Tutorial C++

Département Image et Traitement de l'Information
ENST de Bretagne

Version 1.2 de Mars 2002

Gwenaël Brunet
Gwenael.Brunet@enst-bretagne.fr
http://perso-iti.enst-bretagne.fr/~brunet/Cours/Tutorial_C++/index.html

Sommaire

- **Avant propos...**
- **Ses premiers pas en Visual C++**
  - Création et lancement d'un projet
  - Compilation et exécution
  - Le debugger
- **Présentation Générale du C++**
  - Généralités
  - La programmation Orientée Objet (POO)
  - Les différences entre C et C++
- **Les bases du C++**
  - Les spécificités de C++
  - Les entrée/sortie C++
- **La notion de classe**
  - Ecriture d'une première classe
  - Utilisation de la classe
  - Constructeur et destructeur
  - Les fonctions membre
- **Construction, destruction et initialisation d'objets**
  - Constructeur par défaut / « initialisant »
  - Constructeur par recopie
  - Mise en œuvre
- **Surdéfinition d'opérateur**
  - Comment ça marche
  - Opérateurs simples
  - Opérateur d'affectation
- **L'héritage**
  - Définition et mise en oeuvre
  - Utilisation des membres de la classe de base
  - Redéfinition des fonctions membre et appel des constructeurs
  - « Statuts » de dérivation
  - Notion d'héritage : élargissement
- **Fonctions virtuelles**
  - Utilité
  - Mécanisme
- **Exercices...**

Avant Propos

Le tutorial qui vous est proposé, comme son nom l'indique, n'a pas pour vocation de vous apprendre à programmer en C++ : il sert de simple « **tremplin** » à la programmation C++. Il donne en outre un aperçu très sommaire des possibilités offertes par ce vaste langage.

De plus, des connaissances en C sont indispensables à une bonne compréhension. Car nous ne reviendrons pas sur des notions telles que les pointeurs.

Les personnes intéressées par ce type de programmation sont par conséquent invitées à consulter des cours/docs plus exhaustifs, comme celui proposé sur le site de référence des programmeurs Visual C++ : Codeguru. Ce cours est celui de Bruce Eckel, et se nomme "Thinking in C++". Suivez le lien !

Je vous recommande aussi tout particulièrement le livre "**Programmer en C++**" de *Claude Delannoy*, *Eyrolles*, dont je me suis beaucoup servi pour ce cours (et avant aussi !).

Ce tutorial est composé de quelques chapitres qui traitent de notions essentielles pour la programmation C++, notamment celles dont vous seriez amené à avoir besoin en utilisant **Visual C++**. Nous verrons donc par exemple comment créer ses propres classes, comment fonctionne la technique de l'héritage en C++, et d'autres spécificités plus ou moins difficiles...

Bon courage !

Premiers pas en Visual C++

- Création et lancement d'un projet
- Compilation et exécution
- Le debugger

Le présent chapitre a pour but de vous apprendre très succinctement, la manipulation de Visual C++, en ce qui concerne la création d'un projet très simple, afin de pouvoir compiler les exemples futurs.
Bien entendu, **si votre compilateur n'est pas Visual C++, vous pouvez passer ce chapitre !**

Création et lancement d'un projet

Visual C++ fonctionne suivant les notions de "**Projet**" et "**Espace de travail**", comme désormais la plupart des environnements de programmation intégrés.
L'espace de travail est l'environnement de développement du programmeur, et est sauvegardé dans un fichier dès sa fermeture, afin d'offrir à l'utilisateur les mêmes conditions lors de la réouverture. Il faut également savoir qu'un "*Workspace*" peut contenir plusieurs projets.

Un *projet* contient des informations plus techniques relatives à la programmation. C'est dans ce fichier que seront stockés le nom des fichiers présents, les librairies utiles, etc. C'est un peu l'équivalent d'un "**makefile**" Unix, sauf qu'il est auto-généré, d'après des renseignements entrés dans une boîte de dialogue.

Pour créer un nouveau projet (qui fera partie automatiquement d'un espace de travail par défaut), vous devez choisir le menu "*File*" puis "*Nouveau*". Attention, le bouton "new" de la barre d'outils n'a pas le même rôle : il permet de créer seulement un nouveau fichier, mais pas de projet.

Vous arrivez alors devant la boîte de dialogue suivante :

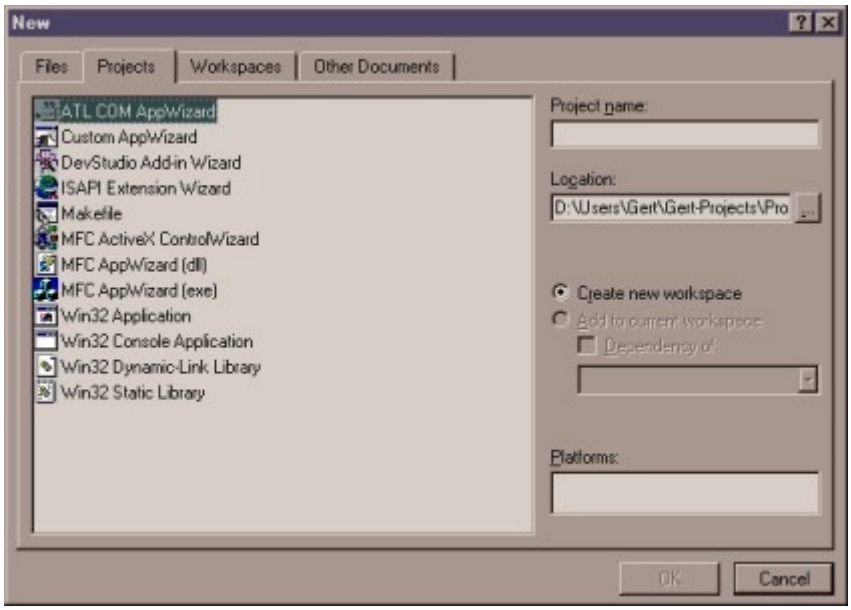


Figure 1 : Création d'un nouveau projet

Vous pouvez remarquer la présence de quatre onglets différents : *Files*, ***Projects***, *Workspaces* et *Other Documents*. Nous verrons dans un prochain cours leur signification respective ; pour l'instant nous allons juste créer un projet pour la compilation de code C++. Pour cela, choisissez l'onglet "*Projects*" si ce n'est pas encore fait, puis cliquez sur le 10ème choix de la liste : "**Win32 Console Application**".
Visual C++ permet en effet d'effectuer de nombreux types de programmes différents, comme par exemple, des librairies, des applications Windows classiques ou encore des programmes qui s'exécutent en mode "**Dos**" (fenêtre Dos). C'est de cette dernière possibilité dont nous avons besoin pour étudier le C++, afin de passer outre les contraintes de programmation d'interfaces.

Vous pouvez alors entrer un nom pour votre projet dans la boîte d'édition "**Project Name**" ("`MyProject`" dans l'exemple) et la localisation physique de votre projet sur le disque dur, dans "**Location**" ("`D:\Temp\`" pour l'exemple).

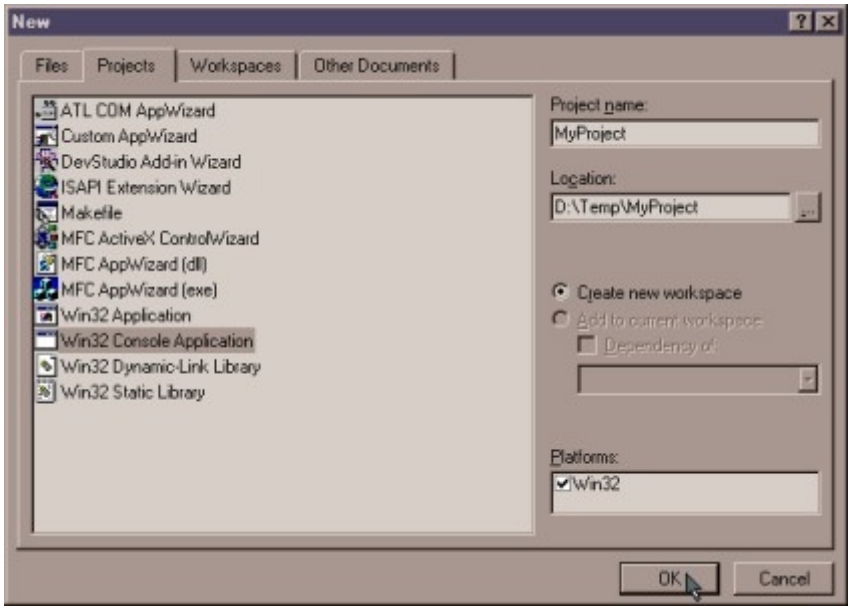


Figure 2 : choix de type de projet

Vous pouvez alors appuyer sur "**Ok**". Visual C++ se charge alors de créer pour vous tout l'environnement de base, nécessaire à la réalisation de votre programme.

Bien sûr, dans notre cas "*d'application console*", il ne vous crée aucun fichier de code : c'est à vous de jouer !

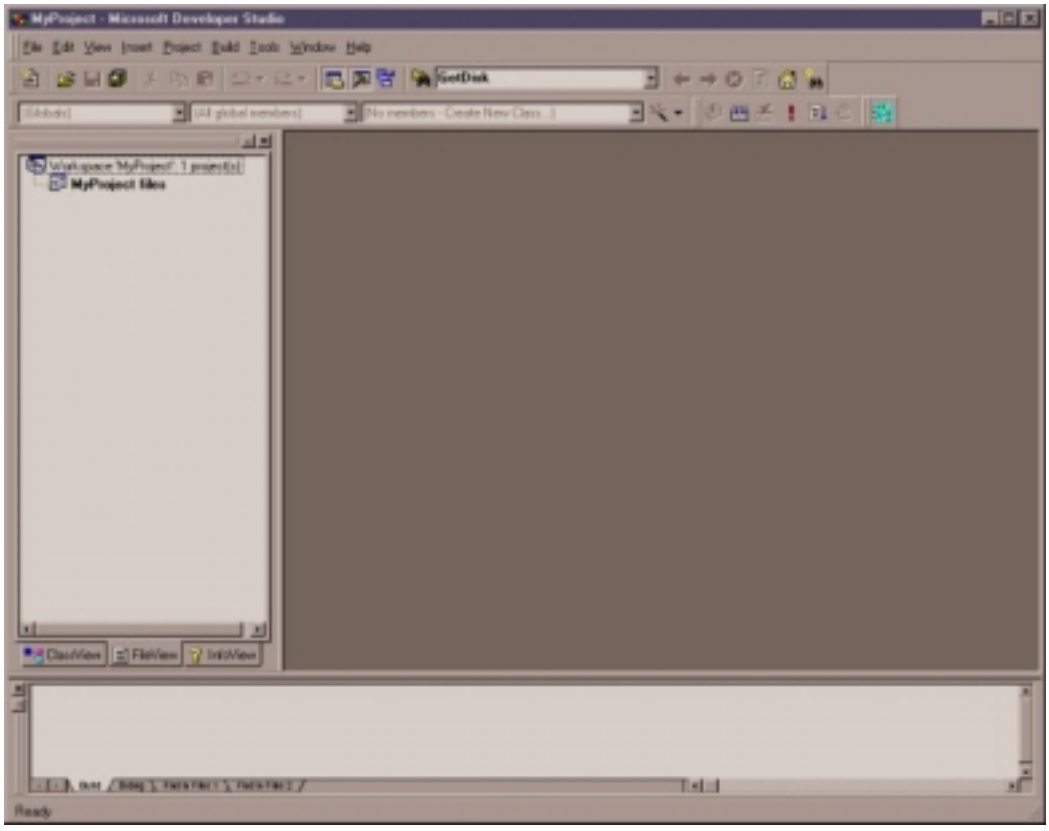


Figure 3 : l'interface de développement Visual C++

Compilation et exécution

Votre projet est créé, il ne vous reste "plus" qu'à l'étoffer avec quelques lignes de code... Pour cela, nous allons commencer par un programme très simple en **langage C**, permettant de voir comment on compile, on exécute et on debugge (même si je me doute que cela ne vous sera jamais utile !).

Tout d'abord, créons un nouveau fichier. Pour cela, vous devez appuyer sur le bouton "*New*" de la barre d'outils (image de gauche).

Vous vous retrouvez alors avec une nouvelle fenêtre ouverte, vide et sans titre (enfin, c'est juste un titre par défaut du style "*Text1*").

La première chose à faire consiste à donner un nom à ce fichier. Pour cela, faites un "*Save*" ou "*Save As*" du menu "*Files*" ou bien cliquez sur le bouton correspondant. Pour notre exemple, appelons le "**main.c**".

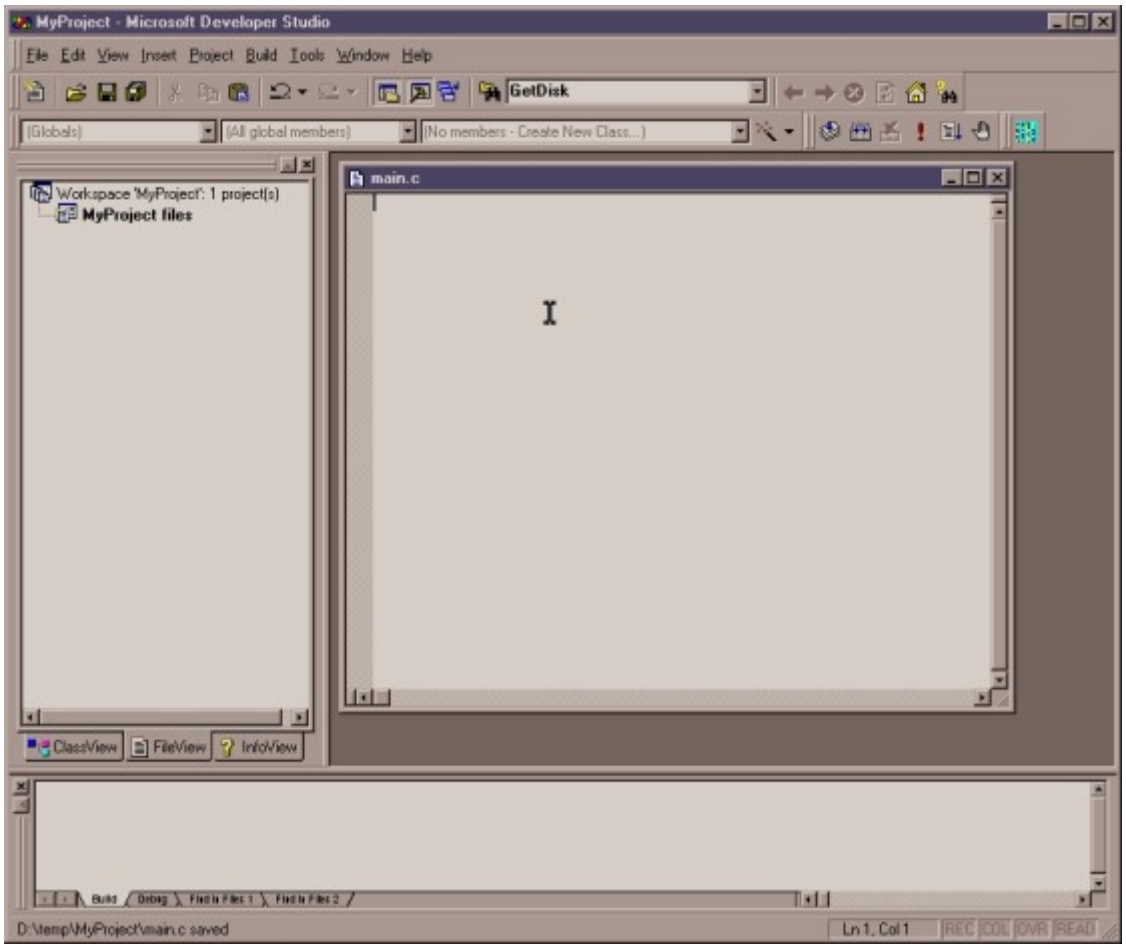


Figure 4 : le fichier "main.c"

Il est maintenant possible d'insérer le dit fichier dans le projet. En effet, vous avez pour l'instant créé un fichier, mais il ne fait même pas partie du

projet : sous Visual C++, vous pouvez charger des fichiers externes au projet sans problème.

Pour cela, le plus simple est de cliquer avec le bouton droit du mulot, sur la fenêtre du fichier, désormais nommée `main.c`. Choisissez alors "**Insert File Into Project**", puis "**MyProject**" (ou le nom de votre projet). Une autre solution, plus longue, passe par le menu "*Project*", puis "*Add To Project*" et enfin "*Files*". Là, vous pouvez choisir votre nouveau fichier fraîchement créé.

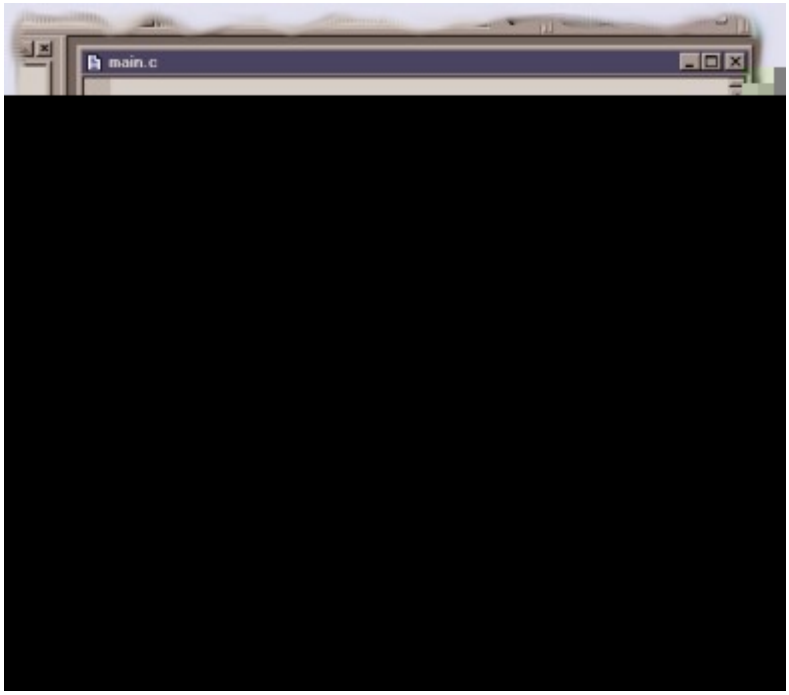


Figure 5 : insérer le fichier dans le projet

Vous avez donc maintenant un fichier "main.c" dans le projet. C'est le seul pour l'instant (partie gauche de l'interface de Visual, onglet "**FileView**").

Il est temps de se mettre enfin à programmer. Nous allons commencer en douceur, avec une application de type "Hello World !", afin de voir rapidement les rouages de l'environnement.

La compilation

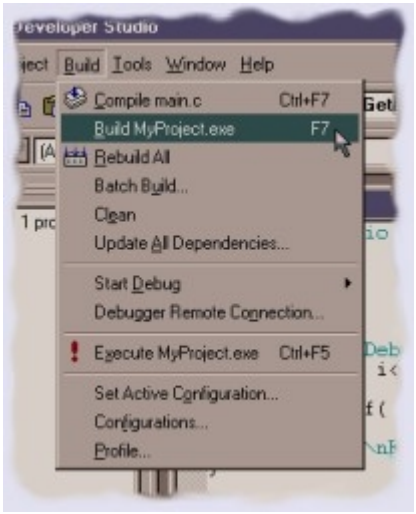
La première tâche consiste à entrer le bout de code suivant dans la fenêtre :

```
#include <stdio.h>

void main()
{
 int i;
```

```
 printf( "Debut boucle\n" );
 for( i=0; i<200; i++ )
 {
  printf( "*" );
 }
 printf( "\nFin boucle\n" );
}
```

Vous aviez été prévenu, ce n'est pas de la haute voltige. Maintenant que le code a été entré, il faut le compiler.



Il existe deux compilations différentes : soit vous compilez le fichier courant seul ("**Maj+F7**") ou bien le projet dans sa globalité ("**F7**"). C'est la deuxième qui permet la génération d'un exécutable, bien entendu.

Il est également toujours possible d'accéder à ces actions, par l'intermédiaire du menu "**Build**" de l'interface, ou encore par le bouton correspondant.

L'exécution

L'exécution est elle aussi très simple. A partir du moment où vous avez compilé le projet, vous pouvez l'exécuter en sélectionnant "**Execute MyProject.exe**" (**Ctrl+F5**).
Une fenêtre Dos s'affiche alors : elle contient les sorties du programme que vous venez décrire.

Le debugger

Par défaut, Visual C++ compile les projets en mode "**Debug**". Cela signifie que des informations de debuggage sont incluses dans l'exécutable, afin d'aider le programmeur à retrouver certaines erreurs. En effet, le programme est alors **traçable**, c'est-à-dire qu'à tout moment durant l'exécution, il est possible de savoir où on se trouve dans le code.
De même, lors d'un plantage, le debugger pourra rendre la main si on le désire, exactement à l'endroit où l'erreur a eu lieu. Ceci simplifie bien évidemment grandement la tâche du développeur.

Tracer un programme

Comme il a été signalé précédemment, il est possible de "tracer" un programme. Cela consiste à l'exécuter ligne par ligne, expression après expression. Pour ce faire, il y a plusieurs solutions.

La première fonctionne suivant la technique des "**Breakpoints**" : on place un point d'arrêt quelque part dans le code (**Ctrl+B**, cf. *Figure 6*), à une ligne donnée (en fait, la ligne courante du curseur de texte).

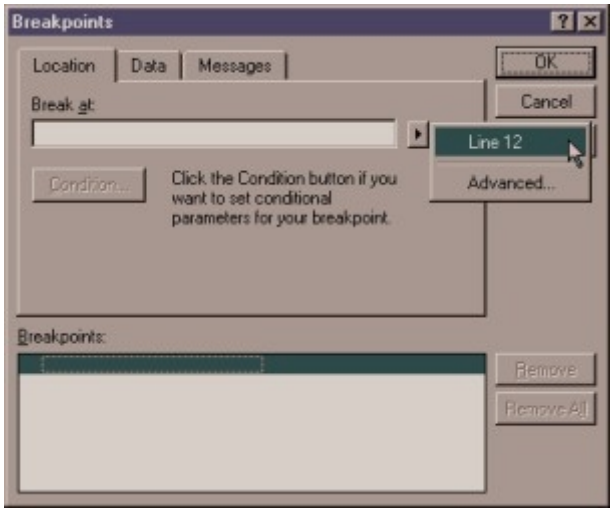


Figure 6 : ajouter un breakpoint dans le code

On obtient un petit rond rouge sur la gauche de la ligne de code, qui signifie qu'un breakpoint est présent (*Figure 7*).

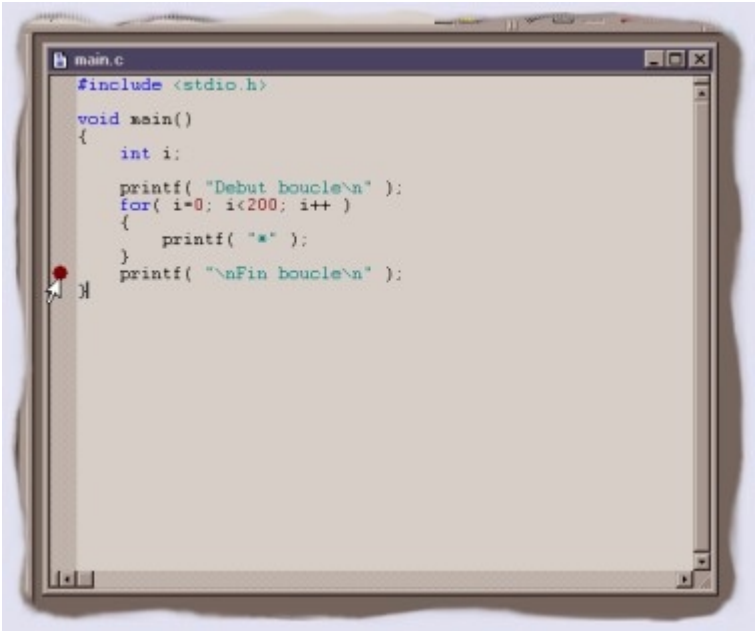


Figure 7 : un breakpoint mis en place

On lance alors l'exécution en mode Debug (**F5**), et le programme s'arrête automatiquement à l'endroit voulu. Il est alors possible de tracer le programme ligne par ligne (**F10** ou **F11**), ou bien de le relancer jusqu'à la fin ou le prochain point d'arrêt.

Une seconde solution consiste à ne pas mettre de breakpoint, mais à exécuter un programme jusqu'à une ligne donnée. Cette possibilité est offerte par Visual grâce au menu "**Build**", "**Start Debug**", puis "**Run To Cursor**" (**Ctrl+F10**).

C'est personnellement la méthode que j'utilise, puisqu'elle est très simple et surtout rapide à mettre en oeuvre.

Essayez les deux !

Les données

C'est bien joli de pouvoir tracer un programme, mais il est peut-être plus intéressant encore de pouvoir jeter un oeil sur l'état des données (variables) du programme.

Pour cela, vous pouvez vous servir de la barre qui s'affiche en bas de l'interface de Visual en mode debuggage (*Figure 8*). La partie de droite est notamment très utile pour suivre l'évolution d'une variable lorsque vous tracez un programme. Pour rajouter une variable dans cette partie, il suffit d'effectuer un *glisser-déposer* de cette dernière.

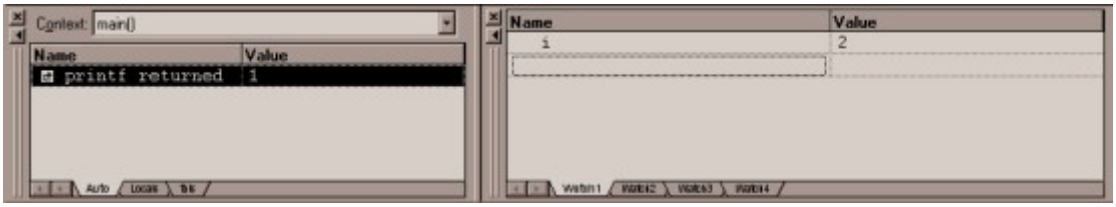


Figure 8 : à gauche, les variables et fonctions en cours d'utilisation, à droite, les variables que l'utilisateur souhaite examiner

Vous pouvez également avoir accès au contenu d'une variable en cliquant sur elle à l'aide du bouton droit de la souris, puis "**Quickwatch**".

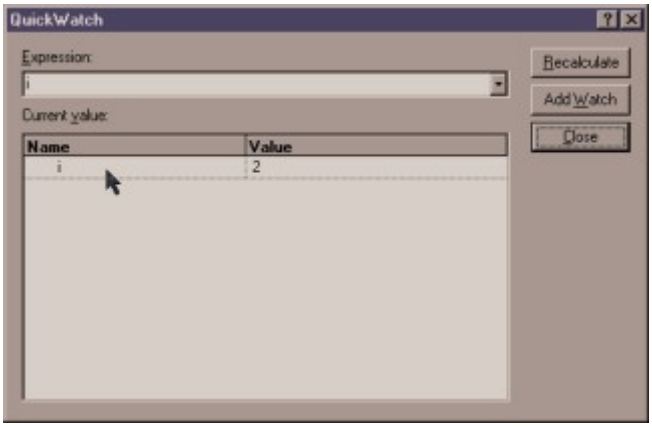


Figure 9 : Quickwatch d'une variable (ici, "i")

Vous savez désormais créer un nouveau projet, y inclure de nouveaux fichiers, le compiler, l'exécuter et enfin le debugger. Il est temps de passer aux choses sérieuses : le **langage C++** en lui-même...

Attention ! : *désormais, nous allons faire du C++, l'extension du fichier contenant le "**main**", ainsi que de tout autre fichier d'implémentation, doit être "**cpp**" (pour l'instant, c'était "**c**").*
*De plus, si vous ne changez pas de projet, veillez à retirer votre fichier C du projet, en effectuant un "**Suppr**" dans l'onglet "**FileView**" de Visual C++.*

Présentation Générale

- Généralités
- La programmation Orientée Objet (POO)
- Les différences entre C et C++

Généralités

Puisque ça n'est pas foncièrement utile, je vous fais grâce de l'historique du langage C++. Vous devez juste savoir, éventuellement, qu'il a été conçu par **Bjarne Stroustrup**, ce qui aide pour trouver l'excellente bible du C++, écrite par lui-même.

Pour commencer cette présentation, parlons de quelques généralités, et d'abord du *pourquoi du C++*.

C'est avant tout une nécessité de répondre à des besoins générés par de gros projets. Ils nécessitent une façon de travailler plus rigoureuse, pour un code plus structuré, extensible, réutilisable et enfin si possible, portable. Ceci est assez limité lorsqu'on emploie un langage simplement structuré tel que **C** ou **Turbo Pascal**.

Programmation Orientée Objet

La **Programmation Orientée Objet** (P.O.O.) est une solution. Elle permet d'introduire le concept d'**objet** justement, qui consiste en un ensemble de données et de procédures qui agissent sur ces données.

Lorsque l'objet est parfaitement bien écrit, il introduit la notion fondamentale d'**Encapsulation des données**. Ceci signifie qu'il n'est plus possible pour l'utilisateur de l'objet, d'accéder directement aux données : il doit passer par des méthodes spécifiques écrites par le concepteur de l'objet, et qui servent d'*interface* entre l'objet et ses utilisateurs. L'intérêt de cette technique est évident. L'utilisateur ne peut pas intervenir directement sur l'objet, ce qui diminue les risques d'erreur, ce dernier devenant une "*boîte noire*".

Une autre notion importante en P.O.O. est l'**héritage**. Elle permet la définition d'une nouvelle classe à partir d'une classe existante. Il est alors possible de lui adjoindre de nouvelles données, de nouvelles fonctions membres (procédures) pour la spécialiser.

Différences entre C et C++

Nous allons parler ici d'un certain nombre de différences existant entre le C et le C++. Nous pourrions d'ailleurs plutôt utiliser le terme d'incompatibilités.

Les fonctions

Les fonctions en C peuvent être définies suivant deux modèles :

```
int CalculeSomme ( a, b )
int a;
int b;
{
     ... /* Fonction */
}
```

```
int CalculeSomme ( int a, int b )
{
     ... /* Fonction */
}
```

Il faut simplement savoir que le C++ n'accepte que la seconde méthode.

Const

Le C++ a quelque peu modifié l'utilisation "*C*" de ce qualificatif. Pour rappel, "**const**" est utilisé pour définir une variable constante. C'est une bonne alternative à un **define**.

La portée en C++ est désormais plus locale. En C, un const permettait pour une variable globale d'être "visible" partout. C++ limite quant à lui la portée d'une telle variable, au fichier source contenant la déclaration.

Compatibilité de pointeurs

En C ANSI, un "**void***" est compatible avec tout autre type de pointeurs, et inversement.
Par exemple, ceci est légal en C :

```
/* C */
void * pQqch;    /* Pointeur générique */
int * pEntier;    /* Pointeur sur un entier */
pEntier = pQqch;
pQqch = pEntier;
```

Ces affectations font intervenir des conversions implicites. En C++, seule la conversion *int*->void** est implicite. L'autre reste possible, mais nécessite un "**cast**" :

```
// C++
void * pQqch;           // Pointeur générique
int * pEntier;          // Pointeur sur un entier
pEntier = (int*)pQqch;  // "cast" en entier
```

Les bases du C++

- Les spécificités de C++
- Les entrée/sortie C++

Les spécificités de C++

Le langage C++ a adopté un certain nombre de nouvelles spécificités qu'il faut connaître avant de se lancer dans la P.O.O. proprement dite.

Les commentaires

Les commentaires d'un code source peuvent maintenant être indiqués de deux façons différentes :

```
/* Ceci est commentaire "typique" venant du C */
int nEntier; // Et ceci est la seconde possibilité
```

Ces nouveaux commentaites sont utilisables uniquement dans le cas où tout le reste de la ligne est un commentaire.

Déclarations

En C, vous avez été habitué à déclarer les variables en début de bloc, c'est-à-dire en début de fonction ou de procédure. En C++, il est possible de déclarer une variable à tout moment dans le code.

```
/* Code C */
int FaitQqch( int a, int b)
{
     int nRetour;
     int i;
     int fVar;

     fVar = a + b;
     for( i=0; i<20; i++ )
     {
          fVar = fVar + i;
     }

     nRetour = fVar - a*b;
     return nRetour;
}
```

```
// Code C++
int FaitQqch( int a, int b)
{
     int fVar = a + b;

     for( int i=0; i<20; i++ )
     {
          fVar = fVar + i;
     }

     int nRetour = fVar - a*b;
     return nRetour;
}
```

Cet exemple est bien entendu dénué de tout intérêt, mais il montre la liberté offerte par C++ en ce qui concerne les déclarations et les initialisations des variables.

Référence

C++ introduit une nouvelle notion fondamentale : les **références**. C'est une notion qui peut sembler difficile à assimiler au départ, notamment pour des personnes qui ne sont pas encore habituées à utiliser des pointeurs en C.

La notion de référence est directement liée au passage de paramètres à des fonctions en C. Nous savons tous que lorsque nous voulons transmettre à une fonction la valeur d'une variable ou au contraire la donnée réelle (en fait l'adresse), nous n'utilisons pas les mêmes méthodes.
Par exemple, soit les fonctions suivantes :

```
int FaitQqch( int a, int b)
{
     int nRet;
     if( a==0 )
          a=10;
     nRet = a + b;

     return nRet;
}
```

```
int FaitQqch2( int * a, int * b)
{
     int nRet;
     if( *a==0 )
          *a=10;
     nRet = *a + *b;

     return nRet;
}
```

Nous voyons tout de suite que dans le cas d'un appel comme celui-ci :

```
...
int a, b, c, d;
a = 0;
b = 5;
c = FaitQqch(a,b);
d = FaitQqch2(&a,&b);
...
```

c et **d** auront la même valeur, par contre, ce qui est intéressant, c'est qu'à la sortie de `FaitQqch`, la valeur restera inchangée, alors que pour `FaitQqch2`, `a` vaudra désormais `10` ! Ceci est dû au passage par adresse, et non par valeur.

Tout cela, vous devez le savoir. En revanche, vous allez apprendre une nouvelle technique qui est une sorte de mélange des deux précédentes : les **références**.

Voici ce que devient notre fonction, `FaitQqch3` :

```
int FaitQqch3(int &a, int &b)
{
     int nRet;
     if( a==0 )
          a=10;
     nRet = a + b;

     return nRet;
}
```

Le "**&**" signifie que l'on passe par référence. La ligne de déclaration de la fonction est en fait la seule différence avec une transmission par valeur, d'un point de vue code. C'est-à-dire que l'utilisation des variables dans la

fonction s'opère sans "*****" et l'appel à la fonction sans "**&**". C'est ce qui fait la puissance des références : c'est transparent pour l'implémentation, mais cela possède la puissance des pointeurs.

Nous nous habituerons à leur utilisation au fur et à mesure.

Opérateurs new et delete

En plus des "anciens" **malloc** et **free** du C, C++ possède un nouveau jeu d'opérateurs d'allocation/désallocation de mémoire : **new** et **delete**. Ils ont été créés principalement pour la gestion dynamique des objets, mais on peut les utiliser également pour des variables simples.
Voici une comparaison d'utilisation :

```
...
/* pour un simple pointeur */
int * pInt;
pInt = (int*)malloc(1*sizeof(int));
free(pInt);
...
/* pour un tableau */
pInt = (int*)malloc(100*sizeof(int));
free(pInt);
...
```

```
...
// pour un simple pointeur
int * pInt;
pInt = new int;
delete pInt;
...
// pour un tableau
pInt = new int[100];
delete pInt;
...
// Tableau de classes
pToto = new MyClass[50];
delete []MyClass;
```

Vous remarquerez donc tout de suite les différences, qui sont évidentes. Insistons simplement sur la désallocation à l'aide de l'opérateur **delete**, qui change suivant que le pointeur est simple, ou bien qu'il correspond à un tableau lorsqu'il est composé d'objets.

Les entrées/sorties en C++

Le langage C++ dispose de nouvelles routines d'entrées/sorties qui sont plus simples à utiliser.

La sortie standard "cout" :

```
// include indispensable pour cout
#include <iostream.h>

void main()
{
     cout << "Hello World !";
     cout << "Hello World !\n";
     cout << "Hello World !" << endl;

     int n = 5;
     cout << "La valeur est " << n << endl;

     float f = 3.14f;
     char *ch = "Coucou";
```

```
     cout << ch << " float = " << f << endl;
}
```

Ce programme donne en sortie :

```
Hello World !Hello World !
Hello World !
La valeur est 5
Coucou float = 3.14
```

Cette nouvelle sortie standard est donc très intuitive à employer du fait qu'il est inutile de lui préciser le format de la valeur que l'on souhaite afficher.
Le "**endl**" est en fait disponible pour éviter d'éventuels "\n", en fin de ligne.

L'entrée standard "cin" :

```
// include indispensable pour cout et cin
#include <iostream.h>

void main()
{
     int n;
     cout << "Entrez un entier : ";
     cin >> n;
     cout << "Vous avez entré : " << n << endl;

     char ch[81];
     float f;
     cout << "Entrez un entier, une chaine, puis un float :";
     cin >> n >> ch >> f;
     cout << "Vous avez entré : " << n << ch << f << endl;
}
```

Cet exemple illustre brièvement comment fonctionne "**cin**". Bien entendu, aucun contrôle de type n'est effectué, c'est donc à l'utilisateur qu'il advient de faire attention.

La notion de Classe

- Ecriture d'une première classe
- Utilisation de la classe
- Constructeur et destructeur
- Les fonctions membre

Nous allons enfin parler, dans ce chapitre, de **Programmation Orientée Objet**. Nous allons commencer par comprendre le mécanisme des **classes**.

Ecriture d'une première classe

Une classe est en quelque sorte une structure complexe qui permet **l'encapsulation de données**.
Une classe est composée de données et de méthodes. Lorsque l'encapsulation des données est parfaite, seules les (certaines) méthodes sont accessibles. Ceci évite en principe à l'utilisateur de la classe, de se soucier de son fonctionnement et de faire des erreurs en changeant directement la valeur de certaines données.
Prenons un exemple concret et simple : l'écriture d'une classe **Point**. Cet exemple va nous suivre tout au long de ce chapitre.

En **C**, nous aurions fait une structure comme suit :

```
struct Point
{
     int x;     // Abscisse du point
     int y;     // Ordonnée
};
```

La déclaration précédente fonctionne parfaitement en C++. Mais nous aimerions rajouter des fonctions qui sont fortement liées à ces données, comme l'affichage d'un point, son déplacement, etc. Voici une solution en **C++** :

```
class Point
{
public :
     int x;
     int y;

     void Init(int, int); // Initialisation d'un point
     void Deplace(int, int); // Déplacement du point
     void Affiche();          // Affichage du point
};
```

Vous remarquerez tout de suite plusieurs éléments :

• **`class`** : le "`struct`" a été remplacé, même si dans cet exemple précis, il aurait pu être conservé. Mais nous ne rentrerons pas dans les détails.

- **public** : le terme public signifie que tous les membres qui suivent (données comme méthodes) sont accessibles de l'extérieur de la classe. Nous verrons les différentes possibilités plus tard.
- L'ajout des fonctions (ou plutôt méthodes puisqu'elles font partie de la classe) "**Init**", "**Deplace**" et "**Affiche**". Elles permettent respectivement d'initialiser un point, de le déplacer (addition de coordonnées) et de l'afficher (contenu des variables x et y).

Utilisation de la classe

Voici maintenant un programme complet pour mettre en application tout ceci :

```
#include <iostream.h>

class Point
{
public :
     int x;
     int y;

     void Init(int a, int b){ x = a; y = b; }
     void Deplace(int a, int b){ x += a; y += b; }
     void Affiche(){ cout << x << ", " << y << endl; }
};

void main()
{
     Point p;
     p.Init(3,4);
     p.Affiche();
     p.Deplace(4,6);
     p.Affiche();
}
```

Les méthodes de la classe *Point* sont implémentées dans la classe même. Ceci fonctionne très bien, mais devient bien entendu assez lourd lorsque le code est plus long. C'est pourquoi il vaut mieux placer la déclaration seulement, au sein de la classe. Notre code devient alors :

```
#include <iostream.h>

class Point
{
public :
     int x;
     int y;

     void Init(int a, int b);
     void Deplace(int a, int b);
     void Affiche();
};

void Point::Init(int a, int b)
{
```

```
	x = a;
	y = b;
}
void Point::Deplace(int a, int b)
{
	x += a;
	y += b;
}
void Point::Affiche()
{
	cout << x << ", " << y << endl;
}

void main()
{
	Point p;
	p.Init(3,4);
	p.Affiche();
	p.Deplace(4,6);
	p.Affiche();
}
```

Vous avez remarqué la présence du "**`Point::`**" qui signifie que la fonction est en faite une méthode de la classe *Point*. Le reste est complètement identique. La seule différence entre ces 2 programmes vient du fait qu'on dit que les méthodes du premier programme (dont l'implémentation est faite dans la classe), sont "**inline**". Ceci signifie que chaque appel à la méthode sera remplacé dans l'exécutable, par la méthode en elle-même (un peu comme une **macro** en C). D'où un gain de temps certain, mais une augmentation de la taille du fichier en sortie.

Mais la différence entre une structure et une classe n'a pas encore été vraiment détaillée. En effet, vous pourriez très bien compiler le même source en retirant le terme "`public`" et en remplaçant "`class`" par "`struct`".
En fait, l'intérêt réel du C++ tourne autour de cette notion importante d'encapsulation de données. Dans l'exemple de la classe *Point*, nous n'avons pour l'instant spécifier aucune protection de données ; vous pouvez rajouter ces quelques lignes, sans erreur de compilation :

```
	...
	p.x = 25; // Accès aux variables de la classe point
	p.y = p.x + 10;
	cout << "le point est en " << p.x << ", " << p.y << endl;
	...
```

L'encapsulation a pour objet d'empêcher cela, afin de notamment limiter la nécessité de compréhension d'un objet pour l'utilisateur. La classe devient alors une espèce de "*boîte noire*" avec des interfaces d'entrée et de sortie. D'où la déclaration suivante :

```
class Point
{
```

```
private :
     int x;
     int y;

public :
     void Init(int a, int b);
     void Deplace(int a, int b);
     void Affiche();
};
```

Il est alors interdit d'accéder aux variables x et y qui sont des membres "**privés**", en dehors de la classe *Point* (elles restent accessibles dans les méthodes de Point !).

Il est également possible d'effectuer une affection de classe, comme pour une structure C. Ceci a le même effet puisque l'affectation a lieu sur les données membre :

```
#include <iostream.h>

... // déclaration de la classe point

void main()
{
     Point p;
     p.Init(3,4);
     p.Affiche();

     Point p2;
     p2 = p;
     p2.Affiche();
}
```

Constructeur et Destructeur

Au cours de l'élaboration de cet exemple aussi simple que concret, vous vous êtes peut-être dit qu'il serait intéressant d'initialiser l'objet au moment de sa déclaration. En effet,il faut de toute façon généralement utiliser tout de suite après la méthode "Init", alors pourquoi ne pas faire d'une pierre deux coups ! Ceci est bien entendu possible en C++ : il s'agit des constructeurs.
Voici comment nous pouvons mettre en oeuvre un constructeur :

```
class Point
{
     // Ici le "private" est optionnel dans la mesure où tout
     // ce qui suit la première accolade est privé par défaut
     int x;
     int y;

public :
     Point(int, int); // Constructeur de la classe point
     void Init(int a, int b);
     void Deplace(int a, int b);
```

```
     void Affiche();
};
```

Vous vous demandez sûrement comment déclarer désormais, une variable de type *Point* ! Vous pensez peut-être pouvoir faire ceci :

```
Point p;
```

En fait, non. A partir du moment où un constructeur est défini, il doit pouvoir être appelé par défaut pour la création de n'importe quel objet. Dans notre cas il faut par conséquent préciser les paramètres, par exemple :

```
Point p(4,5);
```

Pour laisser plus de liberté, et permettre une déclaration sans initialisation, il faut prévoir un **constructeur par défaut** :

```
class Point
{
     int x;
     int y;

public :
     Point();  // Constructeur par défaut
     Point(int, int);
     void Init(int a, int b);
     void Deplace(int a, int b);
     void Affiche();
};
```

Tout comme il existe un constructeur, on peut spécifier un destructeur. Ce dernier est appelé lors de la destruction de l'objet, explicite ou non.

```
class Point
{
     int x;
     int y;

public :
     Point();
     Point(int, int);
     ~Point(); // Destructeur de la classe Point
     void Init(int a, int b);
     void Deplace(int a, int b);
     void Affiche();
};
```

Dans notre cas de classe *Point*,le destructeur a peu d'utilité. On pourrait à la rigueur placer une instruction permettant de tracer la destruction. En revanche, lorsqu'une classe possède par exemple des pointeurs comme

données membre, il est possible de désallouer la mémoire à cet endroit. Un exemple :

```
#include <iostream.h>

class Test
{
     int nSize;

public :
     // pas d'encapsulation pour ce membre.
     // Ca n'est pas très bon, mais c'est juste pour l'exemple.
     int *pArray;

     Test(int n);
     ~Test();
     int GetSize(){ return nSize; }
};
Test::Test(int n)
{
     cout << "-- Constructeur --" << endl;
     nSize = n;
     pArray = new int[nSize];
}
Test::~Test()
{
     cout << "-- Destructeur --" << endl;
     if( pArray )
       delete []pArray;
}

void main()
{
     Test t(5);
     for( int i=0; i<t.GetSize(); i++ )
     {
          t.pArray[i]=i;
     }

     Test *t2; // Pointeur d'objet
     t2 = new Test(10);  // Allocation dynamique
     for( i=0; i<t2->GetSize(); i++ )
     {
          t2->pArray[i]=i;
     }
     delete t2; // Destruction explicite
}
```

Les Fonctions membre

Surdéfinition

Nous avons vu dans le chapitre précédent qu'il était possible de définir plusieurs constructeurs différents. Nous pouvons étendre cette possibilité de *surdéfinition* à d'autres méthodes que le constructeur (sauf le destructeur !) :

```
class Point
{
     int x;
     int y;

public :
     Point();
     Point(int, int);
     ~Point();
     void Init(int a, int b);
     void Init(int a); // Initialisation avec une même valeur
     void Deplace(int a, int b);
     void Deplace(int a);
     void Affiche();
     void Affiche(char* strMesg); // Affichage avec un message
};
```

Arguments par défaut

Tout comme une fonction C++ classique, il est possible de définir des arguments par défaut. Ceux-ci permettent à l'utilisateur de ne pas renseigner certains paramètres. Par exemple, imaginons que l'initialisation par défaut d'un point soit `(0,0)`. Nous pouvons donc changer la méthode **`Init`**, de sorte qu'elle devienne :

```
void Init(int a=0);
```

Désormais, quand l'utilisateur appelle cette méthode, il a la possibilité de ne pas donner de paramètre, signifiant qu'il veut initialiser son point à 0. De même :

```
void Affiche(char* strMesg="");
```

Permet de remplacer l'implémentation de deux méthodes par une seule, mais qui prend en compte le non-renseignement du paramètre. Notre programme devient donc :

```
#include <iostream.h>

class Point
{
     int x;
     int y;
```

```
public :
     Point();
     Point(int, int);
     ~Point();
     void Init(int a, int b);
     void Init(int a=0);
     void Deplace(int a, int b);
     void Deplace(int a=0);
     void Affiche(char* strMesg="");
};

Point::Point()
{
     cout << "--Constructeur par defaut--" << endl;
}
Point::Point(int a, int b)
{
     cout << "--Constructeur (a,b)--" << endl;
     Init(a,b);
}
Point::~Point()
{
     cout << "--Destructeur--" << endl;
}
void Point::Init(int a, int b)
{
     x = a;
     y = b;
}
void Point::Init(int a)
{
     Init(a,a);
}
void Point::Deplace(int a, int b)
{
     x += a;
     y += b;
}
void Point::Deplace(int a)
{
     Deplace(a,a);
}
void Point::Affiche(char *strMesg) // On ne rajoute pas le paramètre par
                                   // défaut dans l'implémentation !
{
     cout << strMesg << x << ", " << y << endl;
}

void main()
{
     Point p(1,2);
     p.Deplace(4);
     p.Affiche("Le point vaut ");
     p.Init(10);
     p.Affiche("Le point vaut desormais : ");

     Point pp;
     pp = p;
     p.Deplace(12,13);
     pp.Deplace(5);
```

```
     p.Affiche("Le point p vaut ");
     pp.Affiche("Le point pp vaut ");
}
```

Vous commencez à avoir un programme un peu plus long...

Objets transmis en argument d'une fonction membre

Nous pouvons maintenant imaginer vouloir comparer deux points, afin de savoir s'ils sont égaux. Pour cela, nous allons mettre en oeuvre une méthode "Coincide" qui renvoie "1" lorsque les coordonnées des deux points sont égales :

```
class Point
{
     int x;
     int y;

public :
     Point(int a, int b){ x=a; y=b; }
     int Coincide(Point p);
};
int Point::Coincide(Point p)
{
     if( (p.x==x) && (p.y==y) )
       return 1;
     else
       return 0;
}
```

Cette partie de programme fonctionne parfaitement, mais elle possède un inconvénient majeur : la **passage de paramètre par valeur**, ce qui implique une "duplication" de l'objet d'origine. Cela n'est bien sûr pas très efficace.
La solution qui vous vient à l'esprit dans un premier temps est probablement de passer par un pointeur. Cette solution est possible, mais n'est pas la meilleure, dans la mesure où nous savons fort bien que ces pointeurs sont toujours sources d'erreurs (lorsqu'ils sont non initialisés, par exemple).

La vraie solution offerte par le C++ est de passer par des références. Avec ce type de passage de paramètre, aucune erreur est possible puisque l'objet à passer doit déjà exister (être instancié). En plus, les références offrent une simplification d'écriture, par rapport aux pointeurs :

```
#include <iostream.h>

class Point
{
     int x;
     int y;
```

```
public :
     Point(int a=0, int b=0){ x=a; y=b; }
     int Coincide(Point &);
};
int Point::Coincide(Point & p)
{
     if( (p.x==x) && (p.y==y) )
       return 1;
     else
       return 0;
}

void main()
{
     Point p(2,0);
     Point pp(2);
     if( p.Coincide(pp) )
          cout << "p et pp coincident !" << endl;
     if( pp.Coincide(p) )
          cout << "pp et p coincident !" << endl;
}
```

Construction, destruction et initialisation d'objets

- Constructeur par défaut / "initialisant"
- Constructeur par recopie
- Mise en œuvre

Constructeur par défaut / initialisant

Constructeurs simples

Nous avons abordé précédemment la possibilité d'initialiser un objet lors de sa construction, ou encore de proposer un constructeur par défaut. Nous allons dans ce chapitre généraliser le principe tout en l'approfondissant.

Tout d'abord, il faut bien comprendre l'intérêt de la chose : un constructeur initialisant permet comme son nom l'indique d'initialiser les données membre, que ces valeurs soient entrées par l'utilisateur, ou bien qu'elles soient par défaut. Le destructeur quant à lui est appelé à la fin de la vie de l'objet, de façon automatique ou non (si l'objet est un pointeur). Ceci prend toute son importance dans le cas d'un objet qui contient lui-même des données de type pointeur. Prenons par exemple une classe **Vecteur** qui gère un vecteur mémoire d'entiers. Nous allons essayer d'intégrer dans un premier temps :

- *un **constructeur initialisant**, connaissant la **taille** du vecteur,*
- *un **destructeur**,*
- *une méthode de **lecture** d'une valeur dans le vecteur,*
- *---------- d'**écriture** ----------,*
- *une méthode qui renvoie la **taille** du vecteur.*

Dans la mesure où vous devriez être capable de réaliser cette classe, essayez de la concevoir tout seul dans un premier temps...

Solution

```
#include <iostream.h>

class Vecteur
{
     int *pVecteur;
     int nTaille;

public :
     Vecteur(int); // Construction connaissant la taille du vecteur
     ~Vecteur();   // Destruction

     int GetAt(int ind)       // Lecture de valeur en "inline"
     {
          if( ind>=0 && ind<nTaille )
            return pVecteur[ind];
          else
          // comportement indéfini, pour l'instant retour 0;
            return 0;
     }
     void SetAt(int ind, int val) // Ecriture de valeur
     {
          if( ind>=0 && ind<nTaille )
               pVecteur[ind]=val;
     }
     int Size(){ return nTaille; }
};

Vecteur::Vecteur(int Taille)
{
     // Aucun contrôle de taille n'est fait pour simplifier
     nTaille = Taille;
     pVecteur = new int[nTaille];
}
Vecteur::~Vecteur()
{
     delete pVecteur;
}

void main()
{
     Vecteur *v;
     v = new Vecteur(10);
     for( int i=0; i<v->Size(); i++ )
          v->SetAt(i, i*i-i);
     for( i=0; i<v->Size(); i++ )
          cout << v->GetAt(i) << " ";
     cout << endl;
     delete v;
}
```

Le pourquoi du comment

Il faut bien comprendre le rôle du destructeur. Un destructeur de classe est appelé à la destruction de l'objet, c'est-à-dire à la fin du bloc dans lequel il a été instancié, ou bien à un endroit spécifié par l'utilisateur, si l'objet a été créé dynamiquement (appel à un `delete` à ce moment là). Mais le destructeur détruit *seulement* l'objet, soit la mémoire allouée pour les membres. En outre, il ne peut pas savoir que la classe est composée de pointeurs et qu'une allocation mémoire a été effectuée vers tel segment mémoire. C'est au créateur de la classe qu'il advient de spécifier la désallocation des pointeurs (tout comme l'allocation d'ailleurs).

Quant au *main*, il est vraiment très simple. Il se propose juste de créer un objet de type *Vecteur*, dynamiquement, de le remplir avec différentes valeurs, puis de les afficher.

Pour comprendre un peu le fonctionnement de la classe, essayez d'autres opérations telles que différentes créations d'objets statiquement et dynamiquement.

Constructeur par recopie

Reprenons notre classe *Point*. Nous avons donc déjà deux constructeurs, à savoir un par défaut et un autre par initialisation des coordonnées.
Il apparaît qu'il pourrait être intéressant de réaliser un constructeur à partir d'un point !
C++ permet cette fonctionnalité par défaut, en voici l'exemple :

```
// Définition de la classe Point
...
     Point pt(1,2);
     Point pt2(pt); // Construction par recopie de Point
...
```

Il est possible de redéfinir ce constructeur. Bien entendu, il faut qu'il est un intérêt quelconque.

```
class Point
{
     int x;
     int y;

public :
     Point(){x=-1;y=-1;}
     Point(int a, int b){ x=a; y=b; }
     Point(const Point & pt) // Constructeur par recopie de Point
     {
          x = pt.x;
          y = pt.y;
     }
     ... // D'éventuelles autres méthodes
};
```

Ajouter un constructeur par recopie est donc assez simple dans l'esprit. En ce qui concerne la syntaxe, vous remarquerez deux choses :

- le *passage par référence* : C++ oblige un passage par référence dans le cas d'une construction par recopie. Ceci vient du fait qu'il faut réellement utiliser l'objet lui-même, et non une copie.
- la *présence d'un "`const`"* : en fait, rien n'oblige de le placer ici, c'est simplement une protection de l'objet à recopier. En effet, lors de cette recopie, nous faisons appel à l'objet lui-même, et il n'y aurait aucun sens à vouloir le modifier !

Cet exemple est utile pour introduire la nécessité d'un constructeur par recopie. En effet, dans le cas d'un objet qui comporte un pointeur (notre classe *Vecteur*, par exemple), lorsqu'on veut recopier un objet, on ne recopie pas ce qui est pointé, mais l'adresse du pointeur seulement. Du coup, on se retrouve avec deux objets différents, mais qui possèdent une donnée qui pointe vers la même chose ! Ceci est bien entendu fort dangereux, et par là même, est à éviter.
Un petit schéma pour mieux comprendre :

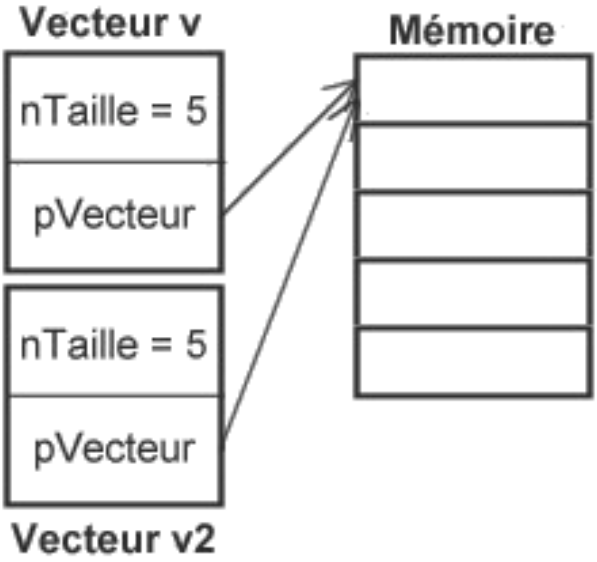


Figure 10 : recopie simple d'un objet Vecteur

On voit clairement qu'après recopie, les membres **`pVecteur`** de **`v`** et de **`v2`** pointent au même endroit.

La parade à celà vient tout naturellement du constructeur par recopie. Au lieu de copier "bêtement" le pointeur, on peut allouer la mémoire et puis recopier les valeurs.

Mise en oeuvre

Application à la classe **`Vecteur`** :

```
#include <iostream.h>

class Vecteur
{
     int *pVecteur;
     int nTaille;

public :
     Vecteur(int); // Construction connaissant la taille du vecteur
     Vecteur(const Vecteur &); // Construction par recopie
     ~Vecteur();   // Destruction

     int GetAt(int ind)       // Lecture de valeur en "inline"
     {
          if( ind>=0 && ind<nTaille )
            return pVecteur[ind];
          else
          // comportement indéfini, pour l'instant retour 0;
            return 0;
     }
     void SetAt(int ind, int val) // Ecriture de valeur
     {
          if( ind>=0 && ind<nTaille )
               pVecteur[ind]=val;
     }
     int Size(){ return nTaille; }
};

Vecteur::Vecteur(int Taille)
{
     // Aucun contrôle de taille n'est fait pour simplifier
     nTaille = Taille;
     pVecteur = new int[nTaille];
}
Vecteur::Vecteur(const Vecteur & v)
{
     nTaille = v.nTaille;
     pVecteur = new int[nTaille]; // allocation de la mémoire
     for( int i=0; i<nTaille; i++ ) // recopie des valeurs
          pVecteur[i]=v.pVecteur[i];
}
Vecteur::~Vecteur()
{
     delete pVecteur;
}

void main()
{
     Vecteur *v;
     v = new Vecteur(10);
     for( int i=0; i<v->Size(); i++ )
          v->SetAt(i, i*i-i);

     Vecteur vv(*v); // construction par recopie
     vv.SetAt(0, 13); // Changement de quelques valeurs
     vv.SetAt(1, 13);
     vv.SetAt(2, 13);

     for( i=0; i<v->Size(); i++ )
```

```
          cout << v->GetAt(i) << " ";
     cout << endl;

     for( i=0; i<vv.Size(); i++ )
          cout << vv.GetAt(i) << " ";
     cout << endl;

     delete v;
}
```

Avec l'exemple de dessus, vous devez obtenir en sortie :

```
0 0 2 6 12 20 30 42 56 72
13 13 13 6 12 20 30 42 56 72
```

Ce qui montre bel et bien l'intérêt de la recopie.

Surdéfinition d'opérateur

- Comment ça marche
- Opérateurs simples
- Opérateur d'affectation

Comment ça marche

Nous avons commencé à réaliser une classe *Point* qui permet la recopie d'objet. Mais jusqu'à maintenant, nous avons été obligés de créer des méthodes telles que "`Coincide`" qui vérifie que deux points sont égaux. Nous aurions pu également réaliser une fonction membre qui ajoute un point à un autre :

```
class Point
{
     int x;
     int y;
public :
     Point();
     Point(int a, int b);
     Point(const Point & pt);
     void Ajoute(const Point & pt);
     ... // D'éventuelles autres méthodes
};
```

Vous sentez alors qu'il serait très appréciable et plus naturel de pouvoir faire quelque chose comme :

```
     Point a(1,2);
     Point b(3,4);
     Point c;
     c = a + b;
```

C++ permet de réaliser une surdéfinition des opérateurs de base, comme "**+**", "**-**", "*****", "**/**", "**&**", "**^**", etc. La liberté étant totale, vous pouvez faire réellement ce que vous voulez, par exemple une soustraction pour l'opérateur d'addition et inversement. Mais il est clair qu'il est plus que conseillé de respecter la signification de chacun des ces opérateurs.

Opérateurs simples

Mettre en oeuvre les opérateurs simples est une opération assez rapide. Un exemple est utile pour vous montrer comment on fait :

```
class Point
{
     int x;
     int y;
public :
     Point();
     Point(int a, int b);
     Point(const Point & pt);
```

```
     Point operator+(const Point & a)
     {
          Point p;
          p.x = a.x + x;
          p.y = a.y + y;
        return p;
     }
     ... // D'éventuelles autres méthodes
};
```

Des explications sont nécessaires. Tout d'abord, vous remarquerez le "`const Point & a `". Ceci signifie que l'on passe un point en paramètre (l'autre point de l'addition est en fait la classe appelante elle-même). Ce dernier est transmis par référence, afin d'éviter une recopie, lourde et moins rapide. Le "`const`" est optionnel mais permet d'éviter de modifier les paramètres et également autorise l'utilisation d'objets constants.
L'opérateur rend un *Point*. En fait, on rend en fin de méthode le point qui a été créé temporairement au départ et qui contient la somme des deux paramètres. Plus exactement, on rend une copie de cet objet, le retour de la fonction étant "`Point`", et non "`Point&`" ou "`Point*`". Ceci est normal. Il faut savoir que l'objet créé dans la méthode sera automatiquement détruit à la fin de cette dernière. On ne peut bien évidemment pas rendre l'adresse d'un objet qui sera détruit ! D'où la nécessité d'une recopie.

Ce que nous avons mis en oeuvre pour l'opérateur d'addition, nous pouvons en faire de même pour tous les autres opérateurs "*simples*". Quelques exemples :

```
class Point
{
 int x;
 int y;
public :
 Point(){}
 Point(int a, int b){ x=a; y=b; }
 Point operator +(const Point & a);
 Point operator -(const Point & a);
 int operator==(const Point & p);

 void Affiche()
 {
  // "this" est un pointeur sur la classe même
  cout << this << "->" << x << ", " << y << endl;
 }
};

Point Point::operator +(const Point & a)
{ // Addition de 2 points
 Point p;
 p.x = x + a.x;
 p.y = y + a.y;

 return p;
```

```
}
Point Point::operator -(const Point & a)
{ // Soustraction de 2 points
 Point p;
 p.x = x - a.x;
 p.y = y - a.y;

 return p;
}

int Point::operator==(const Point & p)
{ // Egalité de 2 points (remplace "Coincide")
 if( x==p.x && y==p.y )
  return 1;
 else
  return 0;
}

void main()
{
 Point p(1,2);
 p.Affiche();
 Point pp(3,4);
 pp.Affiche();
 Point ppp = p+pp;
 ppp.Affiche();

 if( p==pp )
  cout << "p==pp" << endl;
 else
  cout << "p!=pp" << endl;

 p = ppp;
 p.Affiche();
 pp = p-ppp;
 pp.Affiche();

 if( p==ppp )
  cout << "p==ppp" << endl;
 else
  cout << "p!=ppp" << endl;
}
```

Opérateur d'affectation

L'opérateur d'affectation "**=**" est assez particulier. En effet, nous retrouvons le même problème que lors de la construction par recopie : il est toujours possible d'effectuer une affectation entre deux objets (de même type), mais que se passe-t-il s'ils contiennent des pointeurs (cf. *problématique de recopie*) ! C'est pourquoi il est souvent important d'implémenter ce type d'opérateur.

Il n'est pas plus difficile à mettre en oeuvre :

```
#include <iostream.h>

class Vecteur
{
 int *pVecteur;
 int nTaille;

public :
 Vecteur(int);
 Vecteur(const Vecteur &);
 ~Vecteur();

 int GetAt(int ind)
 {
  if( ind>=0 && ind<nTaille )
   return pVecteur[ind];
  else
   return 0;
 }
 void SetAt(int ind, int val)
 {
  if( ind>=0 && ind<nTaille )
   pVecteur[ind]=val;
 }
 int Size(){ return nTaille; }
 Vecteur& operator =(const Vecteur & v);
};

Vecteur::Vecteur(int Taille)
{
 nTaille = Taille;
 pVecteur = newint[nTaille];
}
Vecteur::Vecteur(const Vecteur & v)
{
 nTaille = v.nTaille;
 pVecteur = newint[nTaille];

 for( int i=0; i<nTaille; i++ )
  pVecteur[i]=v.pVecteur[i];
}
Vecteur::~Vecteur()
{
 delete pVecteur;
}

Vecteur& Vecteur::operator =(const Vecteur & v)
{
 // On vérifie que les objets ne sont pas les mêmes !
 if( this != &v )
 {
  delete pVecteur; // Effacement du vecteur
  nTaille = v.nTaille;
  pVecteur = newint[nTaille]; // Allocation

  // Recopie des valeurs
  for( int i=0; i<nTaille; i++ )
```

```
   pVecteur[i]=v.pVecteur[i];
 }
 return *this;
}

void main()
{
 Vecteur *v;
 v = new Vecteur(10);
 for( int i=0; i<v->Size(); i++ )
  v->SetAt(i, i*i-i);

 Vecteur vv(5);
 vv = *v;
 vv.SetAt(0, 13);
 vv.SetAt(1, 13);
 vv.SetAt(2, 13);

 for( i=0; i<v->Size(); i++ )
  cout << v->GetAt(i) << " ";
 cout << endl;

 for( i=0; i<vv.Size(); i++ )
  cout << vv.GetAt(i) << " ";
 cout << endl;

 delete v;
}
```

Cette fois-ci, nous rendons par contre une référence sur l'objet, car nous devons rendre la classe elle-même et non une copie, comme vous pouvez le comprendre !

L'Héritage

- Définition et mise en oeuvre
- Utilisation des membres de la classe de base
- Redéfinition des fonctions membre et appel des constructeurs
- "Statuts" de dérivation
- Notion d'héritage : élargissement

Définition et mise en oeuvre

Définition

Nous avons vu dans le premier chapitre que l'**héritage** est en C++, et plus généralement en P.O.O., un concept fondamental.
En effet, il permet de définir une nouvelle classe "*fille*", qui héritera donc des caractéristiques de la classe de base. Le terme "*caractéristiques*" inclut en fait l'ensemble de la définition de cette classe mère.
D'un point de vue pratique, il faut savoir qu'il n'est pas nécessaire à la classe fille de connaître l'implémentation de la base : sa définition suffit. Ceci sera mis en application. De plus, on peut hériter plusieurs fois de la même classe, et une classe fille pourra également servir de base pour une autre. Il est alors possible de décrire un véritable "**arbre d'héritage**".

Mise en œuvre

Mettre en oeuvre la technique de l'héritage est assez en simple en C++. Le plus difficile reste en fait la **conception**, qui nécessite un gros travail afin de bien séparer les différentes classes.

Le premier exemple qui nous permettra de réaliser notre premier héritage, propose de définir une classe **PointCol** qui hérite de **Point**.
Sémantiquement parlant, *PointCol* est un *Point* auquel on rajoute la gestion de la couleur. Nous obtenons alors :

```
class PointCol : public Point
{
     unsigned char byRed;   // La composante rouge
     unsigned char byGreen; // La composante verte
     unsigned char byBlue;  // La composante bleue
public :
     // Coloration d'un point
     void Colore( unsigned char R, unsigned char G, unsigned char B )
     {
          byRed = R;
          byGreen = G;
          byBlue = B;
     }
};
```

Vous pouvez remarquer la notation "`: public Point`". Ceci signifie que notre point coloré hérite **publiquement** de *Point*, c'est-à-dire que **tous les membres publics de *Point* seront membres publics de *PointCol***.
En déclarant un objet de type *PointCol*, il est ainsi possible d'accéder aux

membres publics de *PointCol*, donc, mais également de *Point*. C'est une notion essentielle de la P.O.O. !

Pour mettre en application notre exemple, nous allons utiliser la classe Point qui suit. Nous allons pour la première fois faire cela dans les "règles de l'art", en séparant physiquement la définition de l'implémentation (un fichier "**.h**" et un fichier "**.cpp**").

Fichier **Point.h** :

```
class Point
{
 int x;
 int y;
public :
 Point(){}

 void Init(int a, int b){ x=a; y=b; }
 void Deplace(int a, int b){ x+=a; y+=b; }
 void Affiche();
};
```

Fichier **Point.cpp** :

```
#include <iostream.h>
#include "Point.h"
void Point::Affiche()
{
 cout << this << "->" << x << ", " << y << endl;
}
```

Vous pouvez maintenant rajouter le fichier `main.cpp` suivant :

```
#include "Point.h"

class PointCol : public Point
{
 unsigned char byRed;   // La composante rouge
 unsigned char byGreen; // La composante verte
 unsigned char byBlue;  // La composante bleue
public :
 // Coloration d'un point
 void Colore( unsigned char R, unsigned char G, unsigned char B )
 {
  byRed = R;
  byGreen = G;
  byBlue = B;
 }
};

void main()
{
 PointCol ptc;
 ptc.Init( 5, 10 );
```

```
 ptc.Colore( 64, 128, 192 );
 ptc.Affiche();
 ptc.Deplace( 3, 6 );
 ptc.Affiche();
}
```

Utilisation des membres de la classe de base

Utiliser des membres de la classe de base est simple à réaliser. Il faut cependant faire attention leur statut. Les membres privés ne peuvent être appelés. Soit une méthode "`InitCol`" qui initialise un point coloré :

```
void InitCol( int Abs, int Ord, unsigned char R, unsigned char G, unsigned
char B )
{
 Point::Init(Abs, Ord);
 byRed = R;
 byGreen = G;
 byBlue = B;
}
```

Il suffit donc d'appeler la méthode souhaitée, précédée de la classe.

Redéfinition des fonctions membre et appel des constructeurs

L'appel à la méthode "`Affiche`" fonctionne très bien, et utilise en fait la déclaration faite dans la classe *Point*, ce qui est logique puisque *PointCol* n'en possède aucune autre. Mais que se passe-t-il si on veut afficher un point coloré ?
Une première solution consiste à introduire une nouvelle méthode "`AfficheCol`" dans la classe fille.

En plus de cette méthode, nous allons ajouter un constructeur qui permet l'initialisation de la classe *PointCol*. Vous voyez immédiatement quelle en pourrait être la définition :

```
PointCol( int Abs, int Ord, unsigned char R, unsigned char G, unsigned char
B);
```

Il contient donc les coordonnées du point, plus les composantes de la couleur. La mise en oeuvre est un peu plus complexe. Soit notre classe *Point* :

```
class Point
{
 int x;
 int y;
public :
 Point(int, int);

 void Deplace(int a, int b){ x+=a; y+=b; }
 void Affiche();
```

```
};
```

Nous voyons bien que, quelque part, il faudrait passer les coordonnées entrées en paramètres du constructeur initialisant de *PointCol*, vers celui de *Point* ! Ceci s'effectue de la façon suivante :

```
#include "Point.h"

class PointCol : public Point
{
 unsigned char byRed;
 unsigned char byGreen;
 unsigned char byBlue;
public :
 PointCol( int, int, unsigned char, unsigned char, unsigned char );

 void Colore( unsigned char, unsigned char, unsigned char );
};

// Constructeur initialisant de la classe PointCol,
// faisant appel au constructeur initialisant de Point.
PointCol::PointCol( int Abs, int Ord, unsigned char R, unsigned char G,
unsigned char B) : Point(Abs, Ord)
{
 byRed = R;
 byGreen = G;
 byBlue = B;
}

void PointCol::Colore( unsigned char R, unsigned char G, unsigned char B )
{
 byRed = R;
 byGreen = G;
 byBlue = B;
}
```

Remarquez la transmission de paramètres effectuée par le "**:Point(Abs, Ord)**". C'est en fait un appel au constructeur initialisant de la classe de base.
Il est possible d'étendre cette mise en oeuvre à tous les constructeurs, par exemple par recopie. Essayez ! Vous pouvez également changer le type d'héritage et le rendre "`private`", pour voir la différence.

Dans certains cas, il peut être intéressant de pouvoir avoir accès aux données membres de la classe de base. Par exemple, reprenons notre affichage dans *PointCol* :

```
#include "Point.h"

class PointCol : public Point
{
 unsigned char byRed;
 unsigned char byGreen;
```

```
 unsigned char byBlue;
public :
 PointCol( int, int, unsigned char, unsigned char, unsigned char );

 void Colore( unsigned char, unsigned char, unsigned char );
 void AfficheCol();
};

PointCol::PointCol( int Abs, int Ord, unsigned char R, unsigned char G,
unsigned char B) : Point(Abs, Ord)
{
 byRed = R;
 byGreen = G;
 byBlue = B;
}

void PointCol::Colore( unsigned char R, unsigned char G, unsigned char B )
{
 byRed = R;
 byGreen = G;
 byBlue = B;
}

void PointCol::AfficheCol()
{
 Point::Affiche();
 // Notez le "cast" en "int" des composantes nécessaire, sinon le
 // compilateur prend les valeurs (char) pour des caractères...
 cout << "Couleur : RGB(" << (int)byRed << "," << (int)byGreen;
 cout << "," << (int)byBlue << ")." << endl;
}
```

Le résultat est satisfaisant, mais un appel à la méthode `Affiche` de *Point* est peut-être fastidieux, d'autant plus qu'il pourrait être intéressant d'avoir accès aux coordonnées du ponit, directement.
Ceci n'est pas possible ! Si vous essayez, le compilateur vous donnera une erreur de type : **`cannot access private member declared in class 'Point'`**. Ceci vient du fait que les membres `x` et `y` de *Point* sont privés.

"Statuts" de dérivation

La solution à ce problème permet de laisser les membres inaccessibles aux utilisateurs de la classe, mais pas des classes qui en héritent. Il suffit de remplacer le "**`private`**" par "**`protected`**". Notre classe *Point* devient alors :

```
class Point
{
protected :
 int x;
 int y;
public :
 Point(int abs, int ord){ x=abs; y=ord; }

 void Deplace(int a, int b){ x+=a; y+=b; }
```

```
 void Affiche();
};
```

Et notre méthode `AfficheCol` :

```
void PointCol::AfficheCol()
{
 cout << "Point (" << x << ", "<< y << ") ";
 cout << "de couleur : RGB(" << (int)byRed << "," << (int)byGreen;
 cout << "," << (int)byBlue << ")." << endl;
}
```

L'intérêt du statut **protégé** est donc double puisque les données se retrouvent inaccessibles pour l'extérieur, ce qui préserve l'encapsulation des données de la classe, mais contre sont toujours utilisables pour d'éventuels héritages.

Notion d'héritage : élargissement

Il est possible d'étendre la notion d'héritage à plusieurs classes : un véritable arbre peut-être créé, par exemple une classe C qui hériterait de A et B (héritage multiple). Nous n'aborderons pas ces fonctionnalités dans ce tutorial, mais vous pouvez consulter un livre plus complet abordera cela.

Fonctions Virtuelles

- Utilité
- Mécanisme

Utilité

Nous avons acquis dans le chapitre précédent, la notion d'héritage. Elle nous permet en outre de créer de véritables arbres de classes. Reprenons notre exemple de *Point* et de *PointCol*. Nous avons implémenté des méthodes qui permettent l'affichage, ou encore l'initalisation des données, dans chacune des deux classes : `Affiche` dans *Point*, `AfficheCol` dans *PointCol* par exemple.
Je suppose que vous vous êtes demandé pourquoi nous ne leur avons pas donné le même nom ! Le mieux pour le comprendre est d'essayer.

```
#include "Point.h"

class PointCol : public Point
{
 unsigned char byRed;
 unsigned char byGreen;
 unsigned char byBlue;
public :
 PointCol( int, int, unsigned char, unsigned char, unsigned char );

 void Colore( unsigned char, unsigned char, unsigned char );
 void Affiche();
};

PointCol::PointCol( int Abs, int Ord, unsigned char R, unsigned char G,
unsigned char B) : Point(Abs, Ord)
{
 byRed = R;
 byGreen = G;
 byBlue = B;
}

void PointCol::Colore( unsigned char R, unsigned char G, unsigned char B )
{
 byRed = R;
 byGreen = G;
 byBlue = B;
}

void PointCol::Affiche()
{
 cout << "Point (" << x << ", " << y << ")";
 cout << "de couleur : RGB(" << (int)byRed << "," << (int)byGreen;
 cout << "," << (int)byBlue << ")." << endl;
}
```

Cette déclaration de la classe *PointCol* à l'exécution, donne les résultats voulus, à savoir que c'est la bonne méthode `Affiche` qui est appelée. En fait, la liaison est établie statiquement à la compilation. Ici, le compilateur

sait très bien quelle fonction membre utiliser. Mais maintenant, imaginons l'utilisation suivante :

```
void main()
{
 PointCol ptc(5,10, 50,150,200);
 ptc.Affiche();

 Point pt(52,17);
 pt.Affiche();

 Point *ppt; // Pointeur de point
 ppt = &ptc; // le pointeur pointe désormais sur un point coloré : légal !
 ppt->Affiche();
}
```

Le résultat peut vous sembler surprenant. En fait, la typage étant effectué statiquement, pour le compilateur, **`ppt`** reste quoi qu'il advienne un pointeur sur *Point*.
Or, nous n'avions pas vu cela encore, mais il est possible d'affecter une adresse de classe fille à un pointeur de classe de base...
Dans ce cas, vu que l'affectation est dynamique, un appel de la méthode `Affiche` utilise en fait la déclaration de la classe de base, *Point*.

Un autre exemple pour illustrer l'utilité des fonctions virtuelles, consisterait à réaliser un affichage "descendant". Il s'agit de réaliser un affichage de toutes les informations relatives aux deux classes, *Point* et *PointCol*, en appelant une seule méthode de *Point*. Vous comprendrez mieux cela, en étudiant le code :

```
// Point.h
class Point
{
 int x;
 int y;
public :
 Point(int a, int b){ x=a; y=b; }

 void Init(int a, int b){ x=a; y=b; }
 void Deplace(int a, int b){ x+=a; y+=b; }
 void Affiche();
 void AfficheTout();
};

// Point.cpp
void Point::Affiche()
{
 cout << this << "->" << x << ", " << y << endl;
}

void Point::AfficheTout()
{
 cout << this << "->" << x << ", " << y << endl;
 Affiche();
```

```
}
#include "Point.h"

class PointCol : public Point
{
 unsigned char byRed;
 unsigned char byGreen;
 unsigned char byBlue;
public :
 PointCol( int, int, unsigned char, unsigned char, unsigned char );

 void Colore( unsigned char, unsigned char, unsigned char );
 void Affiche();
};

PointCol::PointCol( int Abs, int Ord, unsigned char R, unsigned char G,
unsigned char B) : Point(Abs, Ord)
{
 byRed = R;
 byGreen = G;
 byBlue = B;
}

void PointCol::Colore( unsigned char R, unsigned char G, unsigned char B )
{
 byRed = R;
 byGreen = G;
 byBlue = B;
}

void PointCol::Affiche()
{
 cout << "Couleur : RGB(" << (int)byRed << "," << (int)byGreen;
 cout << "," << (int)byBlue << ")." << endl;
}
```

En appelant `AfficheTout` dans le programme, nous souhaitons afficher les renseignements de la classe *Point* (code contenu dans la première ligne de la méthode), mais aussi ceux de la classe appelante. Par exemple :

```
void main()
{
 PointCol ptc(5,10, 50,150,200);
 ptc.Affiche();

 Point pt(52,17);
 pt.Affiche();

 ptc.AfficheTout();
}
```

Dans ce cas précis, nous affichons les informations de couleurs de `ptc`, puis de coordonnées de `pt`. La dernière ligne devrait afficher les informations de coordonnées et de couleurs de `ptc`. Mais de ce premier

test, il n'en est rien !

Mécanisme

Dans le paragraphe précédent, nous avons vu que dans certaines conditions, donner le même nom à une méthode de la classe fille qu'une méthode de la classe de base, peut être source d'erreur, ou plutôt, d'incompréhension.
Pour éviter cela, il faudrait pouvoir indiquer au compilateur de lier certaines méthodes **dynamiquement**. En effet, il y a des cas où, seulement à l'exécution, le programme peut savoir quelle fonction membre employer. Les fonctions virtuelles servent à celà. Soit la définition suivante :

```
// Point.h
class Point
{
 int x;
 int y;
public :
 Point(int a, int b){ x=a; y=b; }

 void Init(int a, int b){ x=a; y=b; }
 void Deplace(int a, int b){ x+=a; y+=b; }
 virtual void Affiche();
 void AfficheTout();
};

// Point.cpp
void Point::Affiche()
{
 cout << this << "->" << x << ", " << y << endl;
}

void Point::AfficheTout()
{
 cout << this << "->" << x << ", " << y << endl;
 Affiche();
}
```

Essayez maintenant les deux tests précédemment implémentés. Vous constaterez que grâce à ce `virtual`, la liaison est désormais dynamique et donc, que la **bonne méthode est appelée**.

Exercices...

Vous avez appris très rapidement les "bases" du C++ dans les précédents chapitres. Il est évident que vous n'avez pas encore pu assimiler toutes ces notions. Pour se faire, rien n'est plus efficace qu'une mise en application.

L'exercice qui vous est proposé, a pour objet la réalisation d'un ensemble de deux classes, qui s'occupent de la gestion de chaînes de caractères formatées (mise en italique, gras et couleur).

La première classe gère la première partie, à savoir la chaîne de caractères. En voici sa description rapide :

- **gestion d'une chaîne de caractères et de sa taille,**
- constructeur par défaut,
- constructeur initialisant à partir d'une chaîne de caractères (`char*`),
- constructeur par recopie,
- destructeur,
- surcharge de l'opérateur = (affectation de chaîne),
- surcharge de l'opérateur == (égalité de chaîne),
- surcharge de l'opérateur += (trois différents, un qui gère un `String`, un autre un `char*` et un dernier un `char`),
- surcharge de l'opérateur + (concaténation),
- surcharge de l'opérateur [] (accès à un caractère de la chaîne stockée),
- vérification de l'initialisation de la classe (on vérifie que la chaîne n'est pas vide),
- mise à zéro des paramètres (chaîne vide),
- renvoie de la taille de la chaîne,
- et affichage de la chaîne.

Vous êtes bien entendu libre de rajouter d'autres méthodes.

La deuxième classe hérite de la première, et rajoute une couche gérant le formatage du texte. Description sommaire :

- **gestion du formatage : Italic, Bold et Couleur** (un short pour faire simple)**,**
- constructeur par défaut,
- constructeur connaissant une chaîne de caractères et *éventuellement* les options de formatage,
- constructeur connaissant une chaîne de caractères de type **String** et *éventuellement* les options de formatage,
- constructeur connaissant les options de formatage,
- constructeur par recopie,

- destructeur (optionnel),
- surcharge de l'opérateur = (affectation) pour le cas d'une copie de **chaîne formatée**,
- surcharge de l'opérateur = (affectation) pour le cas d'une copie de **chaîne non formatée** (classe de base),
- méthodes permet la gestion de l'italique (mise en "italic" et renvoie d'information),
- méthodes permet la gestion de Bold (mise en "Bold" et renvoie d'information),
- Colorisation et renvoie de couleur,
- Affichage de la chaîne et des informations de formatage, en utilisation la notation HTML, à savoir :
    1. **** pour la mise en italique et **** à la fin,
    2. **** pour la mise en gras et **** à la fin,
    3. **** pour la couleur et **** à la fin.

La réalisation de ces classes est assez simple si vous procédez par étape. Utilisez également beaucoup les exemples fournis auparavant. N'hésitez à regarder comment on déclare un constructeur par recopie, etc.

C'est également le moment pour voir de plus près comment fonctionne le debugger, car vous ne serez pas sans faire quelques erreurs... de frappe !

Bon courage !

Solution de l'exercice

La solution a été décomposée en quatre blocs, correspondant aux deux classes (`.h` et `.cpp`) :

- La classe chaîne de base : .h et .cpp
- La classe chaîne formatée : .h et .cpp

Il est également possible de consulter les fichiers de description des classes auto-générés à partir du code source commenté...:

- Classe String,
- classe FormatedString.

```
// chaine.h         : Interface de la classe String
//////////////////////////////////////////////////////////////////////////////////
//
// Auteur :         Gwenaël Brunet      E-Mail : brunet@traezh.enst-bretagne.fr
// --------         Dpt iTi - LATIM
//                  ENST Bretagne       Tél. : 02.98.00.10.61
//                  BP 832              Fax  : 02.98.00.10.98
//                  29285 BREST Cedex
//
//////////////////////////////////////////////////////////////////////////////////
//
// Classe String :
// ---------------
// Description :    Interface de la classe String
// Utilisation :    Gestion des chaînes de caractères, au sens large, c'est-à-dire qu'il\
//                  n'y a pas de distinction faite entre un mot, une ligne, etc..
//
// Création    :    11/06/99
//
// Dernière
// Mise à jour :    14/06/99
//
/////[ INFORMATIONS ]/////////////////////////////////////////////////////////////
//
// Cette classe sert d'illustration, d'exemple, au tutorial de C++ suivant :\n
// \n
// http://www-iti.enst-bretagne.fr/~brunet/MesCours/Tutorial_C++/\n
// \n
// Elle n'a donc aucune autre utilité que la pédagogie...
//
//////////////////////////////////////////////////////////////////////////////////
#ifndef _CLASS_STRING
#define _CLASS_STRING

// Quelques "define" utiles.
#define UINT unsigned int
#define NULL 0

class String
{
protected :
          char *m_strString;  // la chaîne en elle-même
          UINT m_uiSize;                // la longueur de la chaîne

public :
          String( );
          String( char* );
          String( const String & );
          ~String( );

          String & operator=( const String & );
          bool operator==( const String & );

          String& operator+=( const String & );
          String& operator+=( const char* );
          String& operator+=( const char );
          String operator+( const String & );

          char & operator[](UINT);

          bool IsEmpty();
          void Empty();
          UINT GetSize(){ return m_uiSize; }

          virtual void Display(bool bEnter=true);
};

#endif // _CLASS_STRING
```

```
// chaine.cpp       : Implémentation de la classe String
//////////////////////////////////////////////////////////////////////////////
//
// Auteur :         Gwenaël Brunet      E-Mail : brunet@traezh.enst-bretagne.fr
// --------         Dpt iTi - LATIM
//                  ENST Bretagne       Tél. : 02.98.00.10.61
//                  BP 832              Fax  : 02.98.00.10.98
//                  29285 BREST Cedex
//
//////////////////////////////////////////////////////////////////////////////
//
// Classe String :
// ---------------
// Description :    Implémentation de la classe String
// Utilisation :    Gestion des chaînes de caractères, au sens large, c'est-à-dire qu'il\
//                  n'y a pas de distinction faite entre un mot, une ligne, etc..
//
// Création    :    11/06/99
//
// Dernière
// Mise à jour :    14/06/99
//
/////[ INFORMATIONS ]/////////////////////////////////////////////////////////
//
// Cette classe sert d'illustration, d'exemple, au tutorial de C++ suivant :\n
// \n
// http://www-iti.enst-bretagne.fr/~brunet/MesCours/Tutorial_C++/\n
// \n
// Elle n'a donc aucune autre utilité que la pédagogie...
//
//////////////////////////////////////////////////////////////////////////////
#include &ltiostream.h>
#include "chaine.h"


//////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 15:33:05]
//  Class   : String
//  Function: String
//
//  Description: Constructeur par défaut de String -> initialisation à "0"
//
//  Parameters: None
//  Return: None
//
//////////////////////////////////////////////////////////////////////////////
String::String( )
{
          m_uiSize = 0;
          m_strString = NULL;
}

//////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 15:57:19]
//  Class   : String
//  Function: String
//
//  Description: Constructeur initialisant par chaîne
//
//  Parameters:
//        char *string -  La chaîne d'initialisation (recopiée)
//  Return: None
//
//////////////////////////////////////////////////////////////////////////////
String::String( char *string )
{
          m_strString = string;
          m_uiSize = 0;

          // Mesure de la chaîne
          while( m_strString[m_uiSize]!='\0' )
                    m_uiSize++;

          m_strString = new char[m_uiSize];       // Allocation mémoire
          for( UINT i=0; i&ltm_uiSize; i++ )
                    // Recopie
                    m_strString[i] = string[i];
}

//////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 15:59:13]
//  Class   : String
//  Function: String
//
//  Description: Constructeur par recopie
//
//  Parameters:
//        const String &string -  chaîne à recopier
//  Return: None
//
```

```
////////////////////////////////////////////////////////////////////////////
String::String( const String &string )
{
          m_uiSize = string.m_uiSize;
          m_strString = new char[m_uiSize];

          for( UINT i=0; i&ltm_uiSize; i++ )
                    // Recopie
                    m_strString[i] = string.m_strString[i];
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 16:00:06]
//  Class   : String
//  Function: ~String
//
//  Description: Destructeur
//
//  Parameters: None
//  Return: None
//
////////////////////////////////////////////////////////////////////////////
String::~String( )
{
          Empty();
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 16:08:19]
//  Class   : String
//  Function: operator=
//
//  Description: Surcharge de l'opérateur d'affectation
//
//  Parameters:
//        const String &string -  la chaîne à copier
//  Return: String & -  *this
//
////////////////////////////////////////////////////////////////////////////
String & String::operator=( const String &string )
{
          if( this!= &string )
          {
                    if( m_strString )
                              delete m_strString;

                    m_uiSize = string.m_uiSize;
                    m_strString = new char[m_uiSize];

                    for( UINT i=0; i&ltm_uiSize; i++ )
                              // Recopie
                              m_strString[i] = string.m_strString[i];
          }

          return *this;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 16:10:30]
//  Class   : String
//  Function: operator==
//
//  Description: Surcharge de l'opérateur d'égalité
//
//  Parameters:
//        const String &string -  la chaîne à comparer
//  Return: bool -  le résultat de la comparaison
//
////////////////////////////////////////////////////////////////////////////
bool String::operator==( const String &string )
{
          // UPDATE : merci Stéphane !
          if( m_uiSize!=string.m_uiSize )
                    return false;
          // ----------------------------

          for( UINT i=0; i&ltm_uiSize; i++ )
                    if( m_strString[i]!=string.m_strString[i] ) return false;
          return true;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 16:50:02]
//  Class   : String
//  Function: operator+
//
//  Description: Surcharge de l'opérateur d'addition
//
//  Parameters:
```

```
//        const String &string -  La chaîne à concaténer
//  Return: String -  la chaîne de retour
//
////////////////////////////////////////////////////////////////////////////
String String::operator+( const String &string )
{
          String buffer; // variable temporaire de retour

          buffer.m_uiSize = m_uiSize + string.m_uiSize;
          buffer.m_strString = new char[buffer.m_uiSize];

          // Recopie
          for( UINT i=0; i<m_uiSize; i++ )
                    buffer.m_strString[i] = m_strString[i];
          for( i=0; i<string.m_uiSize; i++ )
                    buffer.m_strString[i+m_uiSize] = string.m_strString[i];

          return buffer;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 10:10:47]
//  Class   : String
//  Function: operator+=
//
//  Description: Surcharge de l'opérateur += -> concaténation
//
//  Parameters:
//        const String &string -  la chaîne de type String à ajouter
//  Return: String& -  La chaîne en retour
//
////////////////////////////////////////////////////////////////////////////
String& String::operator+=( const String &string )
{
          UINT NewSize = string.m_uiSize + m_uiSize;
          char *buffer = new char[NewSize];

          for( UINT i=0; i<m_uiSize; i++ )
                    buffer[i] = m_strString[i];
          for( i=0; i<string.m_uiSize; i++ )
                    buffer[i+m_uiSize] = string.m_strString[i];

          delete m_strString;
          m_strString = buffer;
          m_uiSize = NewSize;

          return *this;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 10:15:01]
//  Class   : String
//  Function: operator+=
//
//  Description: Surcharge de l'opérateur += -> concaténation
//
//  Parameters:
//        const char string -  la chaîne de type char* à ajouter
//  Return: String& -  La chaîne en retour
//
////////////////////////////////////////////////////////////////////////////
String& String::operator+=( const char *string )
{
          UINT size=0;
          while( string[size]!='\0' )
                    size++;

          char *buffer = new char[size+m_uiSize];

          for( UINT i=0; i<m_uiSize; i++ )
                    buffer[i] = m_strString[i];
          for( i=0; i<size; i++ )
                    buffer[i+m_uiSize] = string[i];

          delete m_strString;
          m_strString = buffer;
          m_uiSize+=size;

          return *this;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 10:18:49]
//  Class   : String
//  Function: operator+=
//
//  Description: Concaténation d'un caractère
//
//  Parameters:
//        const char c -  Le caractère
```

```
//  Return: String& -  La chaîne en retour
//
////////////////////////////////////////////////////////////////////////////
String& String::operator+=( const char c )
{
          m_uiSize++;

          char *buffer = new char[m_uiSize];

          for( UINT i=0; i&ltm_uiSize; i++ )
                    buffer[i] = m_strString[i];
          buffer[m_uiSize-1] = c;

          delete m_strString;
          m_strString = buffer;

          return *this;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 16:51:01]
//  Class   : String
//  Function: operator[]
//
//  Description: Accès à un caractère
//
//  Parameters:
//        int nIndice -  l'indice du caractère
//  Return: char & -  le caractère
//
////////////////////////////////////////////////////////////////////////////
char & String::operator[](UINT nIndice)
{
          if( nIndice<0 || nIndice>=m_uiSize )
                    return m_strString[0];

          return m_strString[nIndice];
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 16:57:02]
//  Class   : String
//  Function: IsEmpty
//
//  Description: Indique si la chaîne est vide
//
//  Parameters: None
//  Return: bool -  true si la chaîne est vide
//
////////////////////////////////////////////////////////////////////////////
bool String::IsEmpty()
{
          if( m_uiSize==0 )
                    return true;

          return false;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 17:01:13]
//  Class   : String
//  Function: Empty
//
//  Description: Efface la chaîne
//
//  Parameters: None
//  Return: None
//
////////////////////////////////////////////////////////////////////////////
void String::Empty()
{
          if( m_strString )
          {
                    delete m_strString;
                    m_strString = NULL;
          }
          m_uiSize = 0;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [11/06/99 17:00:49]
//  Class   : String
//  Function: Display
//
//  Description: Ecrit la chaîne
//
//  Parameters:
//        bool bEnter -  true (défaut) si on veut faire un retour à la fin
//  Return: None
```

```
//
//////////////////////////////////////////////////////////////////////////
void String::Display( bool bEnter )
{
          for( UINT i=0; i&ltm_uiSize; i++ )
                    cout << m_strString[i];
          if( bEnter )
                    cout << endl;
}
```

```
// FormatedString.h : Interface de la classe FormatedString
//////////////////////////////////////////////////////////////////////////////////////
//
// Auteur :          Gwenaël Brunet      E-Mail : brunet@traezh.enst-bretagne.fr
// --------          Dpt iTi - LATIM
//                   ENST Bretagne       Tél. : 02.98.00.10.61
//                   BP 832              Fax  : 02.98.00.10.98
//                   29285 BREST Cedex
//
//////////////////////////////////////////////////////////////////////////////////////
//
// Classe FormatedString :
// -----------------------
// Description :     Interface de la classe FormatedString
// Utilisation :     Gestion des chaînes de caractères formatée : hérite de la classe\
//                   String, et lui rajoute des notions de formatage de texte, italic,\
//                   gras et couleur.
//
// Création :        12/06/99
//
// Dernière
// Mise à jour :     14/06/99
//
/////[ INFORMATIONS ]/////////////////////////////////////////////////////////////////
//
// Cette classe sert d'illustration, d'exemple, au tutorial de C++ suivant :\n
// \n
// http://www-iti.enst-bretagne.fr/~brunet/MesCours/Tutorial_C++/\n
// \n
// Elle n'a donc aucune autre utilité que la pédagogie...
//
//////////////////////////////////////////////////////////////////////////////////////
#ifndef _CLASS_FORMATEDSTRING
#define _CLASS_FORMATEDSTRING

#include "chaine.h"

#define DEFAULT_FORMATEDSTRING_COLOR 0

class FormatedString : public String
{
          bool m_bItalic;
          bool m_bBold;
          short m_sColor;

          // Initialisation interne
          inline void InitFormat( bool bItal=0, bool bBold=0, short
Color=DEFAULT_FORMATEDSTRING_COLOR );

public :
          // Constructeurs / destructeur
          FormatedString( );
          FormatedString( char* str, bool bItal=0, bool bBold=0, short
Color=DEFAULT_FORMATEDSTRING_COLOR );
          FormatedString( const String & str, bool bItal=0, bool bBold=0, short
Color=DEFAULT_FORMATEDSTRING_COLOR );
          FormatedString( bool bItal, bool bBold, short Color=DEFAULT_FORMATEDSTRING_COLOR );
          FormatedString( const FormatedString & fstr);
          ~FormatedString( );

          // Surcharge opérateurs
          FormatedString & operator=( FormatedString & fstr);
          FormatedString & operator=( String & str);

          // Accès aux variables de formatage
          void MakeItalic( bool bItal=true );
          bool IsItalic( );

          void MakeBold( bool bBold=true );
          bool IsBold( );

          void Colorize( short sColor );
          short GetColor( );

          void Display( bool bReturn=true );
};

#endif // _CLASS_FORMATEDSTRING
```

```
// FormatedString.cpp         : Implémentation de la classe FormatedString
//////////////////////////////////////////////////////////////////////////////////////
//
// Auteur :         Gwenaël Brunet      E-Mail : brunet@traezh.enst-bretagne.fr
// --------         Dpt iTi - LATIM
//                  ENST Bretagne       Tél. : 02.98.00.10.61
//                  BP 832              Fax  : 02.98.00.10.98
//                  29285 BREST Cedex
//
//////////////////////////////////////////////////////////////////////////////////////
//
// Classe FormatedString :
// -----------------------
// Description :    Implémentation de la classe FormatedString
// Utilisation :    Gestion des chaînes de caractères formatée : hérite de la classe\
//                  String, et lui rajoute des notions de formatage de texte, italic,\
//                  gras et couleur.
//
// Création    :    12/06/99
//
// Dernière
// Mise à jour :    14/06/99
//
/////[ INFORMATIONS ]/////////////////////////////////////////////////////////////////
//
// Cette classe sert d'illustration, d'exemple, au tutorial de C++ suivant :\n
// \n
// http://www-iti.enst-bretagne.fr/~brunet/MesCours/Tutorial_C++/\n
// \n
// Elle n'a donc aucune autre utilité que la pédagogie...
//
//////////////////////////////////////////////////////////////////////////////////////
#include &ltiostream.h>
#include "FormatedString.h"


//////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 12:42:40]
//  Class   : FormatedString
//  Function: FormatedString
//
//  Description: Constructeur par défaut de la classe FormatedString
//
//  Parameters: None
//  Return: None
//
//////////////////////////////////////////////////////////////////////////////
FormatedString::FormatedString( ) : String( )
{
        InitFormat(  );
}

//////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 12:43:24]
//  Class   : FormatedString
//  Function: FormatedString
//
//  Description: Constructeur initialisant de la classe FormatedString,\
//                  connaissant une chaîne de caractères
//
//  Parameters:
//        char* str -  La chaîne de caractères
//        bool bItal -  Italic false/true
//        bool bBold -  Bold false/true
//        short Color -  Couleur (0)
//  Return: None
//
//////////////////////////////////////////////////////////////////////////////
FormatedString::FormatedString( char* str, bool bItal, bool bBold, short Color ) : String( str )
{
        InitFormat( bItal, bBold, Color );
}

//////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 12:45:36]
//  Class   : FormatedString
//  Function: FormatedString
//
//  Description: Constructeur initialisant de la classe FormatedString, \
//                                      connaissant un String
//
//  Parameters:
//        const String & str -  La chaîne
//        bool bItal -  Italic false/true
//        bool bBold -  Bold false/true
//        short Color -  Couleur (0)
//  Return: None
//
//////////////////////////////////////////////////////////////////////////////
```

```
FormatedString::FormatedString( const String & str, bool bItal, bool bBold, short Color ) : String(
str )
{
          InitFormat( bItal, bBold, Color );
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 12:46:31]
//  Class   : FormatedString
//  Function: FormatedString
//
//  Description: Constructeur initialisant de la classe FormatedString, \
//connaissant le formatage
//
//  Parameters:
//        bool bItal -  Italic false/true
//        bool bBold -  Bold false/true
//        short Color -  Couleur (0)
//  Return: None
//
////////////////////////////////////////////////////////////////////////////
FormatedString::FormatedString( bool bItal, bool bBold, short Color ) : String( )
{
          InitFormat( bItal, bBold, Color );
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 12:47:36]
//  Class   : FormatedString
//  Function: FormatedString
//
//  Description: Constructeur par recopie
//
//  Parameters:
//        const FormatedString & fstr -  la variable à recopier
//  Return: None
//
////////////////////////////////////////////////////////////////////////////
FormatedString::FormatedString( const FormatedString & fstr) : String( (String)fstr )
{
          InitFormat( fstr.m_bItalic, fstr.m_bBold, fstr.m_sColor );
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:02:00]
//  Class   : FormatedString
//  Function: ~FormatedString
//
//  Description: Destructeur
//
//  Parameters: None
//  Return: None
//
////////////////////////////////////////////////////////////////////////////
FormatedString::~FormatedString( )
{
          // Nothing to do
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:03:14]
//  Class   : FormatedString
//  Function: InitFormat
//
//  Description: Initialisation des membres privés
//
//  Parameters:
//        bool bItal -  Italic false/true
//        bool bBold -  Bold false/true
//        short Color -  Couleur (0)
//  Return: None
//
////////////////////////////////////////////////////////////////////////////
void FormatedString::InitFormat( bool bItal, bool bBold, short Color )
{
          m_bItalic = bItal;
          m_bBold             = bBold;
          m_sColor  = Color;
}


////////////////////////////////////////////////////////////////////////////
//// Surcharge opérateurs
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:15:04]
//  Class   : FormatedString
//  Function: operator=
//
//  Description: Surcharge opérateur =
//
```

```
//  Parameters:
//        FormatedString & fstr -  l'objet à recopier de type FormatedString
//  Return: FormatedString & -  *this
//
////////////////////////////////////////////////////////////////////////////////
FormatedString & FormatedString::operator=( FormatedString & fstr)
{
          String *strThis, *strParam;

          strThis = this;
          strParam = &fstr;

          *strThis = *strParam;
          InitFormat( fstr.m_bItalic, fstr.m_bBold, fstr.m_sColor );

          return *this;
}

////////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:15:53]
//  Class   : FormatedString
//  Function: operator=
//
//  Description: Surcharge opérateur =
//
//  Parameters:
//        String & str -  l'objet à recopier de type
//  Return: FormatedString & -  *this
//
////////////////////////////////////////////////////////////////////////////////
FormatedString & FormatedString::operator=( String & str )
{
          String *strThis;
          strThis = this;

          *strThis = str;
          InitFormat(  );

          return *this;
}


////////////////////////////////////////////////////////////////////////////////
//// Accès aux variables de formatage
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:18:09]
//  Class   : FormatedString
//  Function: MakeItalic
//
//  Description: Active/désactive l'italic d'une chaîne
//
//  Parameters:
//        bool bItal -  True pour italic, false sinon.
//  Return: None
//
////////////////////////////////////////////////////////////////////////////////
void FormatedString::MakeItalic( bool bItal )
{
          m_bItalic = bItal;
}

////////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:18:50]
//  Class   : FormatedString
//  Function: IsItalic
//
//  Description: Indique si la chaîne est italique ou non
//
//  Parameters: None
//  Return: bool -  True si la chaîne est italique
//
////////////////////////////////////////////////////////////////////////////////
bool FormatedString::IsItalic( )
{
          return m_bItalic;
}

////////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:19:34]
//  Class   : FormatedString
//  Function: MakeBold
//
//  Description: Active/désactive le bold d'une chaîne
//
//  Parameters:
//        bool bBold -  True pour Bold, false sinon
//  Return: None
//
////////////////////////////////////////////////////////////////////////////////
void FormatedString::MakeBold( bool bBold )
{
```

```
          m_bBold = bBold;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:20:39]
//  Class   : FormatedString
//  Function: IsBold
//
//  Description: Indique si la chaîne est bold ou non
//
//  Parameters: None
//  Return: bool -  True si Bold
//
////////////////////////////////////////////////////////////////////////////
bool FormatedString::IsBold( )
{
          return m_bBold;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:21:24]
//  Class   : FormatedString
//  Function: Colorize
//
//  Description: Initialise la couleur de la chaîne
//
//  Parameters:
//          short Color -  la couleur
//  Return: None
//
////////////////////////////////////////////////////////////////////////////
void FormatedString::Colorize( short Color )
{
          m_sColor = Color;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:21:40]
//  Class   : FormatedString
//  Function: GetColor
//
//  Description: Renvoie la couleur actuelle de la chaîne
//
//  Parameters: None
//  Return: short -  La couleur actuelle
//
////////////////////////////////////////////////////////////////////////////
short FormatedString::GetColor( )
{
          return m_sColor;
}

////////////////////////////////////////////////////////////////////////////
//
//  Author  : [Gwenaël Brunet], Created : [14/06/99 13:23:04]
//  Class   : FormatedString
//  Function: Display
//
//  Description: Affichage de la chaîne formatée (le formatage est signalé \
//                                          en utilisant les symboles HTML)
//
//  Parameters:
//          bool bReturn -  Permet de choisir si on veut faire un retour chariot\
//                                                              à la fin de la ligne (true par défaut)
//  Return: None
//
////////////////////////////////////////////////////////////////////////////
void FormatedString::Display( bool bReturn )
{
          // Formatage début
          if( m_bItalic ) cout << "&lti>";
          if( m_bBold ) cout << "&ltb>";
          if( m_sColor ) cout << "&ltfont color=\"#" << m_sColor << "\">";

          // Affichage de la chaîne
          String::Display( false );

          // Formatage fin
          if( m_sColor ) cout << "</font>";
          if( m_bBold ) cout << "</b>";
          if( m_bItalic ) cout << "</i>";

          if( bReturn ) cout << endl;    // Return si nécessaire
}
```

Projet Visual C++ Complet

Voici l'archive qui contient l'ensemble des fichiers relatifs à l'exercice. Il contient par conséquent les codes sources, ainsi que les fichiers d'environnement.

MyProject.zip
(À ouvrir 27,6 Ko)

Table des figures

Figure 1 : Création d'un nouveau projet 5
Figure 2 : choix de type de projet 6
Figure 3 : l'interface de développement Visual C++ 6
Figure 4 : le fichier "main.c" 7
Figure 5 : insérer le fichier dans le projet 8
Figure 6 : ajouter un breakpoint dans le code 10
Figure 7 : un breakpoint mis en place 10
Figure 8 : à gauche, les variables et fonctions en cours d'utilisation, à droite, les variables que l'utilisateur souhaite examiner 11
Figure 9 : Quickwatch d'une variable (ici, "i") 11
Figure 10 : recopie simple d'un objet Vecteur 32