Un tutoriel de C++

Gwenaël Brunet
Gwenael.Brunet@telecom-bretagne.eu
Département Image et Traitement de l'Information
Institut TELECOM/TELECOM Bretagne
MAJ de Novembre 2008

Sommaire

1 Présentation Générale

1.1 Généralités

Puisque cela 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[1].
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. Ces derniers requièrent une façon de travailler plus rigoureuse, pour un code plus structuré, extensible, réutilisable et enfin si possible, portable. Ceci est assez limité lorsque l'on emploie un langage simplement structuré tel que C ou Turbo Pascal.

1.2 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.

1.3 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.

1.3.1 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.

[1] Ce livre est à conseiller essentiellement aux programmeurs C++ avancés, car il est très technique, et en conséquence ne représente pas un moyen très pédagogique pour débuter dans ce langage…

1.3.2 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.

1.3.3 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 ce que l'on nomme un "*cast*" :

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

2 Les bases du C++

2.1 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.

2.1.1 Les commentaires

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

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

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

2.1.2 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 le C++ en ce qui concerne les déclarations et les initialisations des variables.

2.1.3 Référence

Le langage 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 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 "`&`" (*esperluette* en français dans le texte) 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.

2.1.4 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 (en C) */
int * pInt;
pInt = (int*)malloc(1*sizeof(int));
free(pInt);
```

```
...
/* pour un tableau (en C) */
pInt = (int*)malloc(100*sizeof(int));
free(pInt);
...
...
// pour un simple pointeur (en C++)
int * pInt;
pInt = new int;
delete pInt;
...
// pour un tableau (en C++)
pInt = new int[100];
delete []pInt;
...
// Tableau de classes (en C++)
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[2].

2.2 Les entrées/sorties en C++

Le langage C++ dispose de nouvelles routines d'entrées/sorties qui sont plus simples à utiliser. Elles sont définies dans la librairie `iostream`, qu'il est donc nécessaire d'inclure.

Note : nous verrons par la suite certains détails concernant cette librairie qui a évoluée depuis sa création. Notamment, pour certains compilateurs, il sera indispensable de remplacer l'inclusion donnée dans l'exemple ci-dessous (2.2.1), par la suivante :

```
#include <iostream>
using namespace std;
```

Pour l'heure, nous ne donnerons pas d'explication supplémentaire.

De plus, et si vous êtes sous Microsoft Windows, il vous sera peut-être utile d'ajouter à la fin la ligne suivante, afin de vous permettre de voir le résultat affiché avant que votre fenêtre de commande ne se referme :

```
   system("pause");
```

2.2.1 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";
```

[2] A l'origine, il n'était pas obligatoire d'utiliser "delete[]" (mais juste delete) pour détruire des tableaux de type simple. Mais l'usage à rattraper la convention, et désormais, il apparaît que tout le monde utilise cette notation même pour ces tableaux.

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

Note : il est possible que votre compilateur nécessite une valeur de retour pour la fonction main. Dans ce cas, changez juste la ligne par :

```
int main()
```

et rajoutez à la fin :

```
   return 0;
```

Ce programme renvoie 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 (sachant qu'il est bien sûr toujours possible d'utiliser ce caractère d'échappement…).

2.2.2 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.

3 La notion de Classe

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

3.1 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 certaines méthodes sont accessibles, devenant ainsi l'interface. 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 depuis 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`).

3.2 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 fait une méthode de la classe `Point`. Le reste est complètement identique. La seule différence entre ces deux programmes vient du fait qu'on dit que les méthodes du premier programme (dont l'implémentation est faite dans la classe), sont implicitement "`inline`". Ceci signifie que chaque appel à la méthode sera remplacé dans l'exécutable, par le code de 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[3].

[3] En outre, vous verrez par la suite que l'on sépare généralement la déclaration de la classe de son implémentation, dans deux fichiers différents. Or, les méthodes ainsi implémentées nécessitent une recompilation globale lorsqu'on les modifie dans le .h, ce qui peut être gênant à la longue…

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 spécifié 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 par 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();
}
```

3.3 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
}
```

3.4 Les Fonctions membre

3.4.1 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
};
```

3.4.2 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…

3.4.3 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 : le 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;
}
```

4 Construction, destruction et initialisation d'objets

4.1 Constructeur par défaut/initialisant

4.1.1 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,
- une méthode d'écriture d'une valeur dans le vecteur,
- 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…

4.1.2 Solution

```
#include <iostream.h>
class Vecteur
{
     int *pVecteur;
     int nTaille;
public :
     Vecteur(int); // Construction d'après 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;
}
```

4.2 Le pourquoi du comment

Il faut bien comprendre le rôle du destructeur. Un destructeur de classe est appelé – comme son nom l'indique – 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.

4.3 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 y ait 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
     {
          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 (d'ailleurs, le serpent se mordrait la queue sinon…).
- 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[4].
Un petit schéma pour mieux comprendre :

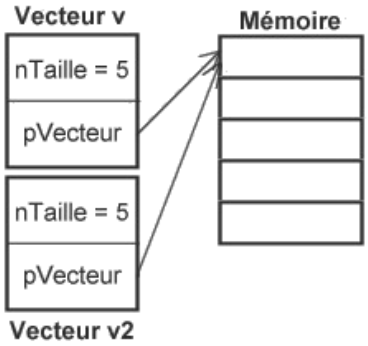

Figure 1 : recopie simple d'un objet Vecteur

On voit clairement qu'après recopie, les membres `pVecteur` de `v` et de `v2` pointent au même endroit.
La parade à cela vient tout naturellement du constructeur par recopie. Au lieu de copier "bêtement" le pointeur, on peut allouer la mémoire et puis recopier les valeurs.

4.4 Mise en oeuvre

Application à la classe `Vecteur` :

```
#include <iostream.h>
class Vecteur
{
     int *pVecteur;
     int nTaille;
public :
     Vecteur(int); // Construction connaissant taille 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;
```

[4] Sauf cas précis et à gérer avec des "pincettes", obligeant généralement l'emploi d'un membre "static", pour compter le nombre d'occurrences pointant sur le segment mémoire.

```
	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.

5 Surdéfinition d'opérateurs

5.1 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 de ces opérateurs.

5.2 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;
}
```

5.3 Opérateur d'affectation

L'opérateur d'affectation "=" représente un cas 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 = new int[nTaille];
}
Vecteur::Vecteur(const Vecteur & v)
{
 nTaille = v.nTaille;
 pVecteur = new int[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 = new int[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.

6 L'Héritage

6.1 Définition et mise en oeuvre

6.1.1 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 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".

6.1.2 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();
}
```

6.2 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 à 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.

6.3 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 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 point, directement[5].

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.

6.4 "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` :

[5] Cela permet de gagner du temps également à l'exécution, par rapport à un appel de fonction par exemple…

```
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 par contre demeurent toujours utilisables pour d'éventuels héritages.

6.5 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 qui abordera certainement cela.

7 Fonctions Virtuelles

7.1 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'initialisation 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.

```
... // Définition de la classe PointCol identique...
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 "générique"
 ppt = &ptc;
 // le pointeur pointe désormais sur un point coloré : légal !
 ppt->Affiche();
}
```

Le résultat peut vous sembler surprenant. En fait, le typage étant effectué statiquement, pour le compilateur, `ppt` reste quoi qu'il advienne un pointeur sur `Point`.
Or, nous n'avions pas encore vu cela, mais il est possible d'affecter une adresse de classe fille à un pointeur de classe de base…[6]
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; }
```

[6] On appelle cela polymorphisme, et permet de passer par exemple un pointeur sur la classe de base, en quelque sorte "générique", lorsqu'on utilise plusieurs classes différentes qui en héritent.

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

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

// PointCol.h
#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 dans ce premier test, il n'en est rien !

7.2 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 cela. 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.

7.3 Remarque

Les fonctions virtuelles permettent ainsi de mettre en œuvre un processus très intéressant de liaison dynamique, en ce sens que la bonne méthode à utiliser est sélectionnée à l'exécution. Cette souplesse est couramment utilisée lorsque l'on se retrouve avec de nombreuses classes héritées, et que l'on passe par exemple en paramètre un pointeur sur la classe mère.
Cette facilité a un coût : elle est très gourmande ! En effet, la liaison dynamique est assez lourde, et il convient d'en disposer avec parcimonie. De ce fait, il est parfois fortement conseillé de concevoir son programme sans utiliser ce processus, par exemple au sein d'un traitement d'image, qui est déjà suffisamment gourmand…

7.4 Identification de type à l'exécution

Pour information, et sans rentrer dans les détails, il est intéressant de savoir que le C++ a introduit au cours de ses évolutions, la possibilité de connaître le type d'une variable

(identification, comparaison), à l'exécution[7]. Cette possibilité fait partie de la fameuse *Librairie Standard*, que nous n'avons pas encore abordée, et qui fera l'objet du chapitre 10.
Voici un exemple très succinct :

```
#include <iostream>
using namespace std;

int n(10);
cout << typeid(n).name() << endl;
int nn(15);
if( typeid(n)==typeid(nn)
      cout << "Meme type !" << endl;
```

La ligne "`using namespace std;`" permet de faire connaître au compilateur cette possibilité.
En bref, `typeid(variable)` renvoie un `id` de type, qui peut être comparé avec l'opérateur classique de comparaison. Il est également possible de connaître le nom du type, avec `typeid(variable).name()`, ou encore, de savoir si une variable est d'un type ascendant à celui d'une autre, à l'aide de `typeid(variable).before()`. Vous comprendrez l'intérêt sous-jacent, notamment en polymorphisme.
Ainsi :

```
#include

class A {
      int n;
};
class B : public A {
      int nn;
};
int main()
{
      A a;
      B b;

      cout << typeid(a).before(typeid(b));

      return 0;
}
```

Nous obtiendrons à l'affichage "`1`", puisque la variable `a` est bien d'une classe mère au type de la variable `b`.
Pour mieux comprendre le fonctionnement de la librairie standard, se référer à la partie 10.
De plus, il faut savoir que cette possibilité est essentiellement utilisée dans le cadre du développement des programmes, par exemple pour le débogage.

[7] Ou R.T.T.I. pour *Run Time Type Identification*, en anglais dans le texte.

8 Exercice préliminaire

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 ce 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 renvoi d'information),
- méthodes permet la gestion de Bold (mise en "`Bold`" et renvoi d'information),
- Colorisation et renvoie de couleur,
- Affichage de la chaîne et des informations de formatage, en utilisation la notation HTML, à savoir :
  1. `<i>` pour la mise en italique et `</i>` à la fin,

2. `<b>` pour la mise en gras et `</b>` à la fin,
3. `<font color="#couleur">` pour la couleur et `</font>` à la fin.

La réalisation de ces classes est assez simple si vous procédez par étape. Utilisez également très allègrement les exemples fournis auparavant. N'hésitez pas 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 !

9 Les templates

9.1 Pourquoi les templates ?

La programmation modulaire permet d'écrire du code en réalisant des "briques" logicielles. Le premier niveau de brique est en quelque sorte les *fonction* et *procédure*.
Ainsi, il est possible de réaliser par exemple une fonction très simple qui retourne la valeur maximale entre deux entiers :

```
const int& Max( const int& a, const int& b )
{
	return a > b ? a : b ;
}
```

N'importe quel programme peut par la suite utiliser cette fonction (pour peu que la visibilité le permette, bien entendu…).

Hélas, comme vous le constaterez, cette modularité possède une limite. En effet, imaginons par exemple qu'il ne s'agit plus de comparer deux entiers, mais deux réels. Il faudra ainsi écrire une autre fonction du style :

```
const double& Max ( const double& a, const double& b )
{
	return a > b ? a : b ;
}
```
[8]

Vous conviendrez que ce genre de limite devient rapidement très contraignant, car il faudrait écrire une fonction pour chaque type de variables (`int`, `float`, `double`, `char`, `unsigned char`, etc.). Pour contourner élégamment cet inconvénient, il faudrait pouvoir définir une sorte de "fonction générique" `Max`, qui puisse prendre n'importe quel type. Cette notion existe, il s'agit des patrons de fonction, ou *template* en C++ dans le texte.

9.2 Patrons de fonction

9.2.1 Ecriture d'un patron de fonction

L'écriture d'un patron de fonction s'effectue de la façon suivante :

```
template <typename T>
const T& Max(const T& a, const T& b)
{
	return a > b ? a : b ;
}
```

Que remarquer :

- `template` : signifie que l'on utilise un patron de fonction.
- `<typename T>` : indique le type générique (`T`) que l'on va utiliser[9]. On aurait pu le désigner par tout autre symbole. Egalement, il est possible de définir plusieurs types différents à ce niveau, si cela est nécessaire, par exemple `<typename T, typename U>`.
- `T` : le type générique noté ci-dessus est ensuite répété aux divers paramètres ou retour de fonction, aux endroits où il est utile, dans notre cas, partout. Mais l'on pourrait

[8] Constatez qu'il est possible de garder le même nom de fonction, Max, puisque les paramètres changent… Le compilateur saura retrouver ses petits sans problème.
[9] Pour information, sachez qu'il est possible de remplacer "`typename`" par "`class`". A l'origine des patrons, il fallait utiliser cette dernière notation, mais elle prêtait à confusion, notamment parce qu'elle laissait supposer que l'on ne pouvait utiliser que des classes, alors que tous les types sont acceptés. Cette notation est néanmoins encore largement utilisée, donc toujours tolérée par le compilateur…

parfaitement mixer des arguments génériques (`T`), avec des non génériques (`int`, chaîne de caractères, classe personnelle, etc.).

Que va-t-il se passer exactement ? Le compilateur connaît la définition générique décrite dans la définition de la fonction. Il va ensuite remplacer chacune des occurrences, par le code adapté au type utilisé dans le programme.
Pour cette raison, la définition et l'implémentation doivent s'effectuer dans le même fichier. Par exemple, si vous souhaitiez déclarer la fonction `Max` dans un fichier interface (`.h`), afin de la réutiliser dans diverses sources, vous êtes obligés de mettre la définition directement à la suite. C'est une limitation parfois gênante des patrons de fonction, mais obligatoire, compte tenu de leur implémentation intrinsèque par le compilateur (et non par l'éditeur de lien !).

9.2.2 Utilisation d'un patron de fonction

L'utilisation d'un patron de classe s'effectue en principe de la façon suivante, concernant notre exemple ci-dessus :

```
cout << Max<int>( 666, 667 ) << endl ;
```

Mais en pratique, sauf lorsque les variables sont de type ambigu, on ne précise pas le type (la notation `<int>`), puisqu'il est implicitement indiqué par les variables elles-mêmes. Ainsi, il est possible d'écrire tout simplement :

```
cout << Max( 666, 667 ) << endl ;[10]
```

Un programme complet implémentant et utilisant notre fonction *template* `Max`, pourrait finalement ressembler à ceci :

```
#include <iostream>

using namespace std;

template <typename T>
const T& Max( const T& a, const T& b )
{
      return a>b?a:b;
}

int main(int argc, char* argv[])
{
      cout << Max<double>( 4, 5.2 ) << endl;
      return 0;
}
```

9.2.3 Paramètres supplémentaires

Il est également possible d'introduire un paramètre dans cette notion de patron. Cela peut être pratique par exemple pour définir un nombre d'éléments donné (et constant…), comme :

```
template<typename N, int taille>
void creerTableau( )
{
      N tab[taille];
      for( int i=0; i<taille; i++ )
            tab[i] = i;
}
```

[10] L'indication de type serait en revanche obligatoire, si en lieu et place des nombres « `666` » et « `667` », nous avions utilisé par exemple « `666` » et « `66.7` », auquel cas, il aurait fallu par exemple préciser qu'il s'agissait de `double`.

Bien entendu, cet exemple n'est pas d'un grand intérêt, du fait que `taille` devra être une constante et seulement une constante, mais il permet de saisir les possibilités offertes par le passage de paramètre.

9.2.4 Spécialisation des fonctions génériques

Il est parfois intéressant, voire indispensable, qu'une fonction générique puisse se comporter de façon différente pour certains types bien spécifiés. Pour reprendre notre fonction `Max`, imaginons que nous lui passons comme paramètres deux chaînes de caractères (`char*`). Que va-t-il se passer ?
En fait, la fonction va bien fonctionner et va réellement comparer les deux paramètres. Sauf que dans ce cas, il s'agit finalement de… pointeurs ! La fonction va par conséquent renvoyer la chaîne de caractères dont le pointeur est le plus grand. Ce n'est pas vraiment l'effet escompté…
Or donc, il paraît nécessaire de préciser un comportement spécial dans le cas où l'on utilise des chaînes de caractères. On appelle cette technique "*spécialisation*", et elle s'effectue de la façon suivante :

```
// Fonction générique
template <typename T>
T& Max( T& a, T& b )
{
      return a>b?a:b;
}
// Fonction spécialisée
template<>
char*& Max( char*& a, char*& b )
{
      int i=0;
      while( a[i]==b[i] && a[i]!='\0' )
            i++;
      return a[i]>b[i]?a:b;
}
```

Ainsi, lorsque nous appellerons la fonction `Max` avec des paramètres de type chaîne de caractères, le programme utilisera non pas la fonction générique, mais la deuxième. Vous pouvez le vérifier en rajoutant un `cout`, par exemple, dans les deux fonctions.

9.2.5 Exercices

Exercice n°1 : Sur le même principe, réaliser une fonction générique qui additionne deux nombres de tout type et qui rend le résultat.

Exercice n°2 : Ensuite, programmer une fonction générique qui effectue l'addition de deux nombres de types différents, et qui rend un double par défaut.

Exercice n°3 : Spécialiser enfin la première fonction d'addition, pour additionner deux chaînes de caractères (concaténation).

9.3 Patrons de classe

9.3.1 Ecriture d'un patron de classe

Nous venons de voir qu'il est possible de réaliser des fonctions génériques, afin d'implémenter une gestion multi-types. Cette possibilité est bien sûr étendue aux classes.
Imaginons le cas d'une classe qui implémente un objet de type *vecteur*. Il est plus qu'intéressant de pouvoir réaliser une classe générique, qui permettra de créer des objets utilisant des types différents, car il est fréquent d'utiliser des vecteurs d'entiers, de réels, voire de classes.
Une classe générique est définie d'une façon similaire à une fonction générique. Dans notre cas de vecteur, voici ce que cela peut donner :

```
template<typename T>
class vecteur
{
      ...
      T* m_vecteur ; // Déclaration de la donnée vecteur
      ...
} ;
```

Les constructeurs et destructeurs ne subiront pas de changement pour l'instant. Nous pouvons les ajouter, ainsi que d'autres méthodes de lecture/écriture :

```
template<typename T>
class vecteur
{
      T* m_vecteur;
      unsigned int m_taille;
public:
      vecteur( unsigned int taille )
      {
            m_taille = taille;
            m_vecteur = new T[m_taille];
      }
      ~vecteur( ) { delete []m_vecteur; }
      unsigned int taille( ) { return m_taille; }

      T litA(unsigned int i)
      {
            if( i<m_taille )
                  return m_vecteur[i];
            else
                  return (T)0;[11]
      }
      void ecritA(unsigned int i, T n)
      {
            if( i<m_taille )
                  m_vecteur[i] = n;
      }
};
```

On peut bien entendu placer l'implémentation des méthodes à l'extérieur de la déclaration de la classe[12]. Le formalisme à utiliser est alors le suivant, en prenant par exemple la méthode `ecritA` :

[11] Dans ce cas précis, nous pouvons remarquer qu'il s'agit d'une gestion simplifiée d'une erreur critique. En effet, que rendre lorsque l'indice est supérieur à la taille du vecteur ? 0 ? La solution n'est pas satisfaisante. Pour information, il faut savoir que l'on gère ces erreurs critiques avec des exceptions, cf. 10.3.
[12] Attention toutefois à ce que l'implémentation soit toujours dans le même cadre de visibilité pour le compilateur (même fichier en général) que la déclaration, comme expliqué auparavant.

```
template<typename T>void vecteur<T>::ecritA(unsigned int i, T n)
{
	if( i<m_taille )
		m vecteur[i] = n;
}
```

9.3.2 Utilisation d'un patron de classe

L'utilisation d'une classe générique est similaire à l'utilisation d'une fonction générique. Il suffit de déclarer un objet du type de la classe, en spécifiant le type désiré :

```
vecteur<int> v( 10 );
```

Dans l'exemple ci-dessus, nous déclarons un vecteur d'entiers (de taille 10, passée en paramètre de constructeur).
Par la suite, l'utilisation de la variable v sera identique à celle de n'importe quelle autre variable objet, par exemple :

```
for( unsigned int i=0; i<v.taille(); i++ )
	v.ecritA(i,i);
```

9.3.3 Paramètres supplémentaires

Identiquement aux fonctions génériques, il est possible de rajouter des paramètres supplémentaires dans la définition du *template* de la classe générique. Ainsi, dans notre classe `vecteur`, il peut sembler intéressant d'ajouter directement la taille du vecteur *a priori*, directement dans cette définition. Bien sûr, la limitation fait que cette taille doit être une constante, ce qui peut fortement en limiter l'intérêt.
Voici un exemple, dans lequel vous constaterez notamment que l'on peut retirer la variable de taille incluse dans la classe précédente, puisqu'elle est désormais définie dans les paramètres du *template* :

```
template<typename T, unsigned int m_taille>
class vecteur
{
	T m_vecteur[m_taille];
public:
	unsigned int taille( ) { return m_taille; }
	T litA(unsigned int i)
	{
		i<m_taille ? return m_vecteur[i] : return (T)0;
	}
	void ecritA(unsigned int i, T n);
};

template<typename T, unsigned int t>
void vecteur<T, t>::ecritA(unsigned int i, T n)
{
	if( i<m_taille )
		m_vecteur[i] = n;
}
```

La taille étant connue a priori, et étant donc une constante, il est possible d'allouer le vecteur directement dans sa déclaration. De ce fait, on peut également faire disparaître les constructeur et destructeur, devenus inutiles.
L'utilisation de cette nouvelle classe s'effectue simplement de la façon suivante :

```
	vecteur<int,10> monvecteur;
	for( unsigned int i=0; i<monvecteur.taille(); i++ )
		monvecteur.ecritA(i,i);
```

Dans ce cas, lorsque nous déclarerons une variable vecteur<char> et que nous utiliserons la méthode affiche, le programme utilisera celle qui a été définie pour le type char.

9.3.4 Spécialisation d'un patron de classe

Une classe générique peut également être spécialisée. Il est possible de la spécialiser complètement (spécialisation de l'ensemble de la classe), ou partiellement (spécialisation de fonctions membres génériques).
Par exemple, on peut demander à notre classe `vecteur` de se comporter différemment lorsque le vecteur en question gère des caractères, auquel cas, on peut supposer qu'il s'agit d'une chaîne de caractères, et que l'on préfère le gérer comme tel (la raison d'être n'est bien évidemment pas immédiate…). Cela nous donnerait à partir de notre premier exemple :

```
template<>
class vecteur<char>
{
      char* m_vecteur ;
      ...
}
```

Cette possibilité ne sera pas détaillée ici, car il s'agit d'une pratique somme toute peu courante.

9.3.5 Spécialisation de fonctions membres

A l'instar d'une fonction isolée, il est possible de spécialiser une fonction membre d'une classe générique.
Imaginons que nous ajoutions à notre classe `vecteur` d'origine, une fonction `affiche`, qui imprime à l'écran l'ensemble des éléments du vecteur :

```
template<typename T>
void vecteur<T>::affiche( )
{
      for( int i=0; i<m_taille; i++ )
            cout << m_vecteur[i] << endl;
}
```

Il peut être intéressant de spécialiser cette fonction, afin de simplement imprimer la chaîne de caractères, lorsque le type est du `char`. Nous obtenons ainsi :

```
template<typename T>
class vecteur
{
      T* m_vecteur;
      unsigned int m_taille;
public:
      vecteur( unsigned int taille )
      {
            m_taille = taille;
            m_vecteur = new T[m_taille];
      }
      ~vecteur( ) { delete []m_vecteur; }
      unsigned int taille( ) { return m_taille; }

      T litA(unsigned int i)
      {
            if( i<m_taille )
                  return m_vecteur[i];
            else
                  return (T)0;
      }
      void ecritA(unsigned int i, T n)
      {
```

```
            if( i<m_taille )
                  m_vecteur[i] = n;
      }
      void affiche( );
};
// méthode générique
template<typename T>
void vecteur<T>::affiche( )
{
      for( int i=0; i<m_taille; i++ )
            cout << m_vecteur[i] << endl;
}
// méthode spécialisée
void vecteur<char>::affiche( )
{
      char* buffer = new char[m_taille+1];
      for( int i=0; i<m_taille; i++ )
            buffer[i] = m_vecteur[i];
      buffer[m_taille] = '\0'; // caractère de fin de chaîne
      cout << buffer << endl;
      delete []buffer;
}
```

9.3.6 Exercices

Exercice n°1 : Concevoir une classe qui gère une matrice d'éléments de type générique. Implémenter les méthodes suivantes : constructeur par défaut, constructeur connaissant le nombre de lignes et colonnes, constructeur par recopie, destructeur, accès à un élément en lecture et écriture, opérateur d'affectation (copie d'une matrice complète ou bien juste un élément). Spécialiser une méthode `affiche`, qui imprime sous forme de matrice les éléments, ou sous forme de texte, lorsqu'il s'agit d'un type `char`.

Exercice n°2 : Effectuer les modifications de conception, en prenant en compte le passage des dimensions directement dans les paramètres du *template*.

10 La librairie standard

Dans l'ancienne norme du C++ ANSI, et contrairement à d'autres langages comme Java, il n'y avait rien pour simplifier des manipulations aussi triviales que celles portant sur les chaînes de caractères, ou encore les vecteurs, les listes, etc.[13] Pourtant, il est bien entendu très fréquent d'utiliser ces types !
La librairie standard comble cette lacune et bien d'autres, en étendant la STL, pour *Standard Template Library*, qui ne représentait à l'origine que la compilation des classes conteneurs[14]. Il s'agit d'une librairie utilisable par tous les programmeurs C++ (norme ANSI), offrant des outils puissants pour la gestion de chaînes de caractères par exemple, mais bien au-delà, de conteneurs génériques, d'itérateurs (*iterators*), d'algorithmes variés, etc.
De facto, ce chapitre ne se veut absolument pas exhaustif, dans la mesure où la librairie standard pourrait très bien faire l'objet d'un cours très long en elle-même.

10.1 Un premier exemple

La librairie standard impliquant l'utilisation d'un certain nombre de notations ou de concepts nouveaux et importants, il est intéressant de commencer par un exemple pour imager l'ensemble :

```
#include <iostream>
#include <string>

using namespace std;

// --------- MAIN --------------
int main(int argc, char* argv[])
{
      string str("hello world !");
      cout << str << endl;
      return 0;
}
```

Rien que dans ce code, il y a beaucoup de détails à expliquer :

- les **entêtes** (`include`) ont changé,
- utilisation de "`using namespace std;`", qui est un **espace de nommage**.

10.1.1 Les entêtes

Tout d'abord, nous pouvons expliquer le passage de `#include <iostream.h>` à `#include <iostream>`, que vous avez peut-être constaté dans le chapitre précédent, concernant les *template*. Sans entrer dans les détails, "`iostream`" est la version librairie standard de "`iostream.h`". En conséquence, dès lors que vous utilisez la librairie standard, il faut changer le header associé.
Dans cet exemple, la fonction `cout` connaît le type `string`. Si vous laissez l'ancienne entête, vous aurez des erreurs à la compilation, car seuls les types simples (`int`, `char`, etc.) sont connus.

10.1.2 Les espaces de nommage

Les espaces de nommage représentent à eux seuls un ajout très important. En effet, jusqu'alors, le C++ ne gérait pas la notion de paquetage, à l'instar de langages tels que Java. Il est pourtant relativement naturel de vouloir regrouper un certain nombre d'objets dans un même sous-ensemble.

[13] De ce fait, beaucoup d'erreurs communes provenaient de cette lacune, car dans le cas des chaînes de caractères par exemple, il est quasiment obligatoire de manipuler des pointeurs.
[14] Du fait de la célébrité de la STL, beaucoup de personne nomme abusivement la librairie standard, STL.

Les espaces de nommage – *namespace* en anglais – permettent de réaliser enfin cela, en offrant la possibilité d'empaqueter un certain nombre de classes, de variables, de constantes et de types. Nous n'aborderons pas en longueur ce sujet, mais voici un exemple des possibilités offertes :

```
namespace mesOutils {
      const int monOutil = 13;
      class A {
      };
      class B {
      };
}

class A {
};
```

En bref, il est possible de définir un peu tout ce que l'on souhaite dans cet espace. Une fois le paquetage défini, il est possible d'accéder à un élément de plusieurs manières :

- `mesOutils::A` : syntaxe identique à une classe.
- `using namespace mesOutils` : fait connaître à l'ensemble du bloc, l'ensemble du paquetage.
- `using mesOutils::A` : fait connaître à l'ensemble du bloc, la classe A.
- `namespace MO = mesOutils` : alias de `mesOutils` en `MO`. Permet de gagner du temps de programmation, lorsqu'un nom de paquetage est trop long.

Un commentaire s'impose : l'utilisation de la première syntaxe est probablement la meilleure qui soit, dans la mesure où elle permet d'éviter les collisions de noms. Car les `namespace` représentent aussi une façon d'éviter ces fréquentes collisions (combien de classes `string` existe-t-il par exemple ?). Il faut donc savoir utiliser parcimonieusement le `using namespace`.
Outre cela, il est intéressant de savoir qu'il est possible d'imbriquer les espaces de nommage, et d'ainsi de créer toute une hiérarchie d'espaces. Par la suite, l'utilisation est à faire simplement tel que :

```
mesOutils::graphisme::maClasseCourbe maCourbe;
```

Il est à noter enfin, qu'un ensemble de classes présentes dans un espace de travail, n'ont pas plus de droit d'accès aux données des unes et des autres, que d'autres classes, contrairement à Java. C'est un manque d'un certain point de vue, mais d'un autre, c'est également une façon de protéger les différentes boîtes noires qui ne sont pas forcément liées autrement que dans un concept abstrait supérieur à l'objet.

10.2 La classe string

Il n'y a rien de plus laborieux que la gestion des chaînes de caractères en C/C++ : c'est lourd, fastidieux, et la plupart des développeurs ont écrit une structure, une classe, qui les gère une bonne fois pour toute.
Mais depuis quelques années déjà, il est offert la possibilité d'utiliser la classe `string` de la librairie standard, elle-même basée sur la classe générique `basic_string`, tout comme `wstring` qui offre les mêmes services, mais avec un format de codage de caractères sur 16bits au lieu de 8.

10.2.1 Déclaration, construction

Il existe plusieurs façons de construire une `string`, par exemple :

```
string str1;
string str2("Un petit exemple...");
string str3( str2 );
string str4( 256, '\0' );
char chaine[] = "Encore un exemple...";
string str5( chaine );
```

```
string str6( chaine, 5 );
string str7( str3, 2, 5 );
```

En voici les explications :

- `str1` crée une chaîne vide, non initialisée, d'une taille indéfinie (en général, et *a priori*, cette dernière serait de 20 caractères, par défaut…).
- `str2` crée une chaîne initialisée à "`Un petit exemple...`".
- `str3` construit une chaîne par recopie de `str2`.
- `str4` construit une chaîne de 256 caractères nuls ('`\0`') ;
- `str5` s'initialise avec le contenu de la variable `char[] chaine`.
- `str6` recopie les cinq premiers caractères de `chaine`.
- `str7` se construit avec les cinq caractères suivant la deuxième position de `str3`.

Veillez à bien comprendre qu'il y a une différence entre la longueur d'une chaîne, et sa capacité. Il est bien entendu possible de redimensionner une chaîne, mais comme toujours, cette opération a un coup non négligeable. Par conséquence, il est souvent préférable de fixer une taille au départ (comme dans `str4`), et de s'en tenir par la suite à cette limite.

10.2.2 Opérations possibles

Comme vous vous en douterez, de nombreuses opérations sur les chaînes de caractères sont possibles. Nous ne les listerons pas ici, mais en voici quelques exemples :

```
string str( "Chaudes larmes..." );
const char *temp = str.data(); // récupération des données
char c = str[10];              // récupération d'un caractère
char c2 = str.at(12);          // récupération d'un caractère
cout << str.size() << endl;    // taille de la chaîne
cout << str.capacity() << endl; // capacité de la chaîne
str.reserve(256);              // augmentation de la capacité
cout << str.capacity() << endl; // capacité de la chaîne
str += " Et au plaisir !";     // concaténation
cout << str << endl;
```

10.3 La gestion des flux : lecture/écriture dans un fichier

La librairie standard propose des outils puissants de gestion des flux (entrée/sortie). Nous avons déjà vu dans le chapitre 2.2 les fonctions `cout` et `cin`. Bien entendu, C++ propose d'autres outils, par exemple pour la gestion des lecture/écriture sur le disque.

10.3.1 La classe fstream

Nous avions vu que la sortie écran et l'entrée clavier avaient été fortement simplifiées, en comparaison des lourdes et inélégantes fonctions C. Il en est de même des fonctions de lecture/écriture sur disque.

La classe `fstream` gère ainsi toutes ces fonctionnalités nécessaires. En voici un premier exemple :

```
#include <fstream>
#include <iostream>

int main()
{
      fstream in( "ReadMe.txt", ios::in );
      if( !in.is_open() )
      {
            cout << "Fichier non trouve !" << endl;
            return 1;
      }

      char c;
```

```
        in >> c;
        cout << c;

        return 0;
}
```

Sans entrer pour l'instant dans les détails, la classe `fstream` est utilisée ici pour ouvrir un fichier en lecture (`ios::in`), et après vérification d'ouverture, pour lire un caractère à l'aide de l'opérateur de flux habituel ">>" (il ne s'agit bien sûr pas de l'unique façon de lire le contenu d'un fichier, texte ou binaire).

10.3.2 ifstream et ofstream

Il existe deux classes qu'il est pratique d'utiliser en lieu et place de (et basées sur) `fstream` : `ifstream` et `ofstream`, la première servant dans le cas de fichier ouvert en lecture, et la seconde dans le cas de fichier ouvert en écriture.
Conceptuellement parlant, elles offrent un attrait certain, dans la mesure où il est général connu *a priori*, si une partie de programme doit lire ou écrire un contenu depuis/dans un fichier. En outre, elles empêchent des erreurs, puisqu'elles ne permettent pas d'écrire lorsqu'il s'agit de lecture, et inversement.
Dans les exemples suivants, nous étudierons les possibilités offertes par `fstream`, mais également `ifstream` ou/et `ofstream`.

10.3.3 Ouverture

L'ouverture d'un fichier peut s'effectuer directement dans le constructeur, ou bien en utilisant la méthode `open` :

```
        fstream entree;
        entree.open( "Unfichier", ios::in | ios::binary );
        fstream sortie( "UnAutreFichier", ios::out );

        ifstream entree2( "UnFichier", ios::binary );
        ofstream sortie2( "UnAutreFichier" );
```

La variable `entree` est un flux ouvert en entrée – lecture de fichier – binaire. `sortie`, quant à elle, est en écriture et par défaut en texte. Ensuite, `entree2` ouvre un fichier binaire en lecture, et `sortie2`, un fichier texte en écriture.
Dans la liste des options disponibles, vous pourrez compter les *flags* suivants :

Flag	Fonction
`ios::in`	Ouverture d'un fichier en lecture. Peut s'utiliser avec `ofstream`, pour éviter la troncation du fichier.
`ios::out`	Ouverture d'un fichier en écriture. Lorsque utilisé avec `ofstream`, sans `ios::app`, `ios::ate` ou `ios::in`, `ios::trunc` est appliqué.
`ios::app`	Ouverture en ajout en fin de fichier (*append*).
`ios::ate`	Ouverture d'un fichier existant, et recherche de la fin.
`ios::nocreate`	Ouverture d'un fichier uniquement s'il existe (sinon, erreur).
`ios::noreplace`	Ouverture d'un fichier uniquement s'il n'existe pas (sinon, erreur).
`ios::trunc`	Ouverture du fichier avec effacement du contenu.
`ios::binary`	Ouverture du fichier en binaire (défaut texte).

10.3.4 Lecture

```
// déclaration d'un objet ifstream pour lecture
ifstream in( "ReadMe.txt" );
if( !in.is_open() )
{     // fichier ouvert ?
      cout << "Fichier non trouve !" << endl;
      return 1;
}

// Lecture d'un caractère
char c;
in >> c;
cout << c;

const int szMax(1024);
char str[szMax];

// Lecture d'une ligne
in.get( str, szMax );
// Lecture d'un caractère
c = in.get( );

// Lecture de n lignes et affichage
while( in.getline(str, szMax) )
      cout << str << endl;

// Lecture de 50 caractères
in.read( str, 50 );
```

Il existe plusieurs façons d'accéder aux données d'un fichier. Pour commencer, il faut noter qu'une ouverture de fichier, par défaut, est en mode texte. Il ne faut par conséquent pas oublier de rajouter le *flag* `ios::binary` lorsque cela est nécessaire.

- L'opérateur >> : permet de lire une variable dans le fichier. La lecture dépend du type de la variable passée en paramètre. Par exemple, s'il s'agit d'un entier, la lecture renverra un entier, que l'ouverture ait été en mode texte ou binaire (pratique pour lire des résultats de calculs…).
- `get` : lit un caractère (sans paramètre) ou n caractères, jusqu'à ce que le délimiteur '\n' soit trouvé, ou bien que la taille passée en paramètre soit atteinte ou la fin du fichier. Aucun caractère de fin de chaîne n'est ajouté.
- `getline` : extrait n caractères, jusqu'à ce que le délimiteur '\n' soit trouvé, ou bien que la taille passée en paramètre soit atteinte ou la fin du fichier. Le caractère de fin de chaîne '\0' est ajouté.
- `read` : permet de lire n octets bruts. Typiquement utilisé pour les lectures de données binaires (images, etc.).

10.3.5 Ecriture

```
// déclaration d'un objet ofstream pour écriture
ofstream out( "test.txt" );

out.put( 'c' );   // écriture d'un caractère

// écriture de différentes variables
out << "Chaudes larmes, et au plaisir...";
out << 456 ;

// écriture de données brutes
int n(123);
float d(12.5);
out.write( (char*)&n, sizeof(int) );
out.write( (char*)&d, sizeof(float) );
```

Presque symétriquement à la lecture, il existe également différentes méthodes d'écriture dans un fichier :

- L'opérateur << : écrit des données de types variables.
- `put` : écrit un caractère.
- `write` : écrit des données brutes dans le fichier. Nécessite de passer un `const char*`, ce qui implique généralement un *cast*.

10.4 Les exceptions

Nous avons constaté dans divers exemples, que la gestion des erreurs est parfois difficile, voire impossible, lorsque l'on utilise des moyens conventionnels, tels que renvoi d'un entier codant l'erreur, affection d'une valeur d'erreur à une variable, etc.
Cette gestion d'erreurs est possible de façon très efficace, en utilisant ce que l'on nomme les exceptions.

10.4.1 Principes fondamentaux

Une exception est une interruption de l'exécution d'un programme, lors d'un événement donné. Le mécanisme est assez simple : il est basé sur la notion d'envoi et d'écoute/réception. Ainsi, dans un programme correctement conçu, on "essaie" de faire une action. Lorsqu'une erreur est générée par cette action, une exception est lancée et ne demande plus qu'à être captée puis traitée comme il se doit.
Nous pouvons observer la syntaxe simplifiée de mise en œuvre suivante :

```
try {
      ... // action susceptible de générer une exception
}
catch (une sorte d'exception) {
      ... // traite l'exception
}
```

Que peut-on "jeter" ? En fait, tout type d'objet. On peut choisir par exemple de créer une classe qui va gérer proprement des erreurs, ou bien, tout simplement, on peut opter pour l'envoi d'un entier que l'on traitera dans la foulée.

```
// Fonction à contrôler
double divise( int a, int b ) throw(int)
{
      if( b==0 )
            throw 1; // erreur de type division par zéro
      else if( a==b )
            throw 2; // un autre exemple
      return (double)a/b;
}

...

// Utilisation
double d;
try { // on essaie de diviser...
      d = divise(2,2);
}
catch(int res)
{     // on traite les erreurs
      if( res==1 )
            cerr << "Division par zero... :(" << endl;
      else if( res==2 )
            cerr << "Pas besoin de calculer cela...;)" << endl;
      d = 0.0f;
```

```
}
cout << d << endl;
```

10.4.2 Exceptions et librairie standard

La librairie standard implémente un certain nombre d'exceptions utiles dans son espace de nommage `std`. Elles héritent toutes de la classe de base `exception`, qui implémente une méthode `what`, que l'on utilise par la suite pour gérer l'erreur.
Voyons un exemple concret :

```
#include <exception>
#include <stdexcept>

...
void faitRien( int n ) throw(std::out_of_range)
{
  if( n==0 )
    throw std::out_of_range("Division par zéro...");
  cout << 123456/n << endl;
}

int main(int argc, char* argv[])
{
  try {
    faitRien( 0 );
  }
  catch( std::out_of_range &e )
  {
    cerr << e.what() << endl;
  }
  return 0;
}
```

Vous remarquerez tout d'abord la manière d'implémenter la gestion des exceptions dans la méthode souhaitée. Rien de bien sorcier *a priori*. Il est cependant utile de présenter les différents types d'exception implémentés dans la librairie :

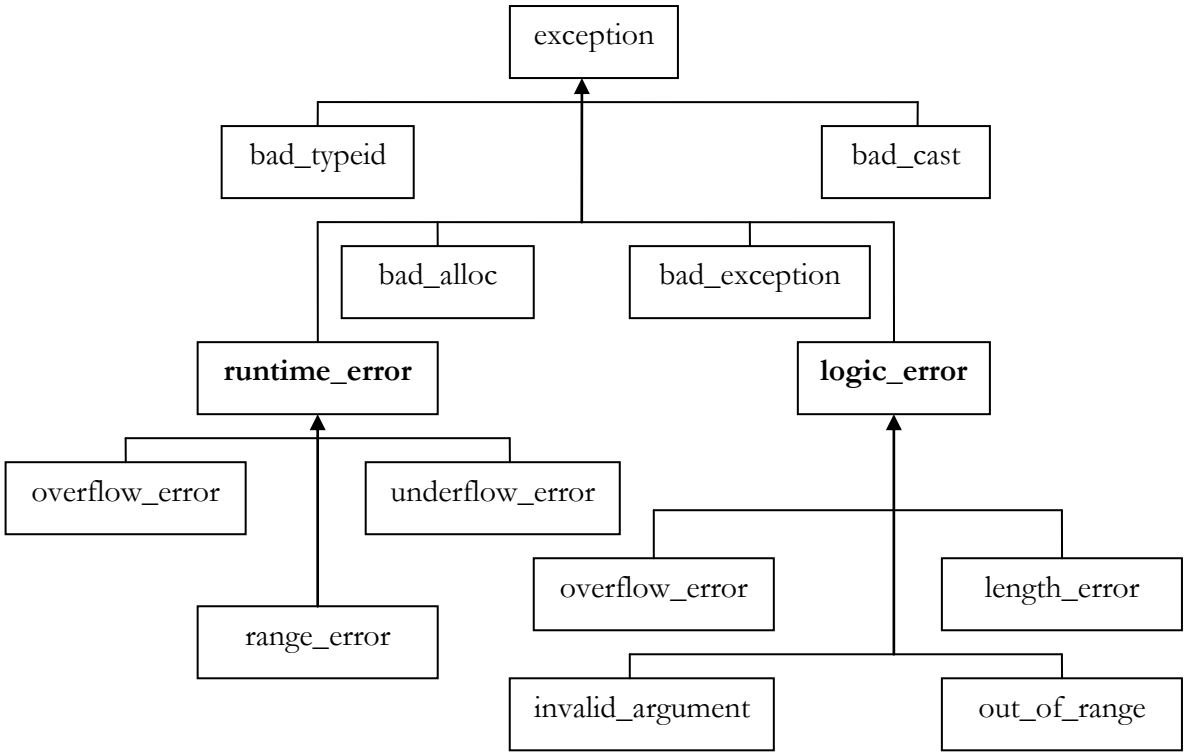

Figure 2 : Hiérarchie des exceptions disponibles dans la librairie standard

Vous pourrez constater que cet arbre possède deux branches principales : `runtime_error` et `logic_error`. La première permet de gérer des erreurs d'exécution, c'est-à-dire des exceptions qui sont créées lors de l'exécution du programme. La seconde, en revanche, est plus intime au développement, et permet de gérer des erreurs qui ne sont sensées se produire que lors de la phase de programmation.
Bien entendu, personne n'étant parfait, il est possible que certaines de ces dernières exceptions soient lancées en cours de déploiement d'un logiciel. De ce fait, il est préférable de les laisser et de gérer correctement ces exceptions, qui sont toujours utiles pour un débogage post-production.

10.5 Les conteneurs

La librairie standard implémente un ensemble de structures de données couramment utilisées, ainsi que des algorithmes s'appliquant à elles. Il s'agit des classes conteneur.
Il existe trois grandes catégories de conteneurs :

- **les séquences élémentaires** (*vector, list, deque*),
- **les utilisations spécialisées des séquences élémentaires** (*queue, stack*),
- **les conteneurs associatifs** (*set, map*).

Nous ne verrons pas l'ensemble des possibilités dans ce tutoriel, mais quelques exemples d'utilisation de ces désormais indispensables outils.

10.5.1 La classe vector

La classe `vector` permet de gérer une donnée qui s'apparente à un tableau. En outre, elle possède des caractéristiques similaires, telles que l'indexation initiale à 0.
Exemple d'utilisation :

```
#include <iostream>
#include <vector>

using namespace std;
```

```
int main()
{
	vector<int> monVecteur( 5 ); // vecteur de 5 éléments
	for( int i=0; i<monVecteur.size(); i++ )
		monVecteur[i] = i;

	for( i=0; i<monVecteur.size(); i++ )
		cout << monVecteur[i] << endl;

	const int nAReserver(10);

	monVecteur.reserve( nAReserver ); // réservation
	cout << monVecteur.capacity() << " "
		<< monVecteur.size() << endl;

	monVecteur[5] = 5;
	cout << monVecteur.capacity() << " "
		<< monVecteur.size() << endl;

	monVecteur.resize(6); // redimensionnement
	monVecteur[5] = 5;
	cout << monVecteur.capacity() << " "
		<< monVecteur.size() << endl;

	monVecteur.front() = -1; // accès au premier élément
	monVecteur.back() = -2; // accès au dernier élément
	monVecteur.push_back(28); // ajoute élément fin de liste
	monVecteur.pop_back( ); // retire élément fin de liste

	for( i=0; i<monVecteur.size(); i++ )
		cout << monVecteur[i] << endl;

	return 0;
}
```

En outre, nous insisterons sur la différence existant entre la réservation d'espace et le redimensionnement du vecteur. Ainsi, la méthode `reserve` permet de réserver une capacité de stockage. Mais la taille utilisée s'en trouve inchangée, tant que l'on ne redimensionne pas le vecteur explicitement, avec une méthode telle que `resize` ou `push_back`.
Quant au reste, le fonctionnement est assez simple. Vous devez savoir que ce type de structure est généralement parcouru à l'aide d'*itérateur* (`iterator`). Nous verrons cela dans un paragraphe suivant (10.5.3).

10.5.2 La classe list

La classe `list` implémente une gestion de liste chaînée classique. Elle possède en grande partie les mêmes fonctionnalités que la classe `vector`, à quelques détails près bien entendu. Par exemple, il est possible de détruire un élément à n'importe quelle place, ou encore, ne possède pas d'opérateur crochets (`operator[]`). Enfin, par essence même, un objet défini par cette structure prendra plus de place en mémoire que son équivalent utilisant la classe `vector`.
Exemple d'utilisation :

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
	list<int> maListe( 5 ); // liste de 5 éléments

	maListe.front() = -1;   // accès au premier élément
	maListe.back() = -2;    // accès au dernier élément
	maListe.push_back( 28 ); // ajoute élément en queue
	maListe.pop_back( ); // retire élément en queue
```

```
	return 0;
}
```

10.5.3 Les "itérateurs"

Les *itérateurs* sont parmi les éléments les plus importants des conteneurs, dans la mesure où ils sont utilisés en permanence lorsqu'on manipule ces dernières, que ce soit en lecture ou en écriture.
Pour faire simple, un *itérateur* est une sorte de pointeur sur un élément d'un conteneur, liste ou vecteur par exemple. Car, s'il demeure toujours possible d'accéder aux données des conteneurs par une indexation classique dans certains cas, vous verrez à l'usage qu'il est impossible ou pour le moins malaisé, pour certaines opérations telles que les insertions.

```
	vector<int> v( 10 );

	vector<int>::iterator it;

	for( it=v.begin(); it!=v.end(); it++ )
		*it = 10;

	for( it=v.begin(); it!=v.end(); it++ )
		cout << *it << " ";
```

Vous constaterez que l'utilisation des *itérateurs* est réellement proche de celle des pointeurs. Pour accéder aux données, il suffit par exemple d'employer l'opérateur `*`.
Vous remarquerez également la notation pour la déclaration de l'*itérateur* :

```
	vector<int>::iterator it;
```

Vous imaginez ainsi aisément qu'il devient rapidement lourd d'utiliser ces notations, d'autant plus lorsqu'on préfère ne pas utiliser le `using namespace`, auquel nous obtenons :

```
	std::vector<int> v ;
	std::vector<int>::iterator it ;
```

Dans ce cas, il est fortement conseillé de définir des alias, comme :

```
	typedef std::vector<int> vectInt;
	typedef vectInt::iterator vecIntIt;

	vectInt v;
	vectIntIt it;
```

Le programme s'en retrouve allégé et bien plus lisible.
Voici maintenant un exemple d'utilisation de l'insertion, et de parcours inversé (*reverse iterator*) :

```
	srand((unsigned)time(NULL));

	vector<int> v( 10 );

	vector<int>::iterator it;
	vector<int>::reverse_iterator rit; // itérateur inversé

	// parcours inversé
	for( rit=v.rbegin(); rit!=v.rend(); rit++ )
		*rit = (int)((double)rand()/RAND_MAX*10);

	// fonction très utile de recherche du premier "5"
	// et d'insertion d'un "-1" juste avant
	bool trouve=false;
	for( it=v.begin(); it !=v.end() && trouve==false; it++ )
	{
		if( *it==5 )
		{
			v.insert(it,-1);
			trouve = true;
```

```
            }
      }

      for( it=v.begin(); it!=v.end(); it++ )
            cout << *it << " ";
```

Vous pourrez remarquer ici que le parcours inversé ne nécessite pas de mettre en œuvre un réel parcours inverse "à la main", notamment en partant de la fin et en décrémentant l'*itérateur*, ce qui demeure d'ailleurs l'intérêt de cette possibilité.
Pour explication, il est obligatoire de quitter la boucle après l'insertion, sinon le programme ne retrouve plus ses petits dans la mesure où le conteneur a changé de séquençage.

10.5.4 Ouvertures

Nous n'irons pas plus loin dans notre quête de la connaissance concernant la librairie standard. Il est néanmoins utile de connaître certaines autres fonctionnalités, afin que vous sachiez qu'elles existent en cas de besoin.

- **`deque`** : les DQ représentent un compromis intéressant entre la liste et le vecteur, c'est-à-dire qu'elle est efficace lors d'ajout et de suppression en début et fin de liste, offre la possibilité d'indexation et enfin, les insertions en milieu de liste sont plus rapides que pour un vecteur.
- **`queue`** : les files gèrent une sorte de liste "premier arrivé, premier traité". Ainsi, il est seulement possible de traiter les éléments par les extrémités (entrée en fin, retrait en début).
- **`stack`** : la pile est un peu l'inverse, dans la mesure où le dernier entré est le premier traité.
- **`priority_queue`** : les files à priorité permettent de gérer des tas dont les éléments sont "classés" par priorité. En fait, plus de classement, il s'agit d'accès. Ainsi, un seul élément est accessible : celui dont la priorité est la plus élevée.
- **`map` & `set`** : ces conteneurs sont particuliers, dans la mesure où ils permettent une indexation par rapport à une clef, telle une base de données (lorsqu'une clef correspond à plusieurs éléments, on utilise `multiset` et `multimap`). La différence entre ces deux concepts réside dans le fait que la `map` possède deux entrées (clef, valeur), là où le `set` n'en a qu'une (en fait la clef elle-même).
- **Autres itérateurs** : il existe un certain nombre d'*itérateurs* intéressants à connaître. Par exemple, la STL permet l'utilisation d'*itérateurs* sur des flux (sortie écran, flux de fichier). Ainsi, il est possible "d'entasser" des informations, pour les écrire (ou lire) en différé.

10.6 Exercices

Exercice n°1 : Ecrire une classe simple qui gère une image comme un vecteur de vecteurs génériques (ces derniers vecteurs étant les lignes) en utilisant le conteneur `vector`. Inclure constructeurs (par défaut, initialisant, par recopie), destructeur, opérateurs utiles (affectation, addition), accès aux données et affichage (à l'écran, sous forme de texte).

Exercice n°2 : Ajouter la lecture et l'écriture dans un fichier.

Exercice n°3 : Ajouter une gestion d'exceptions, par exemple lorsque l'accès à un pixel est impossible, un fichier introuvable, etc.

11 Bibliographie

[Bru02] Gwenaël Brunet, *Tutorial de C++*. GET/ENST de Bretagne, 2002.

[Cas02] Christian Casteyde, *Cours de C/C++*. http://c.developpez.com/cours/, 2002.

[Del04] Claude Delannoy. *Programmer en langage C++*. Eyrolles, 2004.

[Eck99] Bruce Eckel. *Thinking in C++, 2nd Edition*. MindView Inc., http://www.mindview.net/, 1999.

[Gar99] Bruno Garcia, *La librairie standard du C++*. ISIMA, http://c.developpez.com/cours/, 1999.