :

Constructeur	Signature	Génération automatique
par défaut	Pas de paramètre.	Lorsque la définition explicite de la classe ne comporte aucun constructeur.
par copie	Paramètre unique, de type "référence à une instance de la classe".	Lorsque la définition explicite de la classe ne comporte pas de constructeur par copie et que l'usage fait de la classe le nécessite.
de transtypage	Un seul paramètre, d'un type autre que la classe elle-même.	Jamais.
autres	Plusieurs paramètres.	Jamais.

Un des points remarquables de cet ensemble de règles est que

La définition explicite d'un constructeur, quel qu'il soit, inhibe la génération automatique d'un constructeur par défaut.

Ainsi, si la définition d'une classe ne comporte aucun constructeur déclaré explicitement, la simple adjonction d'un constructeur de transtypage, par exemple, rend invalides toutes les lignes de code qui créaient jusqu'à présent sans problème des instances non initialisées. Si cette "instanciation sans initialisation" doit rester possible, il est alors nécessaire de définir explicitement un constructeur par défaut. Dans le cas contraire, la classe peut rester dépourvue de constructeur par défaut, mais toutes les lignes de code qui utilisaient implicitement celui-ci doivent être modifiées, de façon à ce qu'elles mentionnent désormais une valeur d'initialisation. La définition explicite d'un constructeur acceptant un ou plusieurs arguments permet donc de priver une classe de constructeur par défaut, ce qui revient à exiger que toute instanciation de cette classe comporte une initialisation explicite.

La classe `CDuo` définie ci-dessus est dans ce cas : la présence du constructeur à deux arguments inhibe la génération automatique d'un constructeur par défaut, et une définition telle que

```
CDuo maVariable; //ERREUR : constructeur par défaut non disponible
```

provoquerait une erreur de compilation.

Echec d'un constructeur

Les traitements effectués dans un constructeur peuvent être de toutes natures, et il arrive parfois qu'ils soient légitimement susceptibles d'échouer (du fait de l'épuisement d'une ressource qui leur est nécessaire, par exemple). Ce type de situation n'est pas facile à gérer, car les constructeurs sont souvent invoqués implicitement et ne renvoient de toute façon pas de valeur qui leur permettrait de signaler qu'ils ont rencontré un problème.

Le langage C++ propose un mécanisme dit de "gestion des exceptions" (cf. Leçon 23) qui peut permettre de traiter ce genre de cas. Si ce mécanisme n'est pas utilisé, il faut faire en sorte que les constructeurs susceptibles d'échouer laissent les instances sur lesquelles ils opèrent dans un état qui permet de vérifier facilement si leur construction a été complète.

3 - Listes d'initialisations

Nous avons jusqu'à présent évoqué l'utilisation des constructeurs pour initialiser les variables membre des instances en cours de création. Si cet emploi du mot "initialisation" semble justifié par le fait que le constructeur est, par définition, exécuté avant que l'instance ait pu recevoir quelque valeur que ce soit, il n'en reste pas moins que, dans les exemples précédents, les constructeurs effectuent des opérations qui restent, du point de vue syntaxique, de simples affectations. La distinction est, le plus souvent, sans grande importance pratique. Il existe cependant des circonstances où une affectation s'avère impossible, et c'est notamment le cas lorsque les données membre auxquelles le constructeur est censé conférer une valeur initiale sont des constantes, des références, ou des instances d'une classe ne comportant pas de constructeur par défaut.

Il est, bien entendu, impossible de procéder à l'affectation d'une valeur à une constante ou à une référence, mais le problème n'est pas là : la création d'une constante ou d'une référence exige absolument qu'il y ait une véritable initialisation (au sens syntaxique du terme). La difficulté n'est donc pas que le constructeur n'a pas le droit de procéder à des affectations, mais bien que l'instance ne peut même pas être créée pour être ensuite confiée au constructeur. Le même problème se pose lorsque l'une des variables membre est elle-même une instance d'une classe qui ne comporte pas de constructeur par défaut : la création de ce membre *exige* une initialisation. Rendre l'affectation possible (en définissant un constructeur de transtypage, par exemple) n'est donc d'aucun secours, car la variable membre sur laquelle le constructeur aurait alors le droit de procéder à une affectation ne peut pas être créée.

La solution adoptée par les concepteurs de C++ est de doter les constructeurs d'un moyen de spécifier avec quelles valeurs doivent être initialisées les variables membre de l'instance sur laquelle ils opéreront dès qu'elle aura été créée. La syntaxe employée pour obtenir ce résultat consiste à insérer la spécification des valeurs d'initialisation entre la parenthèse qui clôt la liste des paramètres et l'accolade qui ouvre le corps de la fonction. Cette "liste d'initialisation" débute par le symbole "deux points" et énumère les noms des membres devant être initialisés, suivis chacun d'un couple de parenthèses encadrant la valeur devant être utilisée. Le constructeur par défaut de notre classe élémentaire, que nous avons définie ainsi

```
1 CTresSimple::CTresSimple()  
2 {  
3     leMembre = 0;  
4 }
```

pourrait donc également être définie comme ceci :

```
1 CTresSimple::CTresSimple() : leMembre(0)  
2 {} //l'initialisation est faite, le constructeur n'a plus rien à faire...
```

Comme nous l'avons vu dans la Leçon 3, on choisit souvent de définir les fonctions membre très brèves dans la définition de la classe. La définition de la classe devient alors

```
class CTresSimple  
{  
public :  
    int leMembre;           //déclaration de la variable membre  
    CTresSimple() : leMembre(0) { } //DEFINITION directe du constructeur  
};
```

Bien que les valeurs d'initialisations soient utilisées avant que le constructeur ne soit effectivement appelé, il est tout à fait possible que la liste d'initialisation utilise les valeurs qui seront reçues par le constructeur lors de son appel. Le constructeur de la classe CDuo, que nous avons défini ainsi

```

1 CDuo::CDuo (double unDecimal, int unEntier)
2 {
3   m_decimal = unDecimal;
4   m_entier = unEntier;
5 }

```

pourrait donc être réécrit comme cela :

```

1 CDuo::CDuo (double unDouble, int unInt): m_decimal(unDouble), m_entier(unInt)
2 {}//les initialisations sont faites, le constructeur n'a plus rien à faire...

```

Dans les deux exemples qui précèdent, le recours à une liste d'initialisation relève de la pure coquetterie : il n'y a aucun inconvénient réel à "initialiser" les variables membre à l'aide d'affectations effectuées dans le corps du constructeur. Ce n'est en revanche pas le cas de l'exemple suivant, qui concerne une classe définie ainsi:

```

1 class CMoinsSimple
2 {
3 public:
4   const int m_constante;
5   double & m_ref;
6   CDuo m_duo;
7   CMoinsSimple (int uneConstante, double &uneRef, CDuo unDuo);
8 };

```

Si l'on est conscient du problème posé par la nature des membre de la classe CMoinsSimple, la définition de son constructeur ne pose pas réellement de problème :

```

1 CMoinsSimple::CMoinsSimple(int uneConstante, double &uneRef, CDuo unDuo)
2   : m_constante(uneConstante), m_ref(uneRef), m_duo(unDuo)
3 {}//les initialisations sont faites, le constructeur n'a plus rien à faire...

```

La classe CMoinsSimple peut alors être instanciée, à condition de satisfaire le seul constructeur disponible en procédant à une initialisation explicite :

```

1 double unDouble = 2;
2 CDuo monDuo(1.2, 747);
3 CMoinSimple maVar(18, unDouble, unDuo);

```

Les listes d'initialisation présentent une particularité étrange qui est généralement sans importance mais qui peut parfois créer des problèmes : l'ordre dans lequel les initialisations sont effectuées n'est pas celui spécifié par la liste d'initialisation, mais l'ordre dans lequel les variables membre concernées sont déclarées. Le constructeur par défaut de la classe suivante, par exemple, ne produit pas du tout des instances dans l'état souhaité :

```

class CPiege
{
public:
  int deux;
  int un;
  CPiege::CPiege(): un (1), deux(un + 1){}
};

```

En effet, du fait que le membre deux est déclaré avant le membre un, il est le premier à être initialisé, ce qui signifie que le calcul effectué à cette occasion fait intervenir la valeur de un avant que ce membre n'ait été initialisé... d'où un résultat imprévisible.

4 - Destructeurs

De même que la naissance d'une instance s'accompagne toujours de l'appel d'un constructeur, la "mort" d'une instance provoque l'appel du destructeur de la classe.

Un destructeur est une fonction membre qui porte le même nom que la classe, précédé du signe ~, et est dépourvue de type et dépourvue de paramètre.

L'absence de paramètre exclut la surcharge, puisqu'une seule signature est possible⁵ (celle qui est réduite à la liste vide). Une classe peut donc comporter de nombreux constructeurs, mais elle aura toujours un seul et unique destructeur.

L'appel explicite du destructeur d'une classe est possible, mais n'est nécessaire que dans des cas extrêmement particuliers, qui ne nous intéressent pas pour l'instant.

Attention : le destructeur est automatiquement appelé lorsqu'une instance "meurt", mais ceci ne signifie pas que l'appel du destructeur "tue" l'instance. La mémoire occupée par une variable locale, par exemple, ne sera jamais libérée par l'appel explicite du destructeur. Cet appel provoque l'exécution des instructions définissant le destructeur, mais celles-ci ne peuvent en aucun cas appliquer à une variable un traitement qui la ferait disparaître.

Les opérations effectuées par un destructeur dépendent étroitement de la nature de la classe concernée. Il s'agit le plus souvent d'opérations "symétriques" de celles effectuées lors de la construction : fermeture d'un fichier que le constructeur aurait ouvert, arrêt d'un processus que le constructeur aurait lancé, etc. Les constructeurs générés automatiquement par le compilateur se bornant à initialiser les variables membre, ils ne nécessitent aucune opération "symétrique" au moment de la destruction, ce qui explique que :

Lorsqu'une classe est dépourvue de destructeur, le compilateur lui en ajoute automatiquement un, qui n'effectue aucun traitement.

Avec un destructeur défini explicitement, notre exemple le plus simple devient :

```
1 class CTresSimple
2 {
3 public :
4     int leMembre;
5     ~CTresSimple(); //déclaration du destructeur : pas de type
6                     //                               nom de la classe précédé de ~
7 };
```

et la définition d'une fonction équivalente au destructeur généré automatiquement serait :

```
1 CTresSimple::~~CTresSimple
2 {} //ce destructeur ne fait rien
```

5 - Bon, c'est gentil tout ça, mais ça fait déjà 10 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) La signature d'une fonction est déterminée par sa constance et la liste des types de ses paramètres.
- 2) Deux fonctions peuvent porter le même nom si leurs signatures diffèrent.
- 3) Lorsque deux fonctions sont homonymes, on dit qu'elles sont surchargées.
- 4) Lorsqu'une classe est instanciée, il y a toujours exécution d'un de ses constructeurs.
- 5) Le rôle d'un constructeur **n'est pas** de créer l'objet sur lequel il opère.
- 6) Un constructeur est une fonction membre dépourvue de type et portant le nom de la classe.
- 7) Un constructeur sans arguments est appelé constructeur par défaut.
- 8) Si la définition d'une classe ne mentionne aucun constructeur, le compilateur crée automatiquement un constructeur par défaut.
- 9) Un constructeur par copie est un constructeur recevant comme paramètre une référence à une instance de la classe.
- 10) L'initialisation d'une instance au moyen d'une autre nécessite un constructeur par copie.
- 11) Si, lors d'un appel de fonction, la valeur d'un objet est transmise, le paramètre correspondant ne peut être initialisé que par le constructeur par copie.
- 12) Si une classe a besoin d'un constructeur par copie et qu'elle en est dépourvue, le compilateur en génère automatiquement un.
- 13) Les constructeurs générés par le compilateur s'avèrent parfois insuffisants, et l'on peut être conduit d'une part à créer des versions plus élaborées des constructeurs par défaut et par copie, et d'autre part à créer d'autres types de constructeurs.

- 14) Les listes d'initialisation permettent de procéder à de véritables initialisations des membres de l'instance créée, et non à de simples affectations, ce qui est parfois indispensable.
- 15) Lorsqu'une instance disparaît, il y a toujours exécution du destructeur de la classe.
- 16) Un destructeur est une fonction membre dépourvue de type et de paramètre et dont le nom est composé en faisant précéder celui de la classe du signe ~.
- 17) Le rôle du destructeur **n'est pas** de détruire l'instance au titre de laquelle il est exécuté.