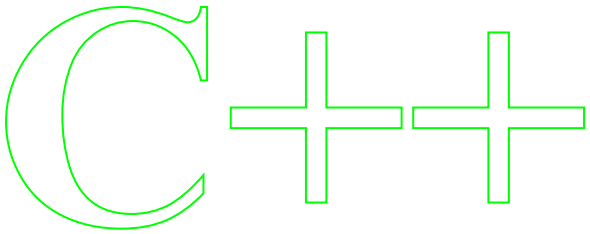
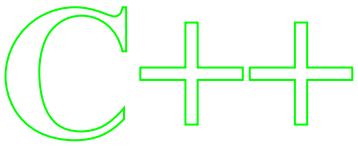
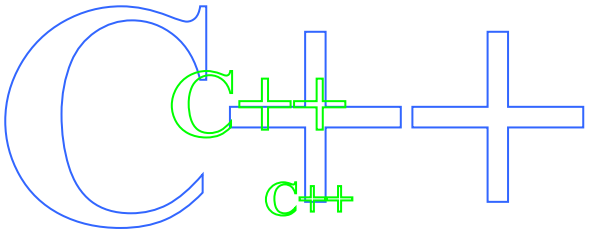
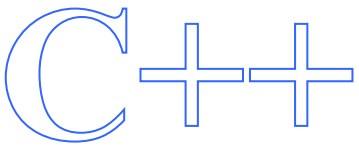
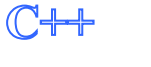








… en passant par



C

Molliet J.-P.

Partie 2

Enumérés, surcharge d'opérateurs et définition de types

Introduction

Dans ce chapitre charnière, nous introduisons le type **enum** ainsi que les principes de base de la surcharge des opérateurs et la définition de type **typedef**.

Le type enum

Les types énumérés seront nos premiers types définis par l'utilisateur. A dire vrai, ils n'apportent que peu de possibilités nouvelles au langage si ce n'est éventuellement d'en améliorer la lisibilité dans certaines circonstances. Très proches des entiers auxquels on pouvait totalement les assimiler en C, la différence entre les 2 notions s'avère à peine plus subtile en C++.

La forme générale d'une déclaration **enum**:[1]

```
enum [ nom_de_type ]
{     identificateur [ = expression ]
     [ , identificateur [ = expression ] ... ]
} [ identificateur [ , ... ] ];
```

Le *nom_de_type* est facultatif, il n'a de sens que:

1. Pour la lisibilité (compréhension).
2. Si l'on désire par la suite déclarer d'autres objets de ce même type.

Les *identificateurs* entre les { } représentent les valeurs du type, ses constantes. L'ordre dans lequel on les énumère fixe la structure d'ordre dans le type énuméré et la valeur que représentent ses constantes. Par défaut, la première constante correspond à 0, la deuxième à 1 et ainsi de suite. Toutefois, l'expression constante facultative qui peut suivre un identificateur de la

[1] Comme nous n'avons pas encore utilisé une telle notation, signalons que les [ ] ne font pas partie de la syntaxe. Ils désignent des parties optionnelles, de même les … indiquent que la partie qui précède peut être répétée.

liste, permet de changer ces valeurs par défaut.

Finalement, le ou les éventuels *identificateurs* (facultatifs) qui peuvent suivre l'accolade fermante, représentent une ou des variables que l'on déclare de ce type. Ils ne font pas réellement partie de la déclaration de type elle-même.

Quelques exemples commentés:

```
enum saison { printemps, ete, automne, hiver };
```

Ici, nous définissons le type sans déclarer d'objet (variable) de ce type et avec ses valeurs par défaut, donc *printemps* correspond à 0, *ete* à 1, etc.

Nous pouvons maintenant déclarer des variables de notre type:

```
enum saison uneSaison, uneAutreSaison;
```

Notez qu'en C nous devons pour l'instant remettre le mot réservé **enum**. Ici nous avons déclaré 2 variables, *uneSaison* et *uneAutreSaison* toutes deux du type *saison*.

En C++ nous pouvons remettre **enum** (compatibilité oblige!), mais nous n'y sommes pas obligés[1]. Nous pouvons aussi écrire la déclaration des 2 variables ci-dessus sous la forme:

```
saison uneSaison, uneAutreSaison;
```

Dans les instructions du programme nous pouvons affecter à une variable, une constante du type:

```
uneSaison = ete;
```

ou une autre variable du même type:

```
uneSaison = uneAutreSaison;
```

Si les compilateurs C permettent un mélange presque sans restriction entre **int** et **enum**, il n'en va pas tout à fait de même en C++ dont nous allons dans la suite décrire les possibilités.

Nous pouvons toujours affecter à une variable entière une valeur d'un type **enum**.

L'opération ne posant jamais de problème (pas de risque d'erreur) le compilateur génère une conversion implicite. En sens inverse, il y a le risque d'affecter une valeur ne correspondant à rien de défini dans le type **enum**. Pour réaliser une telle opération le programmeur doit effectuer une conversion explicite:

```
uneSaison = (saison) 1;
```

[1] Nous ne rappellerons pas cette différence à chaque fois et nous ne donnerons par la suite que la forme C++.

Rappelons l'autre forme possible mais uniquement valable en C++:

```
uneSaison = saison (1);
```

Cette conversion ne garantit en aucun cas que la valeur affectée correspond effectivement à quelque chose de raisonnable pour le type **enum**, mais on espère qu'alors le développeur a bien réfléchit à ce qu'il écrit. Théoriquement, nous ne pouvons affecter une valeur entière à une variable **enum** que pour autant qu'elle ne dépasse pas la puissance de 2 la plus proche de sa plus grande valeur[1].

Les constantes d'énumération peuvent s'utiliser partout où une constante entière le pourrait, par exemple pour définir la dimension d'un tableau:

```
saison tab [ automne ] = { ete, automne };
```
[2]

Nous pouvons appliquer les opérateurs arithmétiques de base entre valeurs d'un type énuméré, entre un énuméré et un entier, et même (mais est-ce bien raisonnable?) entre des valeurs de 2 types énumérés différents. Le résultat d'une telle opération sera toujours un entier qu'il faudra convertir explicitement si nous désirons l'affecter à une variable du type énuméré:[3]

```
uneSaison = ( enum saison ) ( uneSaison + 1 ); --
```
[4]

Nous pouvons changer les valeurs par défaut des constantes d'énumération ou de certaines d'entre elles, par exemple:

```
enum saison { printemps, ete = 5, automne, hiver };
```

Ici *printemps* correspond à 0, *ete* à 5, *automne* à 6 et *hiver* à 7. Dès que l'on ne précise plus une valeur, celle attribuée par défaut correspond à 1 de plus que celle qui précède.

Attention donc, si nous réalisons une boucle du genre:

```
for ( enum saison i = printemps; i <= hiver;
          i = ( enum saison ) (i+1) ) …  --
```
[5]

[1] Ceci pour des valeurs positives. Il faut faire le raisonnement symétrique pour des valeurs négatives. En pratique on constate généralement que les compilateurs laissent faire n'importe quoi!

[2] Même si cet exemple n'est en lui-même pas très recommandable!

[3] Même si en C cette conversion explicite n'est pas nécessaire, nous recommandons de la faire pour des raisons de lisibilité!

[4] En C++, simplement: `une_saison = saison ( une_saison + 1 );`

[5] En C++, simplement: `for ( saison i = printemps; i <= hiver; i = saison (i+1) ) …`

cette boucle va s'exécuter 8 fois et prendre toutes les valeurs successives 0 à 7!!!

Remarques complémentaires (mais qu'il ne faudrait pas mettre en pratique!):

- Les valeurs ne doivent pas nécessairement être données dans l'ordre.
- Rien ne nous interdit de donner à 2 éléments différents du type énuméré la même valeur!!!

A propos des boucles, rappelons que les variables peuvent se déclarer localement dans la boucle, comme nous l'avons fait ci-dessus.

En fait, les seules opérations raisonnables sur ces types énumérés sont l'affectation et le passage en paramètre. Restent toutes les "magouilles" que l'on peut imaginer en les combinant avec d'autres types et en leur appliquant des opérateurs en principe bizarres pour eux.

Dans notre exemple, nous aurions pu déclarer les variables en même temps que la définition du type:

```
enum saison { printemps, ete, automne, hiver }
        uneSaison, uneAutreSaison;
```

Si dans le reste de l'unité, nous n'utilisons plus ce type *saison*, nous pouvons aussi écrire:

```
enum { printemps, ete, automne, hiver }
          uneSaison, uneAutreSaison;
```

Autre exemple:

```
enum temperature
  { froid = 0, tiede = 20, un_peu_plus, chaud = 30 };
```

Souvent le type **enum** n'est utilisé que pour définir des constantes. Cela représente une alternative intéressante à un *#define*, et dans ce cas le nom du type ne présente aucun intérêt.

❑ Énumérations fortement typées (C++11)

Comme nous venons de le voir, les constantes d'énumération sont traitées plus ou moins comme des entiers, et permettent au programmeur de faire bon nombre d'erreurs que le compilateur ne peut pas détecter. Avec les énumérations fortement typées, la norme C++11

introduit des énumérations qui offrent une sûreté de typage.

Pour déclarer une énumération fortement typée on ajoute le mot-clé **class** après **enum** :

```
enum class Orientation { Nord, Est, Sud, Ouest } ;
```

Par la suite, pour utiliser une constante de l'énumération, il faut spécifier le nom de l'énumération et le nom de la constante, séparés par l'opérateur de résolution de portée :

```
Orientation orientation = Orientation::Sud ;
```

Parce qu'on doit spécifier le nom de l'énumération il n'y a plus de collisions de nommage entre constantes venant d'énumérations différentes. Dans l'exemple suivant, le même nom apparaît deux fois sans problème :

```
enum class AlignementHorizontal {
    Gauche,
    Centre,
    Droite
};
enum class AlignementVertical {
    Haut,
    Centre,
    Bas
};
```

Jusqu'à maintenant nous avons parlé de différences de syntaxe. Passons aux similitudes. Tout comme les énumérations normales, on peut changer la valeur des constantes si c'est nécessaire :

```
enum class IPProtcolNumber {
    ICMP = 1,
    IGMP, // = 2
    TCP = 6,
    UDP = 17
};
```

Revenons aux différences. La différence la plus importante, comparé aux énumérations normales, est le typage. Les valeurs d'une énumération fortement typée ne sont pas implicitement converties vers des entiers. La comparaison entre une variable et un entier ne compilera pas :

```
if (orientation == 2) … // Faux !
```

On est tenu d'utiliser les constantes de l'énumération :

```
if (orientation == Orientation::Sud) …
```

Le code suivant ne compilera pas non plus :

```
cout << orientation << endl ;
```

Il faut explicitement convertir la valeur en entier :

```
cout << (int)orientation << endl ;
```

Le fait que les énumérations et les entiers sont de types différents permet d'assurer une certaine sécurité en codant.

❑ *Introduction à la surcharge des opérateurs*

La notion de surcharge des opérateurs, uniquement valable en C++, ne possède pas de liens spécifiques avec les types énumérés. Toutefois, il nous fallait disposer d'un type défini par l'utilisateur pour introduire cette technique. Nous nous contenterons ici d'introduire les éléments de base, sur lesquels nous serons amenés à revenir plus en détails par la suite, principalement dans le contexte des classes.

Dans les boucles qui précèdent vous avez peut-être été surpris par la manière dont nous avons incrémenté notre variable de boucle:

```
for ( saison i = printemps; i <= hiver;
        i = saison ( i+1 ) ) …
```

Si nous disposons des opérateurs arithmétiques de base par conversion implicite vers les entiers, il n'en va pas de même des autres opérateurs qui ne sont pas disponibles. Pour les types qu'il se crée, le programmeur a la possibilité de définir (surcharger) la fonctionnalité qu'il désire pour tous les opérateurs à l'exception de:

:: , . , .* , ?: , **sizeof**, **typeid**, **static_cast**, **dynamic_cast**, **const_cast**, **reinterpret_cast** [1]

La surcharge des opérateurs ne présente pas de réels problèmes. Toutefois, certains d'entre eux nécessitent des traitements quelque peu particuliers. Dans cette optique, nous présenterons ici la situation spécifique du ++ et de <<.

En fait, un opérateur réalise la même tâche qu'une fonction; il nous faut donc définir sous une forme un peu particulière une fonction pour l'opérateur à surcharger. Le prototype d'une telle fonction prend la forme générale suivante:

```
type operator OP ( paramètres )
```

type:	celui du résultat de l'opération.
operator:	le mot réservé, c'est lui qui précise qu'il s'agit d'une fonction opérateur.
OP:	l'opérateur désiré.
paramètres:	les opérandes de l'opérateur.

[1] La "," est là pour séparer les opérateurs, il ne s'agit pas de l'opérateur "," lui-même!

Un exemple de déclaration:

```
saison operator + ( saison , saison );
```

L'utilisateur ne peut se redéfinir que les opérateurs existants.

Pour pouvoir surcharger un opérateur il faut qu'il possède au moins un paramètre d'un type défini par l'utilisateur.

L'opérateur garde le niveau de priorité qu'il possède dans la définition du langage.

L'opérateur doit conserver le nombre de paramètres qu'il possède dans la définition du langage.

Voici un exemple, pas très utile, mais qui démontre de manière simple ce que l'on peut faire. Tout d'abord le prototype de la fonction opérateur:

```
saison  operator + ( const saison , const int );
```

et ensuite sa définition proprement dite:

```
saison  operator + ( const saison  v1, const int v2 )
{
  return  saison  ( int ( v1 ) + v2 );
}
```

Notons simplement la présence du **const** devant les paramètres, pas indispensable mais raisonnable. Elle nous permet, entre autres, d'utiliser des constantes comme opérandes. De plus nous transmettons ces paramètres par valeur mais nous aurions aussi pu le faire par référence:

```
saison  operator + ( const saison & , const int & );
```

Le compilateur ne fait aucune hypothèse sur la propriété de commutativité des opérateurs. Si nous voulons aussi pouvoir "additionner" un **int** avec un objet *saison*, nous devons encore surcharger l'opérateur de la manière suivante:

```
saison  operator + ( const int, const saison );
```

Rappel: il faut au minimum un paramètre d'un type utilisateur pour surcharger un opérateur. Nous pouvons aussi en avoir 2:

```
saison  operator * ( const saison, const saison);
```

Revenons à des situations un peu plus réalistes, notre opérateur ++ qui nous permettrait d'incrémenter plus facilement notre variable de boucle.

Là aussi, si nous voulons indifféremment pouvoir écrire `uneSaison++` ou `++uneSaison` nous tombons alors sur un problème supplémentaire: comment distinguer les 2 surcharges? Avant la version 3 de la norme seule la préincrémentation était possible. Le problème a été résolu en ajoutant un paramètre **int** supplémentaire totalement artificiel et inutilisé pour la version post incrémentation. Voici les 2 versions de cette définition de l'opérateur:

```
/* Version pre incrementation */
saison operator ++ ( saison & val )
{
  return val = saison ( val + 1 );
}
/* Version post incrementation */
saison operator ++ ( saison & val, int bidon )
{
  saison temp = val;
  val = saison ( val + 1 );
  return temp;
}
```

Relevez qu'ici nous n'avons pas le choix, le paramètre doit être passé par référence puisqu'il subit une modification. Dans le cas de la préincrémentation le résultat de la fonction pourrait lui aussi être fourni par référence!

Notez que si vous avez aussi surchargé l'addition entre votre type *saison* et le type **int**, c'est elle qui sera appelée dans la fonction ci-dessus.

Et les entrées/sorties, nous n'en avons pas encore parlé! N'oubliez pas que pour *cin* >> et << pour *cout* représentent aussi des opérateurs. Si pour << nous pouvons éventuellement laisser aller, il y a conversion implicite et affichage de la valeur entière correspondante, il n'en va pas de même pour >>. Pour l'utiliser avec des valeurs d'un type énuméré (par la suite pour n'importe quel type défini par l'utilisateur) nous devons impérativement le surcharger. Voici par exemple ce que nous pouvons imaginer:

```
/* Surcharge de l'opérateur >> pour le type saison */
istream & operator >> ( istream & entree, saison & val )
{
  int tempo;
  entree >> tempo;
  val = saison (tempo);
  return entree;
}
```

Les règles à respecter pour redéfinir l'opérateur >>:

- Le premier paramètre consiste toujours en une référence à un flux d'entrée, soit un objet de type *istream* (type mis indirectement à disposition par *iostream*).
- Le deuxième paramètre: une référence à l'objet auquel affecter la valeur lue.
- L'opérateur doit livrer en retour une référence sur le flux qu'il a reçu comme premier paramètre, ce qui nous permet dans une même instruction *cin* d'enchaîner plusieurs lectures.
- En pratique, ce que nous ne faisons pas dans notre exemple, il faudrait encore gérer les exceptions!

Les mêmes remarques que ci-dessus restent valables pour la redéfinition de l'opérateur <<, mais en remplaçant *istream* par *ostream*. De plus le deuxième paramètre peut se transmettre par valeur puisqu'il s'agit simplement de l'afficher.

Voilà, nous n'en dirons pas plus pour l'instant sur la surcharge des opérateurs, mais comme précisé dès le départ nous aurons à y revenir par la suite. Donnons maintenant un exemple utilisant ces possibilités; nous y avons quelque peu "compliqué" les entrées/sorties pour illustrer d'autres fonctionnalités connues du langage.

```
/*
  Exemple: Utilisation de types enumeres, surcharge d'operateur
    ENUMERES
*/
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;
/* Definition d'un type enumere */
enum chiffre { zero, un, deux, trois, quatre };
string chiffreChaine [] ={"zero", "un", "deux", "trois", "quatre"};

/* Surcharges de l'opérateur ++ pour le type chiffre */
chiffre  operator ++ ( chiffre & );
chiffre  operator ++ ( chiffre &, int );
/* Surcharge de l'opérateur << pour le type chiffre */
ostream & operator << ( ostream & , const chiffre );
/* Surcharge de l'opérateur >> pour le type chiffre */
istream & operator >> ( istream & , chiffre & );
/* Surcharge de l'opérateur + entre un chiffre et un int */
chiffre  operator + ( const chiffre, const int );
```

```
int main ( )
{
  /* Declaration des variables de travail de notre type enumere */
  /* L'initialisation est la pour l'exemple, mais inutile */
  chiffre v1 = deux;
  cout << "Exemple d'enumere\n\n";
  cout << " Donnez un chiffre: ";
  cin >> v1;
  /* Notre operateur + */
  cout << v1 << " + 1 = " << v1 + 1 << endl;
  /* Ci-dessous, resultat entier, pas notre operateur */
  cout << "1 + v1 = " << 1 + v1 << endl;
  /* Ce que l'on devrait faire sans notre operateur */
  cout << "int ( " << v1 << " ) + 1 = "
       << chiffre ( int (v1) + 1 ) << endl;
  /* Exemple de boucle */
  for (  chiffre v = zero; v <= quatre; v++ )
    cout << v << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}


/* Surcharges de l'opérateur ++ pour le type chiffre   */
/* Forme de l'expression: pour ne pas utiliser notre + */
chiffre  operator ++ ( chiffre & val )
{
  return val = chiffre ( int (val) + 1 );
}
chiffre  operator ++ ( chiffre & val, const int bidon )
{
  chiffre temp = val;
  val = chiffre ( int (val) + 1 );
  return temp;
}


/* Surcharge de l'opérateur << pour le type chiffre */
ostream & operator << ( ostream & sortie, const chiffre val )
{
  return sortie << chiffreChaine [val];
}
```

```
/* Surcharge de l'opérateur >> pour le type chiffre */
istream & operator >> ( istream & entree, chiffre & val )
{
  string tempo;
  entree >> tempo;
  for (  val = zero; val <= quatre;  ++val )
    if ( chiffreChaine [val] == tempo ) break;
  return entree;
}


/* Surcharge de l'opérateur + entre un chiffre et un int */
chiffre  operator + ( const chiffre v1, const int v2 )
{ cout << "***"; // Pour montrer que c'est le notre!!
  return  chiffre ( int (v1) + v2 );
}
```

Un exemple d'exécution:

```
Exemple d'enumere

 Donnez un chiffre: deux
***deux + 1 = trois
1 + v1 = 3
int ( deux ) + 1 = trois
zero
un
deux
trois
quatre

Fin du programme...Appuyez sur une touche pour continuer...
```

Du point de vue d'une application réelle, ce programme "monolithique" n'est pas raisonnable. Le but a priori de la définition d'un type par l'utilisateur consiste à mettre à disposition des outils pour manipuler les objets de ce type.

Il nous faut, comme nous l'avons déjà indiqué une fois, découper l'application:

- Un fichier d'inclusion comportant la définition du type et les prototypes des fonctions mises à disposition. Notez que les définitions des opérateurs << et >> font référence pour leurs paramètres à *iostream*, nous devons donc inclure ce fichier.
- Un deuxième fichier contenant les corps des fonctions mises à disposition. En général nous ne fournirons à nos clients qu'une version objet de ce fichier.

- Finalement le fichier comprenant l'application proprement dite, soit le programme principal.

En pratique la structure d'une application se révélera certainement bien plus complexe que cela, mais la démarche générale restera la même.

```
#ifndef FICHIER
#define FICHIER
/* Fichier d'inclusion pour les types enumeres,
    Le type chiffre et quelques operations de base
      ENUM/ENUM.h
*/
#include <iostream>
using namespace std;
/* Definition d'un type enumere */
enum chiffre { zero, un, deux, trois, quatre };

/* Surcharges de l'opérateur ++ pour le type chiffre */
chiffre  operator ++ ( chiffre & );
chiffre  operator ++ ( chiffre &, int );
/* Surcharge de l'opérateur << pour le type chiffre */
ostream & operator << ( ostream & , const chiffre );
/* Surcharge de l'opérateur >> pour le type chiffre */
istream & operator >> ( istream & , chiffre & );
/* Surcharge de l'opérateur + entre un chiffre et un int */
chiffre  operator + ( const chiffre, const int );
#endif

/* Le type chiffre et quelques operation de base
      ENUM/ENUM.cpp
*/
#include "enum.h"
#include <iostream>
#include <string>
using namespace std;
string chiffreChaine [] ={"zero", "un", "deux", "trois", "quatre"};
/* Surcharges de l'opérateur ++ pour le type chiffre   */
/* Forme de l'expression: pour ne pas utiliser notre + */
chiffre  operator ++ ( chiffre & val )
{
  return val = chiffre ( int (val) + 1 );
}
chiffre  operator ++ ( chiffre & val, const int bidon )
{
  chiffre temp = val;
  val = chiffre ( int (val) + 1 );
  return temp;
}
```

```
/* Surcharge de l'opérateur << pour le type chiffre */
ostream & operator << ( ostream & sortie, const chiffre val )
{
  return sortie << chiffreChaine [val];
}
/* Surcharge de l'opérateur >> pour le type chiffre */
istream & operator >> ( istream & entree, chiffre & val )
{
  string tempo;
  entree >> tempo;
  for (  val = zero; val <= quatre;  ++val )
    if ( chiffreChaine [val] == tempo ) break;
  return entree;
}

/* Surcharge de l'opérateur + entre un chiffre et un int */
chiffre  operator + ( const chiffre v1, const int v2 )
{ cout << "***";  // Pour montrer que c'est le notre!
  return  chiffre ( int (v1) + v2 );
}
/*
  Exemple: Utilisation de types enumeres, definition de types
    ENUM/ENUMERES
*/
#include "enum.h"
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  /* Declaration des variables de travail de notre type enumere */
  /* L'initialisation est la pour l'exemple, mais inutile */
  chiffre v1 = deux;
  cout << "Exemple d'enumere\n\n";
  cout << " Donnez un chiffre: ";
  cin >> v1;
  /* Notre operateur + */
  cout << v1 << " + 1 = " << v1 + 1 << endl;
  /* Ci-dessous, resultat entier, pas notre operateur */
  cout << "1 + v1 = " << 1 + v1 << endl;
  /* Ce que l'on devrait faire sans notre operateur */
  cout << "int ( " << v1 << " ) + 1 = "
       << chiffre ( int (v1) + 1 ) << endl;
  /* Exemple de boucle */
  for (  chiffre v = zero; v <= quatre; v++ )
    cout << v << endl;
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

❑ *Définition de type: typedef*

Cette fonctionnalité, bien qu'encore agréable dans certains cas, perd de son importance en C++ par rapport à C. En C, nous venons de le voir, nous devons répéter "**enum** type … "[1] chaque fois que nous déclarons une variable de ce type, chose possible mais non nécessaire en C++.[2]

Forme simple:

```
typedef type NOM_DE_TYPE;
```

- **typedef** est le mot réservé introduisant une telle définition.
- *type* est la description (au sens usuel du terme en C/C++) du type que l'on veut définir.
- *NOM_DE_TYPE* est le nom (identificateur) donné au type que l'on crée; l'usage, sans que cela soit une obligation, veut qu'on l'écrive en majuscules.

Le *type* peut être un type simple, par exemple dans le but de créer un synonyme:

```
typedef int ENTIER;
```

Ou une abréviation:

```
typedef unsigned int UINT;
```

Mais aussi un type plus complexe:

```
typedef enum { vrai, faux } BOOLEEN;
```

Un type ainsi défini s'utilise comme n'importe quel type de base pour déclarer des variables.

Ainsi, avec nos exemples ci-dessus, nous pouvons écrire:

```
ENTIER valeur;  // Idem a int valeur
```

Ou avec une initialisation:

```
UINT max = 127;
```

[1] Il en sera de même dans le prochain chapitre pour les types structures, unions et champs de bits.

[2] Le seul moyen d'alléger les écritures en C consiste à passer par une définition de type.

Ou encore avec une liste de déclarations d'objets:

```
BOOLEEN b1, b2, b3;
```

Ce que nous venons d'introduire sur des types simples ou sur le type **enum**, s'applique de manière similaire avec tous les autres types que nous étudierons dans la suite. Malheureusement, par exemple pour le type **enum**, cela ne lui ajoute aucune propriété spécifique.

Le type ainsi défini n'est qu'un synonyme représentant le même type que celui sur lequel il se construit et non pas un nouveau type; cela signifie qu'ils restent compatibles et peuvent se "mélanger" dans des expressions. De plus les opérateurs **sizeof** et (*cast*) donnent le même résultat, qu'on les applique au type de base ou au type "dérivé".

Avec nos exemples:

```
sizeof ( ENTIER )     est équivalent à     sizeof ( int )
```

De même on utilise indifféremment l'un ou l'autre pour fixer le type d'un paramètre formel.

Les attributs **const** et **volatile** peuvent servir à fixer les caractéristiques (mais pas les classes de mémorisation qui elles ne s'appliquent qu'aux variables):

```
typedef const int ENTIER;
```

Attention, la notion de signe fait entièrement partie de la définition d'un type et ne peut pas se modifier par la suite lors de la déclaration d'objets. Ainsi, avec la déclaration:

```
typedef int ENTIER;
```

nous ne pouvons **pas** par la suite écrire:

```
unsigned ENTIER I;        // Faux!!!
```

Une déclaration de type, sous n'importe quelle forme, est toujours locale à un module et donc non exportable. Si l'on désire pouvoir disposer d'une telle déclaration dans plusieurs modules, il faudra obligatoirement passer par un fichier d'inclusion.

Remarques complémentaires:

Il n'est pas évident de bien comprendre ce que signifie une telle déclaration de type!

Plutôt que de longues et ténébreuses explications, des exemples devraient nous aider dans cette compréhension:

```
typedef  int *PT;
```

PT est alors un type pointeur sur un entier, nous pouvons donc déclarer:

```
PT p;
```

qui revient au même que de déclarer:

```
int *p;
```

Plusieurs définitions peuvent se condenser dans une même déclaration:

```
typedef  int ENTIER, *PT, TAB [ 5 ];
```

qui correspond à:

```
typedef  int ENTIER;
typedef  int *PT;
typedef  int TAB [ 5 ];
```

ENTIER représente un simple type entier signé, *PT* un type pointeur sur un entier signé et *TAB* un type tableau de 5 entiers signés.

Une définition de type peut en utiliser une autre déclarée au préalable:

```
typedef  int ENTIER;
...
typedef ENTIER *PT;
```

Autre exemple:

```
typedef double (*PT)(double);
```

PT correspond à un type pointeur sur une fonction qui a un paramètre de type **double** et qui livre un résultat également du type **double**; nous pouvons donc déclarer:

```
PT f;
```

et, dans le reste du programme, par exemple:

```
double x;
…
f = sin;
cout << f(x) << endl;
```

Un cas particulier se présente pour les tableaux, nous pouvons définir un type sans donner le nombre d'éléments du tableau:

```
typedef float TAB [ ];
```

Mais son utilisation ne peut se faire que sous des conditions particulières:

- Pour déclarer des variables, avec initialisation obligatoire via un agréagat (ce qui en fixe implicitement la taille):

```
TAB tableau = { 0, 1, 2, 3, 4 };
```

- Pour fixer le type d'un paramètre formel:

```
float fct ( TAB );
```

Dans l'exemple de programme que nous vous avons donné pour illustrer les types énumérés nous aurions pu écrire par exemple:

```
typedef enum { zero, un, deux, trois, quatre } chiffre;// [1]
```

et ne rien changer d'autre dans le reste du programme.

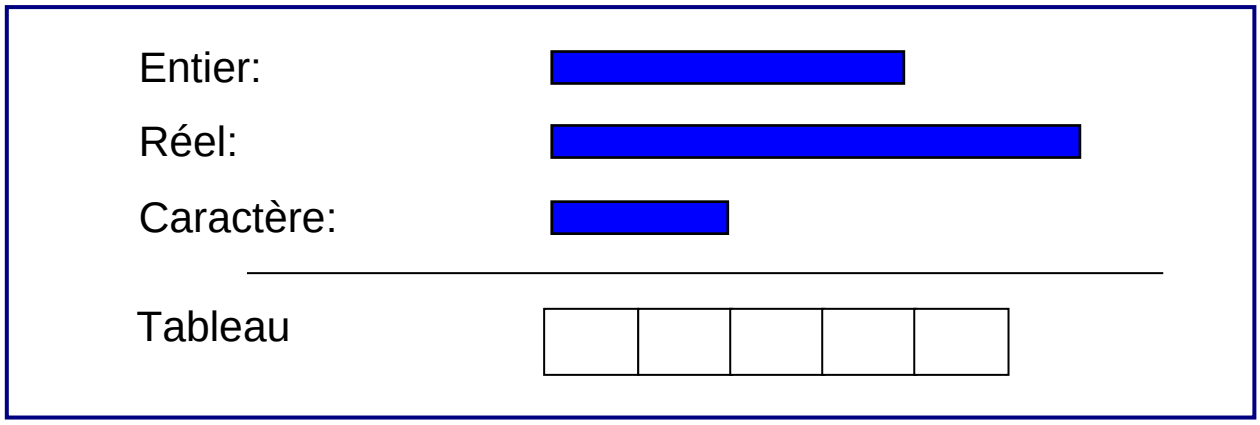
Voilà, nous espérons que ces quelques exemples vous facilitent la compréhension du mécanisme et que vous l'utiliserez dans certaines situations pour rendre vos programmes plus lisibles!

[1] Ici, pour des raisons pratiques, nous ne respectons pas la convention qui veut que le nom d'un type s'écrive en majuscules!

Structures, unions et champs de bits

Les types structures: struct

Jusqu'à présent, à part les tableaux qui représentaient notre premier type structuré, nous n'avons manipulé dans nos différents programmes que des types simples et des tableaux:

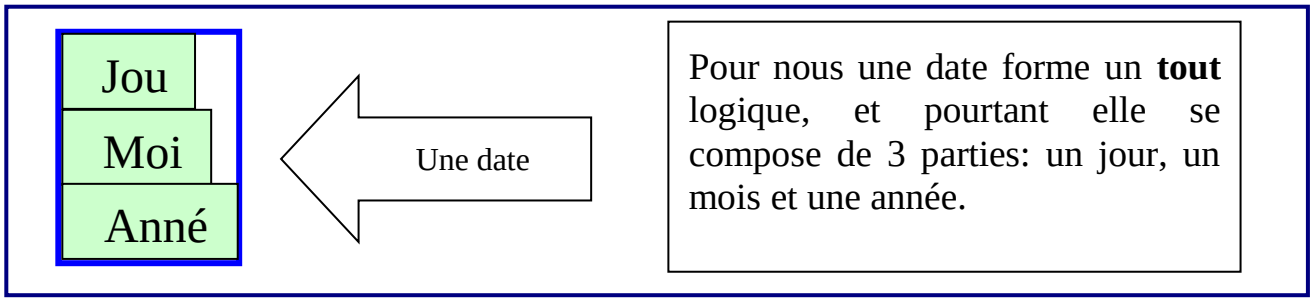


Rappelons au passage qu'un type définit:

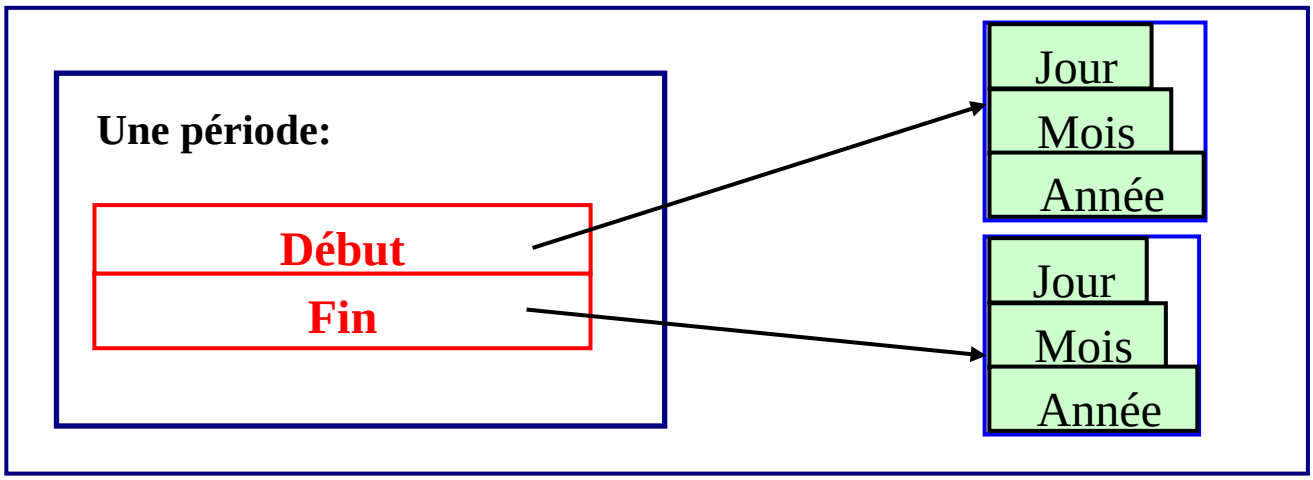
- Les valeurs que peuvent prendre les objets de ce type.
- Les opérations que l'on peut réaliser sur les objets de ce type.
- Et, conséquence indirecte, la taille que les objets de ce type occupent en mémoire.

Le but des structures (**struct**) consiste à définir sous un même nom (une même entité logique) des objets constitués de champs (en C/C++ on parle souvent de membres) de natures différentes.

Prenons l'exemple d'une date:



Les membres qui constituent une structure logique peuvent être de n'importe quel type, y compris une autre structure. Imaginons que nous désirons définir la notion de période! Une période sera un objet qui possède une date de début et une date de fin; nous pouvons donc schématiquement la représenter de la manière suivante:



Le membre *Début* d'une période sera du type *date*, de même que le membre *Fin*!

Une structure correspond à peu près à ce que l'on appelle article ou enregistrement dans d'autres langages. Nous commencerons par présenter les possibilités communes à C/C++, puis les spécificités propres à C++.

.

Forme générale d'une telle déclaration:

```
struct [ nomDeStructure ]
{  declarationDeChamp;
   [ declarationDeChamp; ... ]
} [ identificateur, ... ];
```

Le principe général d'une telle déclaration demeure très proche de celui que nous avons vu pour les types énumérés et il reste valable aussi pour les *champs de bits* et les *unions* que nous présenterons dans la suite :

- **struct**: le mot réservé introduisant une telle définition.
- *nomDeStructure*: le nom (identificateur) que nous donnons à notre définition de type. Sa présence n'est pas obligatoire si nous déclarons directement dans cette même spécification les seuls objets que nous désirons de ce type.
- *déclarationDeChamp*: les membres (champs) de la structure se déclarent entre { }. Chaque déclaration de membre a la même forme qu'une déclaration usuelle de variable sans spécification de classe. Chaque membre peut être d'un type quelconque, y compris d'un autre type structure défini au préalable, mais pas de la structure en cours de définition. Par contre on peut avoir un membre pointeur sur la structure elle-même. Notons aussi qu'un nom de membre (ainsi que le nom d'une structure elle-même) peut être le même que celui d'une variable simple; bien que cela soit syntaxiquement possible, nous vous le déconseillons très vivement pour des raisons de lisibilité.[1]

Les identificateurs qui peuvent facultativement venir après l'accolade fermante représentent des variables que l'on déclare de notre type structure.

Quelques exemples:

1)

```
struct date
{
   int jour;
   int mois;
   int annee;
};
```

Ici, les 3 champs possèdent le même type, chose possible, mais pas obligatoire. Dans cet exemple, nous ne déclarons pas de variable, ceci implique certainement que nous le ferons plus tard dans la suite de l'application.

2)

```
date naissance, mort [ 10 ], *ptDate;
```
[2]

Comme nous le voyons ci-dessus, nous pouvons non seulement déclarer des variables simples, mais aussi des tableaux ou des pointeurs dont les éléments sont d'un type structure. Ceci sera aussi valable pour les autres déclarations de types dans ce chapitre.

[1] Contrairement à d'autres langages, pas de valeur par défaut pour les champs!

[2] En C nous devons remettre le mot réservé **struct** devant le nom du type, à moins de passer par une définition de type **typedef**.

Nous l'avons déjà signalé, une structure peut comporter un membre (ou plusieurs) qui lui-même consiste en une structure:

3)

```
struct
{
   char   nom [ 20 ];
   date   naissance;   // En C mettre struct
   char   indicateur;
} personne_1, personne_2;
```

Ici, nous ne nommons pas notre structure, car nous déclarons directement les seuls objets que nous désirons de ce type. A nouveau, pour des raisons de souplesse et d'adaptation du programme, nous déconseillons d'utiliser ce procédé bien qu'il soit correct. Nous préférons, même si cela nous oblige à écrire un peu plus de code:

```
struct tPersonne
{
   char   nom [ 20 ];
   date naissance;
   char   indicateur;
};
tPersonne personne_1, personne_2;
```

En C les déclarations ci-dessus feraient généralement l'objet de définitions de types dans le but d'éviter de remettre systématiquement dans la suite le mot **struct**. Soit:

1b)

```
typedef struct
{
   int jour;
   int mois;
   int annee;
} DATE;
```

Attention: ici *DATE* correspond au nom donné au type et non pas à une déclaration de variable!

En C++ cette forme, aussi possible, ne présente que peu d'avantages réels.

Comme pour les autres variables, nous pouvons initialiser un objet de type structure lors de

sa déclaration. De même que pour les tableaux, l'initialisation d'une structure se fait par l'intermédiaire d'un agrégat dont les valeurs de chaque membre (constante du type approprié) sont mises entre { } et séparées les unes des autres par des virgules.

Toutefois, il est possible d'initialiser que le début de la structure (ses premiers champs).

Exemples:

```
date uneDate = { 12, 5, 1992 };
tPersonne lesPersonnes [] = { "TOTO", { 12, 5, 1992 }, 'A',
                              "TUTU", { 1,  3, 1960 }, 'Z',
                              "TOTO2",{ 2,  7, 1942 }, 'Z' };
```

Rien de neuf sous le soleil, nous commençons simplement à mettre ensemble plusieurs notions étudiées.

Initialisation en C++11

La norme C++11 permet d'utiliser la notation d'initialisation aussi dans d'autres situations. On peut l'utiliser pour les appels de fonction. Si nous avons par exemple une fonction qui prend comme paramètre une date

```
void f(date debut) { ... }
```

on peut appeler la fonction et passer un agrégat en paramètre

```
f( { 12, 5, 1992} );
```

On peut utiliser la notation aussi pour les valeurs de retour

```
return { 12, 5, 1992};
```

Initialisation en C99

Comme pour les tableaux, la norme C99 a introduit la possibilité de désigner dans un agrégat les membres par leur nom. **Cette possibilité ne se retrouve malheureusement pas en C++.**

L'opération se réalise sous la forme:

```
.nomDuMembre = valeur
```

Exemple avec notre type *DATE*:

```
DATE uneDate = { .jour = 1, .mois = 1, .annee = 2007 };
```

Cette notation offre l'avantage de ne pas devoir nécessairement initialiser tous les champs:

```
DATE uneDate = { .mois = 1 };
```

Ici *jour* et *annee seront* mis par défaut à 0!

Nous pouvons mélanger les notations par nom et par position:

```
DATE uneDate = { .mois = 1, 2007 };
```

La valeur donnée par position qui suit une valeur donnée par nom correspond toujours au champ suivant de la définition du type. Dans notre exemple ci-dessus *jour* prend la valeur par défaut 0, *mois* la valeur 1 et *annee* 2007.

Attention aux effets indésirables:

```
DATE uneDate = { 5, .jour = 1, 3, 2007 };
```

Dans cet exemple *jour* prend la valeur 1, *mois* la valeur 3 et *annee* 2007 et le 5 donné initialement a été écrasé par le 1!

❑ Utilisation des structures

Et que fait-on maintenant avec nos structures?

- On peut réaliser une affectation globale entre objets d'un même type structure:

```
personne_2 = personne_1;
```

- On peut les transmettre comme paramètres (passage par copie de valeur ou par référence en C++) ou comme résultat d'une fonction.

Un membre d'une structure s'utilise partout où une variable simple du même type que le membre serait possible.

Comme dans beaucoup de langages, la notation pointée permet de désigner un membre:

```
personne_1.naissance.annee = 1961;
```

Souvent des pointeurs référencent des structures; dans ce cas on se sert généralement d'une autre notation:

Si *pt* est un pointeur sur une structure (ou une union, c.f. suite), l'accès à un membre de la structure dont l'adresse de base se trouve dans *pt* s'écrit:

```
( *pt ).champ
```

Mais ceci peut également s'écrire:

```
pt -> champ
```

La seconde forme semble plus simple, plus lisible et est généralement plus utilisée.

A noter la syntaxe pour accéder à un élément d'un tableau de structures (pour autant que *ptr* pointe sur le premier élément du tableau):

```
( ptr + 3 ) -> champ
```

ou

```
( *( ptr + 3 ) ).champ
```

ou encore

```
ptr[3].champ
```

Structures en paramètres

Rappelons que les opérateurs "*" et "&" sont au même niveau de priorité, par contre les opérateurs "." et "->" sont plus prioritaires. 5

A titre d'exemple, imaginons que nous avons déclaré:

```
struct date { int jour; int mois; int annee; };
```

Une procédure pour lire une date peut s'écrire (elle n'est pas forcément très propre!):

```
void lireDate ( date *d )
{
  cout << "Donnez le jour: " ); cin >> (*d).jour;
  cout << "Donnez le mois: " ); cin >> (*d).mois;
  cout << "Donnez l'annee: " ); cin >> (*d).annee;
}
```

Et un appel possible de cette fonction:

```
lireDate ( &laDate );
```

Elle peut aussi s'écrire:

```
void lireDate ( date *d )
{
  cout << "Donnez le jour: "; cin >> d->jour;
  cout << "Donnez le mois: "; cin >> d->mois;
  cout << "Donnez l'annee: "; cin >> d->annee;
}
```

Toutefois en C++ il sera plus simple de passer le paramètre par référence et non pas sous la forme d'un pointeur, ce qui nous donne:

```
void lireDate ( date & d )
{
  cout << "Donnez le jour: "; cin >> d.jour;
  cout << "Donnez le mois: "; cin >> d.mois;
  cout << "Donnez l'annee: "; cin >> d.annee;
}
```

Un appel possible de cette fonction:

```
lireDate ( laDate );
```

On peut évidemment aussi sucharger l'opérateur >> pour le type *date*!

Remarques complémentaires

- Nous avions vu que pour les tableaux, il n'est pas possible de réaliser une affectation globale; on peut détourner ce problème en englobant le tableau dans une structure dont le seul champ est le tableau lui-même!
- Les types **struct** nous permettent de construire des structures de données dynamiques, telles que des listes, aspects que nous aborderons dans un chapitre ultérieur.
- Les champs d'une structure peuvent porter les attributs **const** et **volatile**, mais pas de classe de déclaration:

```
struct exemple { float x;
                 const int i; };
```

- Ceci a des conséquences (pour **const**) sur les déclarations d'objets de ce type. Ils devront impérativement être initialisés lors de cette déclaration. Comme une telle initialisation implique de donner une valeur à tous les champs de la structure qui précèdent celui que l'on veut initialiser, il faudra le faire aussi pour ceux qui ne sont pas constants:

```
struct exemple article = { 0.0, 100 };
```

- Il est possible de faire une prédéclaration d'un type structure. Ceci devient indispensable en cas de références croisées entre 2 types structures par l'intermédiaire de pointeurs:

```
struct exemple1;
struct exemple2;
struct exemple1 { float x;
                  struct exemple2 *pt; };
struct exemple2 { float x;
                  struct exemple1 *pt; };
```

- La portée d'un nom de champ d'une structure ou d'une union se limite à cette structure ou cette union. Cela signifie que 2 ou plusieurs de ces types peuvent comporter des noms de champs identiques!

- Lorsque la déclaration d'un type structure (il en sera de même pour les unions) comporte en même temps des déclarations de variables de ce type (et uniquement dans ce cas-là) la classe ***static*** peut précéder les mots **struct** ou **union**:

```
static struct exemple { float x;
                        int I; } val1, val2;
```

  mais alors la classe s'applique aux objets déclarés (ici *val1* et *val2*), mais pas au type lui-même, donc pas aux futurs variables que l'on déclarerait de ce type, sans les préciser explicitement **static**.

- Notez que pour des raisons d'efficacité il n'est pas possible de faire des comparaisons globales sur des structures ou des unions, même pour les opérateurs == et != (la raison réside dans l'éventuelle présence d'octets supplémentaires pour cause d'alignement en fonction du type de certains champs!).

Un exemple

Voilà pour les idées générales. Donnons maintenant un programme exemple. Il s'agit d'un début de système pour travailler avec des vecteurs. Nous reprendrons ce code pour démontrer d'autres possibilités par la suite.

Voici ce code:

```
#ifndef _Vecteurs_
#define _Vecteurs_
/*
   Outils de base pour la gestion de vecteurs
   STRUCTURES1/STRUCTURES.h
*/
/* Definition du type vecteur */
 struct vecteur
{
  float dx;
  float dy;
};

/* Addition de 2 vecteurs */
vecteur operator + ( const vecteur &, const vecteur & );
/* Affichage d'un vecteur */
void affiche ( const vecteur & );
/* Lecture d'un vecteur */
void lire ( vecteur & );
#endif
```

```
/*
   Outils de base pour la gestion de vecteurs
   STRUCTURES1/STRUCTURES.cpp
*/
#include "structures.h"
#include <iostream>
using namespace std;

/* Addition de 2 vecteurs */
vecteur operator + ( const vecteur &v1, const vecteur &v2 )
{
  vecteur temp = { v1.dx + v2.dx, v1.dy + v2.dy };   //[1]
```

[1] **return** { v1.dx + v2.dx, v1.dy + v2.dy} n'est pas possible; un agrégat ne s'utilise que dans une declaration!

```
  return temp;
}

/* Affichage d'un vecteur */
void affiche ( const vecteur &v )
{
  cout << "( " << v.dx << ", " << v.dy << " )";
}

/* Lecture d'un vecteur */
void lire ( vecteur & v )
{
  cout << "\nDonnez la valeur de dx: ";
  cin >> v.dx;
  cout << "Donnez la valeur de dy: ";
  cin >> v.dy;
}
```

```
/*
  Exemple: Utilisation de types struct
    STRUCTURES1/TESTSTRUCTURES
*/
#include "structures.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main ( )
{
  /* Declaration des variables de travail  */
  vecteur v1, v2;

  cout << "Exemple de type structure\n\n";
  cout << " Donnez un premier vecteur: ";
  lire ( v1 );
  cout << " Donnez un deuxieme vecteur: ";
  lire ( v2 );

  cout << "Somme des 2 vecteurs = ";
  affiche ( v1 + v2 );
  cout << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Le résultat d'une exécution de ce programme:

```
Exemple de type structure

 Donnez un premier vecteur:
Donnez la valeur de dx: 1
Donnez la valeur de dy: 2
 Donnez un deuxieme vecteur:
Donnez la valeur de dx: 3
Donnez la valeur de dy: 4
Somme des 2 vecteurs = ( 4, 6 )

Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *C++: fonctions membres*

La possibilité que nous allons présenter maintenant n'existe pas en C. En plus des données que nous venons de voir, en C++ les membres d'une structure peuvent aussi consister en des fonctions que nous appellerons méthodes dans ce contexte. Nous approchons ainsi des notions de programmation objet. Nous pouvons dire qu'une structure représente une première approche de la notion de classe. Il s'agit de classes simplifiées ne comportant a priori pas la notion d'encapsulation (le fait de cacher l'implémentation des données à l'utilisateur).

Si chaque variable (objet) déclarée d'un tel type comporte son propre exemplaire des données, les méthodes elles sont uniques pour l'ensemble des objets déclarés de ce type[1].

Une déclaration de structure prend maintenant la forme générale suivante:

struct [nomDeStructure]
{ declaration_de_champ; | declaration_de_méthode; [2]
[declaration_de_champ; | declaration_de_méthode; ...]
} [identificateur, ...];

Exemple:

```
struct vecteur
{
  float dx;
  float dy;
  /* Affichage d'un vecteur */
  void affiche ( );
  // ...
};
```

Un objet *vecteur* disposera ici de 2 membres données: *dx* et *dy* et d'un membre méthode: *affiche*. En principe, dans la déclaration de la structure nous ne définissons que le prototype de la méthode. Ceci n'est pas obligatoire, mais raisonnable selon le principe de découpage que nous préconisons pour une application: généralement une telle déclaration se trouvera dans un fichier d'inclusion, certainement avec d'autres éléments encore.

Note: bien qu'en théorie nous pouvons alterner à volonté membres données et méthodes, nous vous conseillons d'éviter ce genre de situation.

La définition d'une fonction membre prend une forme un peu particulière puisque nous

[1] Sauf si une fonction est spécifiée **inline**.

[2] Comme les [] et …, le symbole | ne fait pas partie de la syntaxe. Il signifie "ou" donc désigne une alternative possible.

devons signifier sa dépendance à la structure qui contient sa déclaration par l'intermédiaire de l'opérateur "::". Ceci donne pour notre exemple:

```
/* Affichage d'un vecteur [1] */
void vecteur::affiche ( )
{
  cout << "( " << dx << ", " << dy << " )";
}
```

Mais on affiche quoi puisqu'il n'y a pas de paramètre? En fait l'objet auquel on appliquera la méthode *affiche* joue le rôle de premier (et ici unique) paramètre. Donc si dans une application nous déclarons une variable:

```
vecteur  unVecteur;
```

nous utiliserons la notation:

```
unVecteur.affiche ();
```

pour effectivement afficher *unVecteur*.

Bien entendu la méthode peut aussi posséder d'autres paramètres prenant la forme usuelle que nous connaissons.

Dans la méthode on accède directement à ses membres:

```
cout << dx;
```

sous-entendu le champ *dx* de l'objet courant.

L'adresse de l'objet courant se désigne par le mot réservé **this**. Il s'agit d'un pointeur. L'instruction ci-dessus peut donc aussi s'écrire:

```
cout << this -> dx;
```

mais aussi, puisque nous disposons de 2 notations pour accéder à l'objet désigné par un pointeur:

```
cout << (*this).dx;
```

[1] Dans le cas où le corps de la méthode se trouve dans la définition de la structure celui-ci s'écrit de manière habituelle: **void** affiche ( ) … et l'on ne donne pas la spécification!

Reprenons l'exemple rudimentaire basé sur les vecteurs, nous ajouterons quelques indications complémentaires à la suite:

```
#ifndef _Vecteurs_
#define _Vecteurs_
/*
  Outils de base pour la gestion de vecteurs
   STRUCTURES2/STRUCTURES.h
*/
/* Definition du type vecteur */
struct vecteur
{
  float dx;
  float dy;
  /* Addition de 2 vecteurs */
  vecteur operator + ( const vecteur & );
  /* Affichage d'un vecteur */
  void affiche ( );
  /* Lecture d'un vecteur */
  void lire ( );
};
#endif
```

```
/*
  Outils de base pour la gestion de vecteurs
   STRUCTURES2/STRUCTURES.cpp
*/
#include "structures.h"
#include <iostream>
using namespace std;

/* Addition de 2 vecteurs */
vecteur vecteur::operator + ( const vecteur & v )
{
  vecteur temp = { v.dx + dx, v.dy + dy };
  return temp;
}

/* Affichage d'un vecteur */
void vecteur::affiche ( )
{
  cout << "( " << dx << ", " << dy << " )";
}
```

```
/* Lecture d'un vecteur */
void vecteur::lire ( )
{
  cout << "\nDonnez la valeur de dx: ";
  cin >> dx;
  cout << "Donnez la valeur de dy: ";
  cin >> dy;
}
/*
  Exemple: Utilisation de types struct
    STRUCTURES2/TESTSTRUCTURES
*/
#include "structures.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main ( )
{
  /* Declaration des variables de travail */
  vecteur v1, v2, v3;

  cout << "Exemple de type structure\n\n";
  cout << " Donnez un premier vecteur: ";
  v1.lire ();
  cout << " Donnez un deuxieme vecteur: ";
  v2.lire ();

  cout << "Somme des 2 vecteurs = ";
  v3 = v1 + v2;
  v3.affiche ( );
  cout << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Son exécution donne les mêmes résultats que la première version présentée.

Vous constatez sur cet exemple qu'une fonction membre peut aussi surcharger un opérateur:

```
vecteur operator + ( const & vecteur );
```

Il s'utilisera comme un opérateur usuel, soit:

```
vecteur v1, v2, v3
...
v1 = v2 + v3;
```

Mais nous pouvons aussi l'utiliser sous la forme:

```
v1 = v2.operator + ( v3 );
```

Bien que l'on utilise peu cette formulation en pratique, elle peut nous aider à la compréhension du mécanisme!

Mais alors, puisque nous avons surchargé l'opérateur "+" pour implémenter l'addition de 2 vecteurs, pourquoi ne pas en faire de même pour l'affichage du vecteur en surchargeant le "<<", de même pour le ">>"? Une telle opération sera possible par la suite, mais elle fait appel à des notions théoriques qui ne seront abordées qu'ultérieurement[1].

❑ *Les champs de bits*

Il s'agit d'un cas particulier de structures et se déclare comme telle, mais chaque membre est complété par sa taille exprimée en bits.

L'utilisation d'une telle structure de données peut viser 2 objectifs:

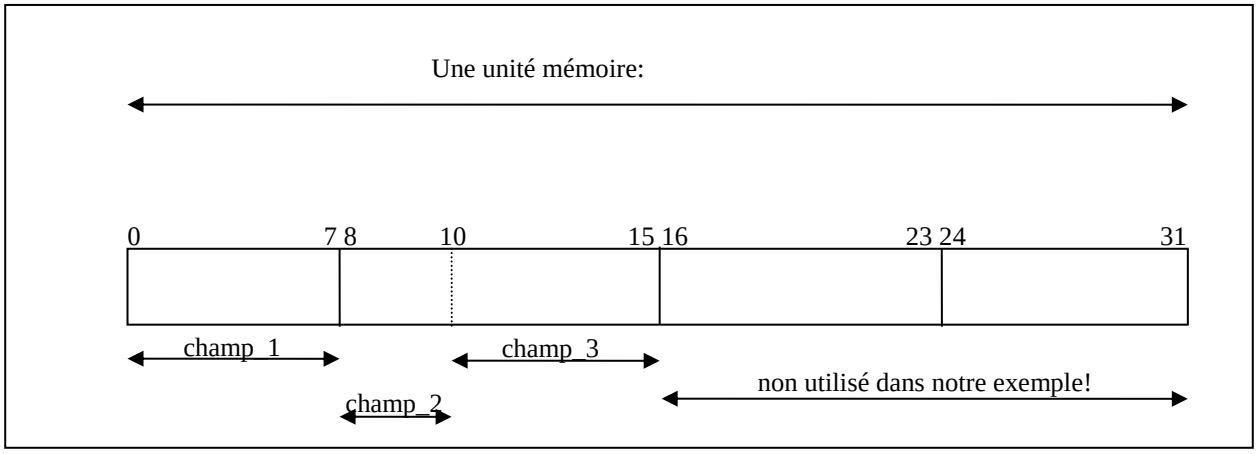
- Pouvoir accéder à des bits spécifiques dans des mots en mémoire, essentiellement pour résoudre des problèmes de programmation système.
- Economiser de la place mémoire en compactant l'information (les membres) dans des bits successifs.

Voici un exemple de déclaration de champs de bits:

```
struct exemple
{
   unsigned champ_1 : 8;
   unsigned champ_2 : 3;
   unsigned champ_3 : 5;
};
```

[1] Il s'agit de la notion de fonction amie (friend) que nous traiterons prochainement dans le chapitre des classes.

Un objet de ce type peut correspondre en mémoire à la représentation suivante[1]:



Notre *exemple* consiste en un type comportant 3 champs. Le premier occupe 8 bits, le second 3 et le dernier 5.

Tous les éléments présentés pour les structures (**struct**) de base restent valables pour les champs de bits.

La taille d'un membre ne peut pas dépasser celle d'un entier et ne peut pas en principe chevaucher la limite de 2 entiers; si tel devait être le cas, généralement le début du champ est automatiquement aligné sur la frontière de l'entier suivant, ce qui peut laisser des bits inutilisés.

Un champ peut ne pas spécifier de longueur. Il occupera la taille usuelle d'un tel objet.

Dans notre exemple, nous avons fixé pour chaque champ un type **unsigned**; nous pouvons aussi utiliser **int**, mais la nature des applications que l'on traite au moyen de ce genre de données, fait qu'en pratique nous verrons presque toujours **unsigned**. Ces applications sont généralement d'ordre système, où l'on manipule par exemple un mot d'état. Les bits sont alloués aux champs en principe depuis le poids faible du mot, vers le poids fort, mais ceci sans garantie de la norme!

Les membres d'une telle variable s'utilisent, du point de vue de l'affectation, comme n'importe quelle variable entière:

```
exemple valeur;    [2]
...
valeur.champ_1 = 255;
valeur.champ_3 = 12;
```

[1] Dépend de l'environnement!

[2] A nouveau en C il faut toujours ajouter **struct** devant *exemple*!

Nous pouvons aussi réaliser des tests sur les valeurs des champs:

```
if ( valeur.champ_2 == 0 ) ...
```

Comme en programmation système les champs sont souvent de 1 bit (flag), si l'on désire mettre plusieurs de ces bits à 1, on utilisera des affectations multiples du genre:

```
flag.bit_1 = flag.bit_3 = flag.bit_4 = 1;
```

Une telle possibilité n'est pas indispensable, car nous aurions pu écrire:

```
#define  BIT_1  1
#define  BIT_3  4
#define  BIT_4  8
unsigned flag;
...
flag = flag | BIT_1 | BIT_3 | BIT_4;
```

Mais la lisibilité devient bien meilleure dans la solution avec structure, et les risques d'erreurs se réduisent considérablement.

Les champs de bits peuvent comporter un champ anonyme, dont le seul but consiste à réserver explicitement des bits non utilisés; de plus un champ peut aussi porter l'un des attributs **const** ou **volatile**:

```
struct etat { unsigned x  : 2;
              unsigned    : 3;     //champ anonyme
              const int i : 5; };
```

Les opérations arithmétiques usuelles sur les entiers s'appliquent aux champs de bits, avec la réserve habituelle: aucun contrôle de débordement n'est effectué; vous le remarquerez avec la dernière opération réalisée dans notre programme exemple!

Notez encore que dans le cas où l'objectif visé consiste à économiser la place mémoire, il sera raisonnable de bien réfléchir au problème avant de prendre une telle décision.

En effet, n'oubliez pas que les miracles n'existent pas:

- Pour chaque référence à un champ de bits le compilateur doit générer du code supplémentaire pour isoler les bits en question, ce qui va évidemment augmenter la taille du programme exécutable! Ce que l'on a gagné d'un côté, on risque de le perdre d'un autre!
- Ce code supplémentaire devra être exécuté, ce qui nécessitera du temps CPU pour le réaliser!

On ne devrait envisager une telle solution que dans les cas où l'on manipule des structures de données importantes en taille (tableaux), avec relativement peu d'endroits dans le code où l'on fait référence à ces objets!

Notons finalement que de par leur nature les champs de bits représentent une notion peu portable!

❑ *Exemple*

```
/*
  Programme exemple: Utilisation de types champs de bits
    CHAMPS_DE_BITS
*/
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;

int main ( )
{
  /* Definition de chaines de caracteres pour les mois de l'annee */
  char lesMois [12] [10]={ "JANVIER", "FEVRIER", "MARS",
                "AVRIL", "MAI", "JUIN", "JUILLET", "AOUT",
                "SEPTEMBRE", "OCTOBRE", "NOVEMBRE", "DECEMBRE" };
  /* Definition d'un type champ de bits pour compacter une date */
  struct date
  {
    unsigned annee : 11;
    unsigned mois  : 4;
    unsigned jour  : 5;
  } ;
  /* Une date arbitraire pour l'exemple */
  date laDate = { 1992, 4, 13 };


  cout << "Exemple d'utilisation de champs de bits\n\n";
  cout << "La date choisie est: " << laDate.jour << '/' << laDate.mois
       << '/' << laDate.annee << endl;
  cout << "Le mois correspondant est: " << lesMois [laDate.mois-1]
       << endl;
  cout << "La taille d'un objet de type \"date\" est: "
       << sizeof ( date ) << endl;
  /* On peut realiser des calculs sur les champs de bits */
  laDate.jour  = laDate.mois;
  laDate.mois  = laDate.mois + 1;
  laDate.annee = laDate.mois + laDate.annee;
```

```
  cout << "La nouvelle date apres calcul est: " << laDate.jour << '/'
       << laDate.mois << '/' << laDate.annee << endl;
  /* Attention toujours pas de controle en C++ */
  laDate.mois = laDate.mois + 21;
  cout << "La nouvelle date est n'importe quoi: "
       << laDate.jour << '/'
       << laDate.mois << '/' << laDate.annee << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Le résultat de l'exécution de ce programme:

```
Exemple d'utilisation de champs de bits

La date choisie est: 13/4/1992
Le mois correspondant est: AVRIL
La taille d'un objet de type "date" est: 4
La nouvelle date apres calcul est: 4/5/1997
La nouvelle date est n'importe quoi: 4/10/1997

Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *Les unions*

Les champs (membres) d'une union occupent tous la même zone mémoire, ils ne sont donc utilisables individuellement qu'à des moments différents de l'exécution du programme.

La syntaxe d'une déclaration de type **union** reste la même que celle d'une structure, le mot réservé **union** remplaçant simplement **struct**:

```
union nom_d_union
{
   champ;
   [ champ; ... ]
} [ identificateur, ... ];
```

La sémantique, elle, change totalement puisque chaque champ de l'union utilise la même zone mémoire; il y a donc recouvrement.

Cela permet:

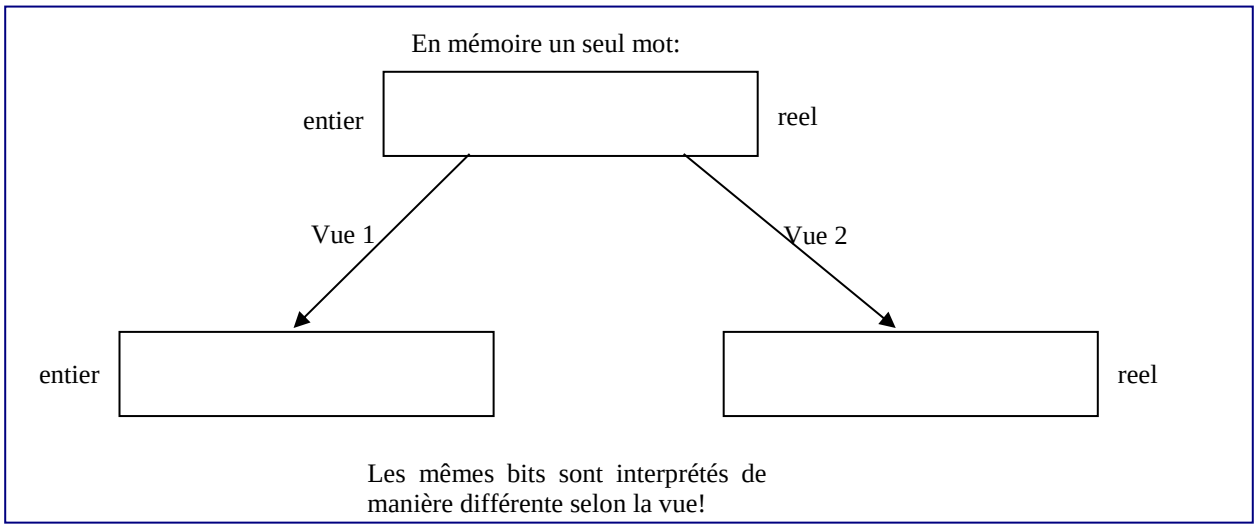
1. De réduire dans une même zone de mémoire des objets de types différents (à des temps différents).
2. De réduire un objet sous la forme d'un certain type et de l'utiliser (l'interpréter) ensuite sous la forme d'un autre type; on trompe ainsi le compilateur.

Exemple*:*

```
union tValeur
{
   long int entier;
   float    reel;
} laValeur;
```

Nous déclarons ici une variable *laValeur* avec la définition du type.

On peut schématiser la représentation en mémoire d'un objet de ce type de la manière suivante:



Si nous travaillons sur une machine où les **long int** et les **float** utilisent le même nombre de bits, cela nous permettra par exemple, à partir d'une valeur **float** d'afficher la valeur **long int** correspondant à sa configuration de bits, ce que nous avons fait dans l'exemple de programme qui suit.

L'utilisation d'un membre d'une union se fait comme pour celui d'une structure, avec la notation pointée:

```
laValeur.reel = ...
...
cout << laValeur.entier << endl;
```

Notez que contrairement à d'autres langages nous ne disposons pas d'un champ qui permet de savoir quelle est la variante actuellement valable dans l'union. C'est au programmeur, par ses propres moyens d'en conserver la trace si nécessaire.

Comme tous les autres, un objet de type **union** peut s'initialiser au moment de sa déclaration, mais dans ce cas une seule valeur est possible (elle correspond toujours à la première variante de l'union):

```
union exemple { float x;
                int i;
              };
exemple valeur = { 3.5 };
exemple valeur = { 3 }; // Par conversion en float!
```

Il y a des restrictions concernant quels types sont permis pour les membres d'une union. D'abord il faut savoir que tous les types primitifs sont permis. Ensuite, en ce qui concerne les objets, seulement un certain type d'objets (des objets qui ont un constructeur par défaut) est permis, mais pas les objets en général. Nous mentionnons qu'avec la norme C++11 cette restriction est levée (on parle alors d' « unions non restreintes» ou « unrestricted unions ») sans

vouloir entrer dans les détails.

❑ *Exemple*

Il consiste à lire une valeur réelle et ensuite à interpréter les bits de ce réel comme s'ils représentaient une valeur entière non signée:

```
/*
  Programme exemple: Utilisation de types union
     UNION
*/
#include <cstdlib>
#include <iostream>
using namespace std;

int main ( )
{
  /* Definition d'un type union, avec declaration de variable */
  union tValeur
  {
     unsigned long entier;
     float         reel;
  } laValeur;

  cout << "Exemple de type union\n\n";
  cout << " DONNEZ UNE VALEUR REELLE: ";
  cin >> laValeur.reel;
  /* Affichage du resultat */
  cout << "La configuration de bits correspond a l'entier: "
       << laValeur.entier << "\nou, en octal: " << oct
       << laValeur.entier << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Exemple d'exécution:

```
Exemple de type union

 DONNEZ UNE VALEUR REELLE: 0.75
La configuration de bits correspond a l'entier: 1061158912
ou, en octal: 7720000000

Fin du programme...Appuyez sur une touche pour continuer...
```

[1] Les réels sont codés selon la norme IEEE 754

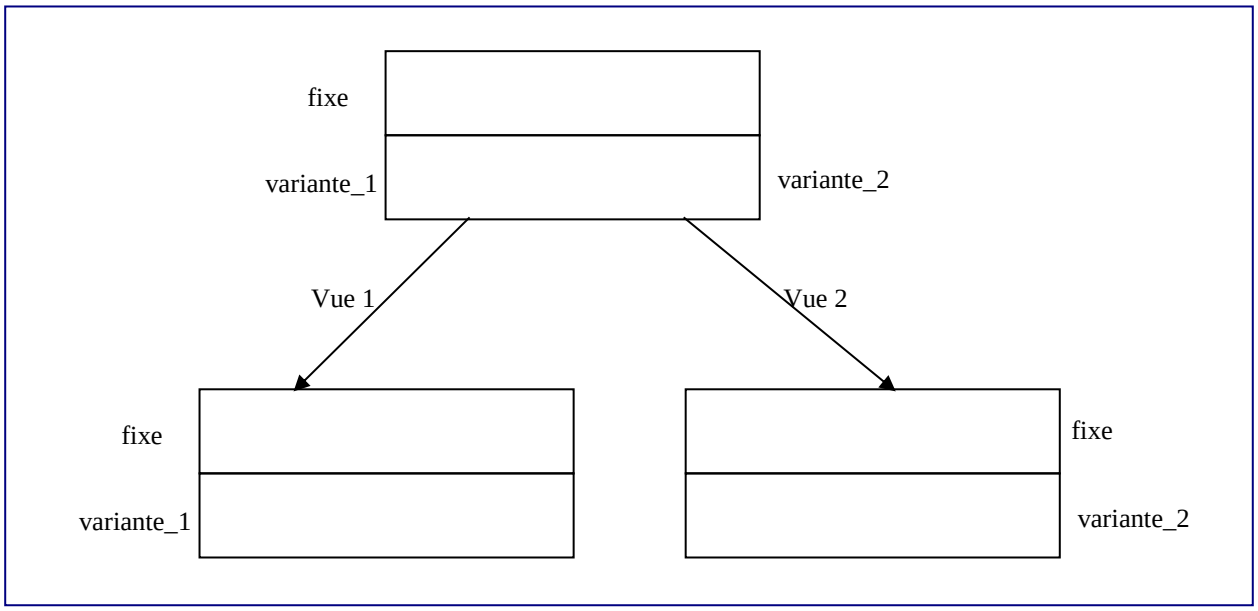
Combinaisons struct/union

Nous pouvons combiner des structures et des unions dans une même déclaration.

Exemple:

```
struct
{
   long int fixe;
   union
   {
      int   variante_1;
      float variante_2;
   } lesVariantes;
} laVariable;
```

Un objet de ce type peut se représenter de la manière suivante:



Avec une telle déclaration, nous pouvons écrire:

```
laVariable.fixe = 0;
laVariable.lesVariantes.variante_2 = 10.0;
```

Il serait certainement préférable dans notre exemple que l'union fasse l'objet d'une déclaration préalable de type que l'on utilise ensuite dans la définition de la structure.

Notez que la partie **union** peut venir n'importe où dans la définition d'une structure et d'ailleurs il peut y avoir plusieurs parties **union** dans une même structure.

Signalons finalement que de toute façon la taille nécessaire à la variante la plus grande est toujours réservée, ce qui permet évidemment de contenir aussi la(les) variante(s) de taille(s) plus petite(s).

En guise de conclusion, et pour illustrer ce qui précède, imaginons que nous voulons représenter l'architecture du processeur 8086 (même si celle-ci est quelque peu dépassée aujourd'hui), ceci en admettant que dans l'environnement de travail les **int** se représentent sur 16 bits:

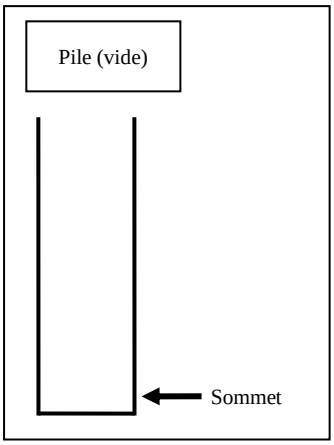
```
struct registresMot
{ unsigned int ax : 16, bx : 16, cx : 16, dx : 16,
               si : 16, di : 16, flag : 16; };


struct registresByte
{ unsigned char al : 8, ah : 8, bl : 8, bh : 8,
                cl : 8, ch : 8, dl : 8, dh : 8; };
union registres
{
   registresMot  x;
   registresByte lh;
};
struct registresBase
{ unsigned int cs, ds, es, ss; };
```

Structures de données

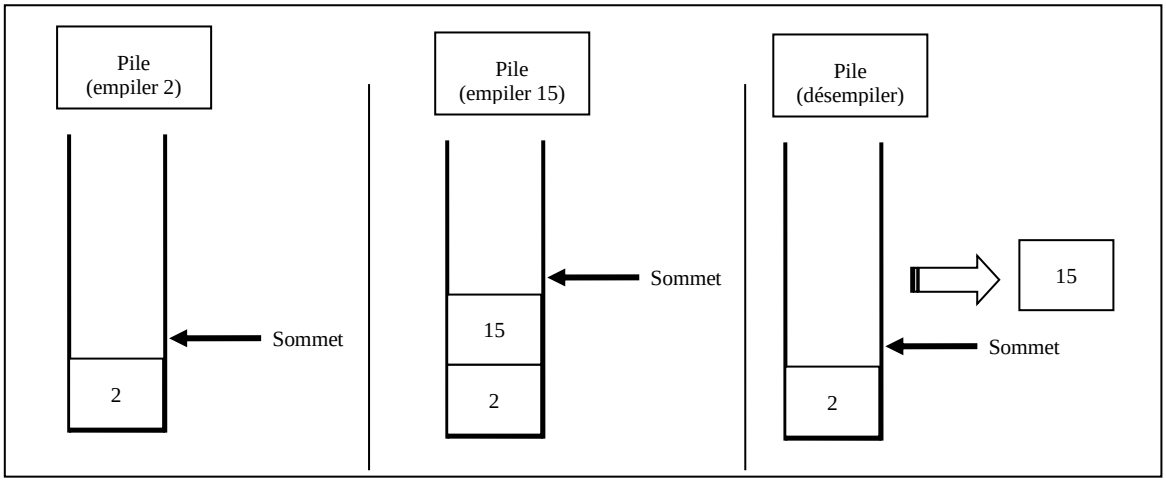
Introduction

Dans ce chapitre qui termine la première partie de ce cours, nous n'allons pas introduire de nouvelles notions sur le langage. Notre but final ici consiste simplement à mettre en pratique la théorie étudiée, et particulièrement l'utilisation des tableaux et des pointeurs. De plus et même si le titre du chapitre "Structures de données" peut le laisser supposer, il ne s'agit pas non plus de faire un cours sur les structures de données. Nous allons simplement utiliser un cas particulier de structure: la pile, pour illustrer certaines possibilités. Mis à part le dernier exemple de ce chapitre, qui lui est purement C++, les autres programmes sont valables en C et C++, mais écrits dans un style C!



Par définition une pile est a priori vide (ne contient pas d'information); elle possède un élément permettant de désigner le sommet de la pile.

Une pile consiste en une structure de données caractérisée par le fait que l'on introduit les nouvelles informations (empile) toujours au même endroit: au sommet de la pile, et que l'on extrait les informations (désempile) par ce même endroit. Donc la dernière information mise dans la structure de données sera aussi la première à en ressortir; ce qui fait que l'on qualifie souvent la pile de "LIFO" (Last In First Out). Au fur et à mesure des modifications, le sommet de la pile est mis à jour.



❑ *Tableaux : Piles statiques*

Une pile peut s'implémenter par l'intermédiaire d'un tableau ou plus précisément d'une structure **struct** contenant un tableau pour l'information et un index indiquant le sommet de la pile. Nous avons là un cas un peu particulier d'utilisation de tableau puisque sa "taille logique" va varier au cours du temps; un nombre variable d'éléments sera stocké dans le tableau, et pourtant le tableau possèdera une taille physique fixe.

Cette taille fixe représente d'ailleurs le principal inconvénient des structures statiques puisqu'elle reste figée pendant toute l'exécution. On pourrait théoriquement la fixer très grande afin d'absorber toutes les situations imaginables, mais alors nous occuperions en général bien plus de place que nécessaire. De toute façon, quelle que soit la valeur apparemment raisonnable choisie, il arrivera bien un jour où cette taille sera insuffisante pour les besoins d'une exécution spécifique.

L'intérêt en général de l'utilisation d'une structure statique réside dans sa relative simplicité et son efficacité.

Voici une proposition pour le début d'une implémentation de pile statique; pour que cet outil devienne utilisable dans la réalité, il faudrait évidemment le développer davantage, mais tel n'est pas notre but ici! Nous vous donnons ci-dessous le code proposé et quelques explications complémentaires relatives à certains choix suivront.

Tout d'abord le fichier d'inclusion:

```
#ifndef  _Pile_
#define  _Pile_
/*
  Gestion de base d'une pile statique
  PileStatique/Pile.h
*/
/* Definition du champ information d'un element */
typedef int INFO;

/* Definitions des prototypes de quelques fonctions de base  */

/* Fonction d'insertion d'un element sur la pile             */
void empiler ( INFO valeur );

/* Fonction d'extraction de l'element au sommet de la pile   */
void desempiler ( INFO *valeur );

/* Fonction pour determiner si la pile est vide              */
bool pileVide ( );
/* ... dans la realite, il y en aurait certainement d'autres */
#endif
```

Ensuite le "corps" de notre outil de gestion de pile:

```
/*
  Gestion de base d'une pile statique
  PileStatique/PILE.cpp
*/
#include "Pile.h"
#define TAILLE 100
typedef struct
{
   int sommet;
   INFO lesElements [ TAILLE ]; // L'information
}  PILE;

static  PILE laPile;

/* Fonction d'insertion d'un element sur la pile  */
/* Pour simplifier le debordement n'est pas teste!*/
void empiler ( INFO valeur )
{
   laPile.lesElements [ laPile.sommet ] = valeur;
   laPile.sommet++;
} // empiler

/* Fonction d'extraction d'un element de la pile  */
/* Pour simplifier la pile vide n'est pas testee! */
void desempiler ( INFO *valeur )
{
   laPile.sommet--;
   *valeur = laPile.lesElements [ laPile.sommet ];
} // desempiler


/* Fonction pour determiner si la pile est vide   */
bool pileVide ( )
{
   return ( laPile.sommet == 0 );
} // pile_vide

/* ... dans la realite, il y en aurait certainement d'autres */
```

Vous remarquerez que dans cette version nous avons volontairement caché complètement l'implémentation de la pile et même son existence en tant que telle. L'utilisateur sait simplement qu'il peut mettre des éléments du type prédéfini sur la pile, les retirer de la pile et tester s'il y a en encore sur la pile. Bref, il ne peut pas déclarer des variables de type *PILE*!

Pour cette raison le type est défini dans le corps du module et non pas dans son interface; il s'agit là d'un choix d'implémentation! La variable *laPile* elle aussi se déclare comme variable globale à ce module, son attribut explicite **static** garantissant la restriction de sa visibilité au module. De plus, de par le fait de l'appartenance à la classe **static** nous avons la certitude que le membre *sommet* est correctement initialisé à 0.

Finalement un petit programme de test rudimentaire:

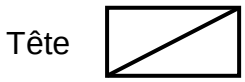
```
/*
  Programme exemple: Test des outils: gestion de pile statique:
  Lit une serie de valeurs et les affiche dans l'ordre inverse!
  Dans le cas present: par l'utilisation d'une pile
 PileStatique/TEST_PILE_STAT
*/
#include "Pile.h"
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  INFO valeur; // La valeur courante du traitement

  /* Traitez toutes les valeurs de l'utilisateur, jusqu'a un 0  */
  do
  {
    cout << "Donnez la valeur a inserer, 0 pour terminer : ";
    cin >> valeur;
    /* Empiler la valeur donnee */
    if ( valeur != 0 )
      empiler ( valeur );
  } while ( valeur != 0 );
  /* Afficher les valeurs dans l'ordre inverse */
  cout << "\n= Les valeurs dans l'ordre inverse =\n";
  /* Tant qu'il y a des elements sur la pile   */
  while ( !pileVide ( ) )
  {
    desempiler ( &valeur );
    cout << valeur << endl;
  }
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

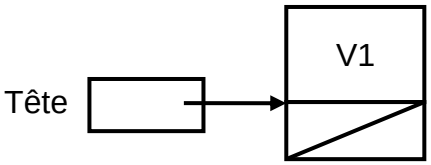
❑ *Listes chaînées : Généralités*

Le principe général des structures dynamiques consiste à ne créer les objets (variables) que lorsque nous en avons réellement besoin, soit en cours d'exécution.

A priori nous ne disposons que d'une seule variable statique, un pointeur nous permettant d'accéder au premier élément de la structure, et puisque pour l'instant la structure est vide (ne comporte aucun élément) le pointeur ne désigne aucun objet, il vaut *NULL*:

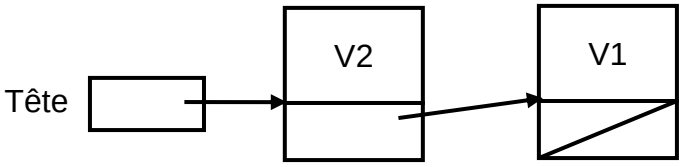


Lorsque nous créons un premier objet dynamique, nous sauvons son adresse dans le pointeur de tête:

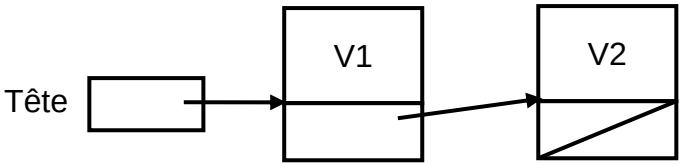


Nous voyons sur la représentation ci-dessus que l'objet créé est de type structure, il ne contient pas seulement la donnée relative à l'information que nous voulons enregistrer, mais également un membre permettant de stocker l'adresse d'un objet de ce type. Ce membre est du même type que la tête de la liste: un pointeur qui pour l'instant ne désigne aucun objet, donc vaut *NULL*!

Ensuite, suivant la nature de la structure que nous voulons réaliser, nous enregistrons l'adresse d'un nouvel objet par exemple dans le pointeur de tête; dans ce cas nous sauvons d'abord dans le membre pointeur de ce nouvel objet l'adresse de l'ancien objet désigné par tête. Si tel n'était pas le cas nous aurons perdu le seul moyen d'accéder à cet ancien objet. Une telle opération nous donne la représentation suivante:



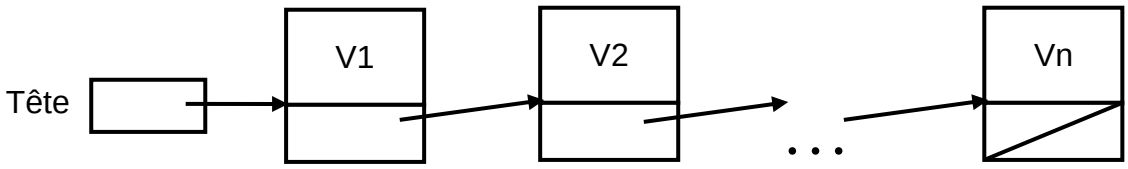
Nous aurions aussi pu insérer le nouvel objet après le précédent. Dans la liste, dans ce cas c'est dans le membre pointeur de l'ancien objet que nous devons sauver l'adresse du nouveau, ce qui nous donnerait la représentation ci-contre:



Des insertions ultérieures peuvent se faire en tête, en queue ou même à l'intérieur de la liste si l'on désire par exemple créer une liste ordonnée.

Si pour gérer une telle structure nous insérons systématiquement un nouvel élément en tête et que lorsque nous prélevons un élément nous le faisons également toujours en tête, nous obtenons alors une gestion de pile.

Remarquons toutefois que si nous disposons d'une liste du genre:

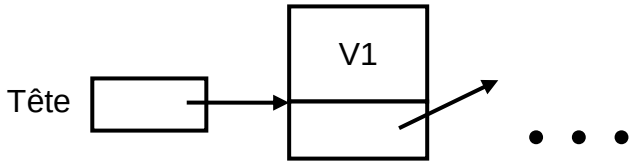


nous ne pouvons accéder à un élément quelconque de la liste qu'en partant de sa tête (puisque c'est le seul objet statique que nous connaissons) et en parcourant tous les éléments qui le précèdent. C'est certainement le principal inconvénient des structures dynamiques. Fort heureusement la nature de nombreux traitements se prêtent bien à une telle utilisation. De plus, par la suite, nous pourrons créer des structures plus complexes et donc plus efficaces suivant le problème à résoudre.

.

❑ *Pile dynamique*

Tout comme dans d'autres langages, nous utilisons la gestion dynamique de la mémoire et le type **struct** pour gérer des structures chaînées. Pour développer une structure du genre:



nous avons besoin des définitions:

```
/* Definition du champ information d'un element */
typedef int INFO;         // …ou un autre type!

/* Definition du type d'un element de la structure */
typedef struct element
{
  INFO valeur;              // L'information
  struct element *suivant; // Pointe l'element suivant
} ELEMENT;

typedef ELEMENT PILE;
```

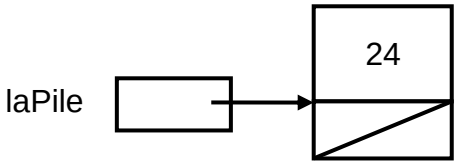
Dans la définition du type structure nous pouvons utiliser un pointeur sur le type que nous sommes en train de définir, par contre nous ne pouvons pas déclarer explicitement un membre de ce type en cours de définition.

Nous avons dit que seule la tête de la structure, qui correspond logiquement pour nous à une pile, serait déclarée statiquement. Ceci implique qu'une application désirant manipuler une pile déclarera une variable du type:

```
PILE *laPile = NULL;
```

Note: nous proposerons une solution plus propre par la suite, mais cette étape intermédiaire devrait nous aider à comprendre les mécanismes!

Pour créer un élément et se mettre dans une situation telle que:



nous devons utiliser des instructions du genre:

```
laPile = ( PILE * ) malloc ( sizeof ( ELEMENT ) );
laPile -> valeur  = 24;
laPile -> suivant = NULL;
```

Notez le paramètre de *malloc* pour obtenir un objet de la taille désirée.

Pour l'insertion de nouveaux éléments nous aurons besoin de récupérer l'ancienne valeur de *laPile*, nous utiliserons donc plutôt la suite d'instructions:

```
PILE *tempo = ( PILE * ) malloc ( sizeof ( ELEMENT ) );
…
tempo -> valeur  = laValeur;
tempo -> suivant = laPile;
laPile           = tempo;
```

En pratique cette opération se fait dans une fonction qui prendra une forme du genre:

```
void empiler ( PILE **sommet, INFO laValeur )
{
   PILE *tempo = (PILE *) malloc ( sizeof ( ELEMENT ) );
   tempo -> valeur  = laValeur;
   tempo -> suivant = *sommet;
   *sommet = tempo;
} // empiler
```

Notez tout de suite l'utilisation d'un pointeur de pointeur comme paramètre de la fonction, ceci pour que l'opération puisse se réaliser complètement dans la fonction, c'est-à-dire pour que le sommet de pile puisse être mis à jour. N'oubliez pas qu'en C/C++, c'est toujours une copie de la valeur du paramètre effectif qui est transmise![1]

Avec cette version du sous-programme (mais nous modifierons ceci d'ici peu!) son appel prend une forme du genre:

```
empiler ( &laPile, laValeur );
```

La fonction *desempiler* peut s'écrire:

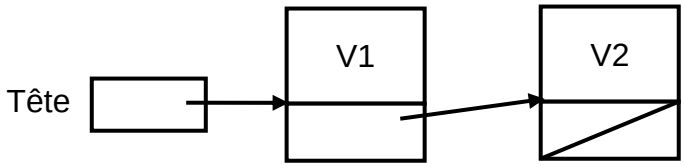
```
void desempiler ( PILE **sommet, INFO *laValeur )
{
   PILE *tempo = *sommet;
   *laValeur   = ( *sommet ) -> valeur;
   *sommet     = ( *sommet ) -> suivant;
   free ( tempo );  // Libere la place
} // desempiler
```

Notez les 2 points suivants:

- L'utilisation d'une variable temporaire pour sauver la valeur du sommet de pile, initialisé lors de sa déclaration.
- L'utilisation de la fonction *free* pour restituer au système la place mémoire dont nous n'avons plus besoin.

Note complémentaire:

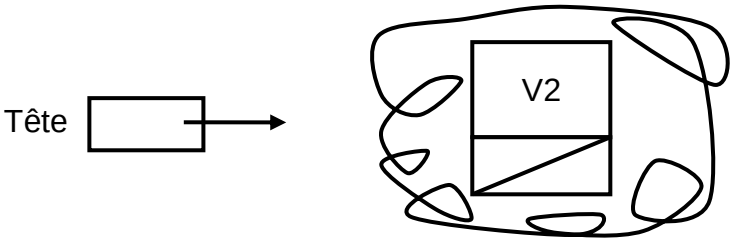
De manière générale soyez attentif lors d'une telle libération, si vous vous trouvez dans la situation suivante:



Que se passe-t-il si vous réalisez l'opération:

```
free ( Tete );
```

Vous vous retrouvez alors dans la situation suivante:



[1] A moins en C++ d'utiliser le mécanisme de passage par référence, ce que nous ferons dans la suite!

A savoir: vous avez bien rendu la place mémoire utilisée par le premier élément de la liste, mais ainsi perdu le moyen d'accéder à la suite de cette structure. N'oubliez pas que pour la parcourir vous devez toujours partir de la tête; vous devez procéder de même pour restituer tous les éléments de la structure: libérer chacun d'eux l'un après l'autre!

Reprenons à titre d'exemple la gestion simplifiée d'une pile, mais cette fois dynamique et avec la possibilité pour l'utilisateur de déclarer sa ou ses pile(s):

```
#ifndef _PILE_
#define _PILE_
/*
  Gestion de base d'une pile dynamique
  PileDynamique/Pile.h
*/
/* Definition du champ information d'un element */
typedef int INFO;
/* Definition du type d'un element de la structure */
typedef struct element
{
  INFO valeur;              // L'information
  struct element *suivant; // Pointeur sur l'element suivant
} ELEMENT;
typedef ELEMENT PILE;
/* Definitions des prototypes de quelques fonctions de base  */
/* Fonction d'insertion d'un element sur la pile             */
void empiler ( PILE **sommet, INFO laValeur );
/* Fonction d'extraction de l'element au sommet de la pile   */
void desempiler ( PILE **sommet, INFO *laValeur );
/* Fonction pour determiner si la pile est vide              */
bool pileVide ( PILE *sommet );
#endif
```

Ensuite donnons le fichier des définitions ("corps") des fonctions utilitaires de manipulation de la pile:

```
/*
  Gestion de base d'une pile dynamique
  PileDynamique/PILE.cpp
*/
#include <cstdlib>
#include "Pile.h"
/* Fonction d'insertion d'un element sur la pile  */
void empiler ( PILE **sommet, INFO laValeur )
{
   PILE *tempo = (PILE *) malloc ( sizeof ( ELEMENT ) );
   tempo -> valeur  = laValeur;
   tempo -> suivant = *sommet;
   *sommet          = tempo;
} // empiler
```

```
/* Fonction d'extraction d'un element de la pile  */
/* On ne teste pas la pile vide !!! */
void desempiler ( PILE **sommet, INFO *laValeur )
{
   PILE *tempo = *sommet;
   *laValeur   = ( *sommet ) -> valeur;
   *sommet     = ( *sommet ) -> suivant;
   free ( tempo );  // Libere la place
} // desempiler

/* Fonction pour determiner si la pile est vide   */
bool pileVide ( PILE *sommet )
{
   return ( sommet == NULL );
} // pileVide
```

Finalement un exemple d'utilisation:

```
/*
  Programme exemple: Utilisation des pointeurs pour gerer des
  structures dynamiques
  Lit une serie de valeurs et les affiche dans l'ordre inverse!
  Dans le cas present: par l'utilisation d'une pile
 PileDynamique/TEST_PILE.cpp
*/
#include "Pile.h"
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  PILE *laPile = NULL;
  INFO valeur; // La valeur courante du traitement

  /* Traitez toutes les valeurs de l'utilisateur, jusqu'a un 0  */
  do
  {
    cout << "Donnez la valeur a inserer, 0 pour terminer : ";
    cin >> valeur;
    /* Empiler la valeur donnee */
    if ( valeur != 0 )
      empiler ( &laPile, valeur );
  } while ( valeur != 0 );
  /* Afficher les valeurs dans l'ordre inverse */
  cout << "\n= Les valeurs dans l'ordre inverse =\n";
```

```
  /* Tant qu'il y a des elements sur la pile   */
  while ( !pileVide ( laPile ) )
  {
    desempiler ( &laPile, &valeur );
    cout << valeur << endl;
  }
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Il n'y a rien de très particulier à dire sur cet exemple si ce n'est de bien observer comment sont passés et traités les paramètres; il s'agit d'une application de ce que nous avons vu en théorie sur les pointeurs!

Nous avons voulu dans cet exemple mettre en évidence la double indirection (pointeurs de pointeurs); toutefois en pratique il serait certainement préférable de cacher davantage à l'utilisateur du module la structure physique de la pile. Dans ce but, le type *PILE* mis à disposition devrait déjà être un pointeur sur un objet de type *ELEMENT*.

Nous allons faire cette modification, applicable aussi bien à un programme C que C++. Toutefois nous profitons aussi de cette deuxième version pour utiliser la mode de passage des paramètres par référence et les opérateurs **new** et **delete** (à la place des fonctions *malloc* et *free*). La version que nous vous proposons ci-dessous n'est donc valable qu'en C++.

```
#ifndef _PILE_
#define _PILE_
/*
  Gestion de base d'une pile dynamique
  PileDynamique2/Pile.h
*/
/* Definition du champ information d'un element */
typedef int INFO;
/* Definition du type d'un element de la structure */
typedef struct element
{
  INFO valeur;        // L'information
  struct element *suivant; // Pointeur sur l'element suivant
} ELEMENT;
typedef ELEMENT * PILE;
/* Definitions des prototypes de quelques fonctions de base  */
/* Fonction d'insertion d'un element sur la pile             */
void empiler ( PILE &sommet, INFO laValeur );

/* Fonction d'extraction de l'element au sommet de la pile   */
void desempiler ( PILE &sommet, INFO &laValeur );

/* Fonction pour determiner si la pile est vide              */
bool pileVide ( PILE sommet );
```

```
#endif
/*
  Gestion de base d'une pile dynamique
  PileDynamique2/PILE.cpp
*/
#include <ctdlib>
#include "Pile.h"
/* Fonction d'insertion d'un element sur la pile  */
void empiler ( PILE &sommet, INFO laValeur )
{
   PILE tempo = new ELEMENT;
   tempo -> valeur  = laValeur;
   tempo -> suivant = sommet;
   sommet           = tempo;
} // empiler


/* Fonction d'extraction d'un element de la pile  */
/* Pile vide non testee                           */
void desempiler ( PILE &sommet, INFO &laValeur )
{
 PILE tempo = sommet;
   laValeur = tempo -> valeur;
   sommet   = tempo -> suivant;
   delete tempo;  // Libere la place
} // desempiler

/* Fonction pour determiner si la pile est vide   */
bool pileVide ( PILE sommet )
{
   return ( sommet == NULL );
} // pile_vide




/*
  Programme exemple: Utilisation des pointeurs pour gerer des
  structures dynamiques
  Lit une serie de valeurs et les affiche dans l'ordre inverse!
  Dans le cas present: par l'utilisation d'une pile
  PileDynamique2/TEST_PILE
*/
#include "Pile.h"
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  PILE laPile = NULL;
```

```
  INFO valeur; // La valeur courante du traitement

  /* Traitez toutes les valeurs de l'utilisateur, jusqu'a un 0  */
  do
  {
    cout << "Donnez la valeur a inserer, 0 pour terminer : ";
    cin >> valeur;
    /* Empiler la valeur donnee */
    if ( valeur != 0 )
      empiler ( laPile, valeur );
  } while ( valeur != 0 );
  /* Afficher les valeurs dans l'ordre inverse */
  cout << "\n= Les valeurs dans l'ordre inverse =\n";
  /* Tant qu'il y a des elements sur la pile   */

  while ( !pileVide ( laPile ) )
  {
    desempiler ( laPile, valeur );
    cout << valeur << endl;
  }
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Comme nous l'avons vu, et sans parler des opérateurs **new** et **delete**, dans une version purement C les paramètres formels du type PILE des fonctions empiler et desempile seraient définis comme pointeurs (donc pointeurs de pointeurs), ce qui nous donnerait par exemple pour empiler le prototype:

```
void empiler ( PILE * sommet, T_INFO valeur );
```

Un appel de cette procédure deviendrait par exemple:

```
empiler ( &laPile, valeur );
```

Comme nous l'avons annoncé au début de ce chapitre, nous ne vous avons donné ici que quelques idées de base. Il vous faudra par la suite aller plus loin dans le développement de ces principes, ce qui sera fait dans le cadre d'un autre cours où vous apprendrez entre autre à manipuler des structures plus complexes: des queues, des arbres, des graphes, etc.

Les exceptions

Le problème

Voilà enfin, comme promis plusieurs fois déjà, les notions de base relatives à la gestion des exceptions. Ces possibilités ne s'appliquent que pour des unités C++ et donc pas en C!

L'idée: toute fonction, compilée séparément ou non, peut éventuellement détecter des erreurs lors de son exécution. Si elle le peut, la fonction résoudra elle-même, en interne, ces problèmes. Très souvent, la fonction est dans l'impossibilité de trouver une solution à une telle situation, seul l'appelant peut prendre les décisions qui s'imposent; mais encore faut-il qu'il sache que problème il y a eu!

En fait les exceptions vont nous offrir la possibilité de signaler à l'appelant le problème et ainsi lui permettre de ne pas continuer son exécution normale à l'instruction qui suit l'appel, mais dans une zone particulière dont les instructions ne sont pas exécutées si aucune exception n'a été levée.

Comme indiqué ci-dessus, en général le problème se pose entre sous-programme appelé et l'appelant. Néanmoins nous pouvons lever et traiter localement une exception. Bien que cette situation soit plus rare, nous commencerons par elle pour introduire les aspects théoriques de base.

Traitement d'une exception

La ou les instructions que nous désirons contrôler, en d'autres termes pour lesquelles nous voulons pouvoir traiter les exceptions, doivent être englobées dans un bloc spécifique introduit par le mot réservé **try**:

```
try
{
     // Les instructions à contrôler
     …
}
```

Un tel bloc se poursuit par un ou plusieurs blocs particuliers introduits par le mot réservé **catch** suivi entre parenthèses d'un paramètre:

```
catch ( "paramètre" )
{
    // Les instructions à exécuter si l'exception
    //   correspondant à ce paramètre a été levée
    …
}
```

Le paramètre, d'un type quelconque, prend la forme usuelle des paramètres de fonction "normale", par exemple: **catch** (**char** * message) si l'instruction ayant propagé l'exception nous transmet une chaîne de caractères ou: **catch** (**int** valeur) si elle nous transmet une valeur entière qui pourrait par exemple représenter un code d'erreur. Ici nous trichons quelque peu, car on lève rarement une exception propageant un type de base. Généralement les exceptions transmettent des objets de type classe. Nous pourrions approcher ceci en travaillant avec des types structures qui comporteraient par exemple un numéro d'erreur et un message.

Si les instructions du bloc **try** ne lèvent aucune exception, tout se passe comme si les blocs **catch** n'existaient pas; en d'autres termes, après la dernière instruction du bloc **try**, le programme se poursuit normalement à la première qui suit le dernier bloc **catch** associé.

Par contre, si les instructions du bloc **try** lèvent une exception, son exécution est abandonnée pour se poursuivre par les instructions du bloc **catch** possédant un paramètre du type de celui qui sera transmis.

Si aucun bloc **catch** ne comporte un paramètre du type approprié, l'exception se propage au bloc de niveau supérieur. Ceci jusqu'à trouver un traitement adapté ou à remonter jusqu'au niveau supérieur ce qui provoque l'arrêt brutal de l'application.

En fait si l'exception remonte jusqu'au programme principal sans trouver un traite-exception lui convenant, la fonction standard *terminate* est appelée. Son comportement dépend partiellement de l'environnement. Il s'agit d'une fonction dont on ne revient pas, elle se termine en principe par un appel (implicite ou explicite!) à *abort()*[1] qui provoque un arrêt anormal de l'application. Toutefois vous pouvez remplacer la fonction *terminate* par votre propre fonction, pour autant qu'elle respecte les règles du jeu: elle ne doit pas avoir de paramètre et se termine par un appel à *abort()*. Elle a donc la structure suivante:

```
void testTerminate ( )
{
  ...
  abort ();
}
```

[1] Une information plus précise sur son utilisation se trouve dans la suite de ce chapitre.

Vous spécifiez que cette fonction doit être appelée à la place du terminate normal par un appel à la fonction set_terminate à laquelle vous transmettez votre fonction en paramètre. Ceci donne pour notre exemple:

```
set_terminate ( testTerminate );
```

Après l'exécution des instructions du bloc **catch** choisi, l'exception n'existe plus et le programme se poursuit à l'instruction suivant le dernier bloc **catch** associé au **try**.

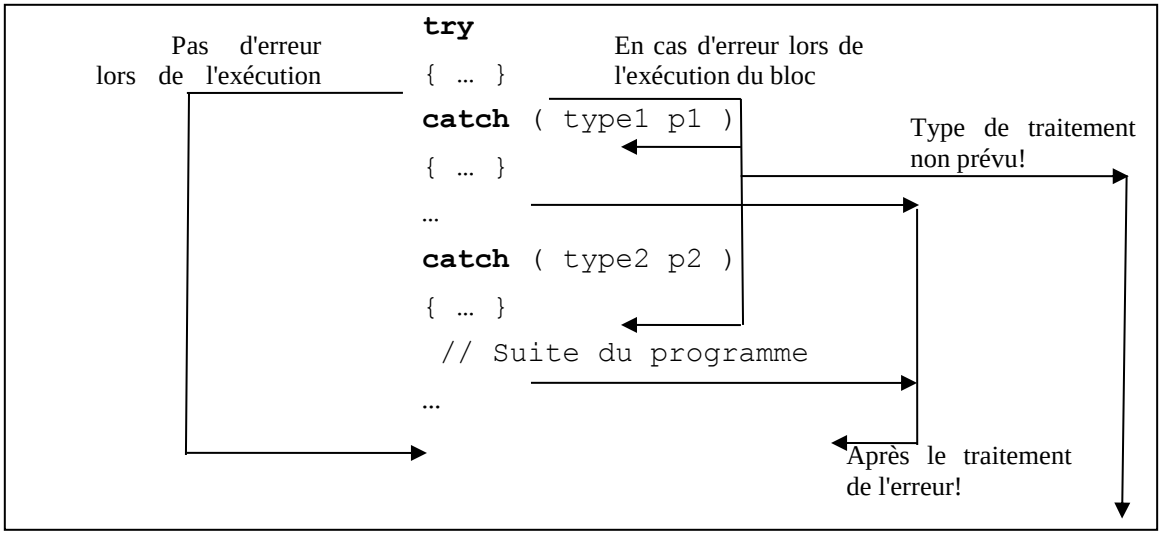
Le dernier **catch** d'un bloc **try** peut prendre la forme:

```
catch ( … )
{   // Les instructions  }
```

Les "…" dans la paire de parenthèses du catch signifiant: "toutes les autres exceptions", toutes celles propageant un paramètre d'un type qui n'a pas été explicitement prévu dans les différentes branches des catch qui précèdent.

Etrangement et bien que cela soit totalement inutile, plusieurs **catch** peuvent s'annoncer comme traitant le même type de "paramètre". Syntaxiquement cela s'avère correct, mais à l'exécution seule la première de ces branches sera toujours prise!

Nous pouvons représenter le comportement d'une telle construction par le schéma suivant:



Boucle de reprise

Nous ne pouvons pas, sans autre, redonner le contrôle à l'endroit où l'exception a été levée. Par contre nous pouvons construire une structure (boucle) nous permettant de reprendre à l'endroit désiré et de laquelle nous sortons (brutalement!) lorsque tout s'est bien passé.

Nous construisons cette reprise par l'intermédiaire d'une boucle (par exemple **while**) que nous quittons par un **break** à la fin du bloc **try** si tout se passe bien. Ceci nous donne une structure du genre:

```
/* Tant qu'il y a des exceptions (des erreurs)! */
while (1)
{
  try  // Bloc controle
  {
    …
    break;  // OK, pas de probleme, on sort!
  }
  catch ( const char *Message )
  {
    …
  }
  /* On recommence en cas de probleme! */
}
/* On continue ici quand tout s'est bien passe */
```

Terminaisons brutales

Parfois en cas d'erreur (d'exception) la situation s'avère totalement désespérée. Le programmeur ne sait plus quoi faire à part éventuellement:

- Sauver des informations, par exemple dans un fichier.
- Faire part de son désespoir à l'utilisateur par un message.

Après cela il ne peut qu'arrêter son application par un appel à l'une des fonctions de *cstdlib* (*stdlib.h* en C):

- `void abort ( void );`
- `void exit ( int status );`

La première de ces fonctions arrête brutalement le programme sans prendre de précautions particulières. Entre-autres, la fermeture des fichiers ne sera certainement pas faite proprement.

La deuxième permet en plus de transmettre au système un code de terminaison (comme pour le **return** mis dans le programme principal!).

Notons toutefois que souvent on termine un programme par:

```
return 0;
```

Strictement parlant ceci n'est pas propre. Bien que cela soit généralement le cas, rien ne garantit que la valeur 0 corresponde à une terminaison sans problème.

En fait *cstdlib* met entre autre à disposition 2 constantes:

- *EXIT_SUCCESS* que nous devons utiliser aussi bien avec *exit* que **return** pour spécifier une terminaison correcte.
- *EXIT_FAILURE* à utiliser pour indiquer une fin en erreur.

De plus *exit* offre aussi la possibilité, par l'intermédiaire de la fonction:

```
int atexit ( void ( * function ) (void) );
```

d'indiquer que la fonction transmise en paramètre (elle n'a pas de paramètre et ne livre pas de résultat) doit être automatiquement appelée lors d'une fin normale du programme. Pour exécuter automatiquement non pas une mais plusieurs fonctions, il faut avoir appelé plusieurs fois *atexit* (une fois pour chaque fonction à exécuter). L'ordre d'exécution de ces fonctions lors de la terminaison sera l'inverse de celui de leur installation par *atexit*.

Voici un petit exemple rudimentaire devant vous aider à comprendre ce comportement:

```
/*
   Utilisation de: exit, atexit
     EXIT
*/
#include <cstdlib>
#include <iostream>
using namespace std;

void sp1 ( )
{
  cout << "SP1debut" << endl;
  cout << "SP1fin" << endl;
  system ( "pause" );
}

void sp2 ( )
{
  cout << "SP2debut" << endl;
  cout << "SP2fin" << endl;
  system ( "pause" );
}

int main ( )
{
  cout << "Debut" << endl;
  atexit ( sp1 );
  atexit ( sp2 );

  /* Pour terminer, non standard!!! */
  cout << "\nFin du programme...";
  system ( "pause" );
//  exit ( EXIT_FAILURE );
  exit ( EXIT_SUCCESS );
//  return EXIT_SUCCESS;
}
```

qui donne le résultat suivant lors de son exécution:

```
Debut

Fin du programme...Appuyez sur une touche pour continuer...
SP2debut
SP2fin
Appuyez sur une touche pour continuer...
SP1debut
SP1fin
Appuyez sur une touche pour continuer...
```

Tout ce que nous venons de dire est valable pour la terminaison dite normale d'un programme, même si nous transmettons *EXIT_FAILURE* comme valeur!

❑ *Lever/propager une exception*

Bien. Nous savons maintenant comment traiter une exception. Mais comment lever une telle exception?

Lorsque nous voulons arrêter l'exécution normale et propager une exception, nous utilisons l'instruction **throw** suivie d'une valeur d'un des types en principe détecté par l'un des blocs **catch**. L'instruction **throw** s'utilise donc comme une fonction qui possèderait un paramètre:

```
throw "paramètre";
```

Un exemple d'application totalement artificielle, mais devant nous aider à comprendre le comportement des exceptions:

```
/*
  Exemple artificiel de traitement des exceptions
  EXCEPTIONS1
*/
#include <cstdlib >
#include <iostream>
using namespace std;
int main ( )
{
  unsigned int aiguillage;
  cout << "Programme exemple: exceptions" << endl;
  /* Tant qu'il y a des exceptions (des erreurs)! */
  while (1)
  {
    try    // Bloc controle
    {
      cout << "\nDonnez une valeur entiere: ";
      cin >> aiguillage;

      /* En fonction du choix utilisateur */
      switch ( aiguillage )
      { /* Ici les break ne sont pas utiles!!! */
        case 1: // On propage un message
          throw "Erreur 1";
        case 2: // On propage une valeur reelle
          throw 3.5;
        case 3: // On propage l'adresse d'un entier
          throw &aiguillage;
        /* Les autres cas: OK, ne rien faire */
      }
      break;  // OK, pas de probleme, on sort!
    }
```

```
    /* Recuperation d'une chaine de caracteres */
    catch ( const char * message )
    {
      cout << "---" << message << endl;
    }
    /* Recuperation d'une valeur reelle */
    catch ( const double valeur )
    {
      cout << valeur << endl;
    }
    /* Recuperation de l'adresse d'un entier */
    catch ( unsigned int *valeur )
    {
      cout << "Au debut du traite-exception " << *valeur << endl;
      *valeur += 1;
    }
    cout << "Apres traitement de l'exception " << aiguillage << endl;
  }

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Exemple d'exécution:

```
Programme exemple: exceptions

Donnez une valeur entiere: 1
---Erreur 1
Apres traitement de l'exception 1

Donnez une valeur entiere: 2
3.5
Apres traitement de l'exception 2

Donnez une valeur entiere: 3
Au debut du traite-exception 3
Apres traitement de l'exception 4

Donnez une valeur entiere: 4

Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *Retour sur les fonctions*

Nous l'avons signalé dès le départ, généralement une fonction détecte un problème et ne peut pas traiter l'erreur localement. Elle va donc lever une exception qu'elle ne traitera pas elle-même et qui sera propagée à l'unité appelante. On espère de la sorte qu'au niveau supérieur le problème puisse se résoudre. L'appelant traitera l'exception selon les mécanismes que nous avons décrits ci-dessus.

Mais comment construire la fonction dans ce but?

Premièrement nous pouvons construire notre fonction strictement comme nous en avons l'habitude, simplement en lui ajoutant les instructions **throw** désirées. Sous cette forme la fonction a le droit de propager n'importe quelle exception, y compris celles qui lui viendrait (éventuellement sans qu'elle le sache!) de l'appel d'une autre fonction.

Si nous reprenons notre programme exemple précédent, mais en faisant en sorte que les exceptions soient levées par une fonction, nous obtenons un code du genre:

```
/*
  Exemple artificiel de traitement des exceptions
  EXCEPTIONS2
*/
#include <cstdlib>
#include <iostream>
using namespace std;
/*  Fonction provoquant artificiellement des exceptions
    de divers types
*/
void artificielle ( unsigned int &param )
{
  cout << "\nDonnez une valeur entiere: ";
  cin >> param;
  /* En fonction du choix utilisateur */
  switch ( param )
  { /* Ici les break ne sont pas utiles */
    case 1: // On propage un message
      throw "Erreur 1";
    case 2: // On propage une valeur reelle
      throw 3.5;
    case 3: // On propage l'adresse d'un entier
      throw &param;
    /* Les autres cas: OK, ne rien faire */
  }
}
```

```
int main ( )
{
  unsigned int aiguillage;
  cout << "Programme exemple: exceptions" << endl;
  /* Tant qu'il y a des exceptions (des erreurs)! */
  while (1)
  {
    try    // Bloc controle
    {
      artificielle ( aiguillage );
      break;  // OK, pas de probleme, on sort!
    }
    /* Recuperation d'une chaine de caracteres */
    catch ( const char * message )
    {
      cout << "---" << message << endl;
    }
    /* Recuperation d'une valeur reelle */
    catch ( const double valeur )
    {
      cout << valeur << endl;
    }

    /* Recuperation de l'adresse d'un entier */
    catch ( unsigned int *valeur )
    {
      cout << "Au debut du traite-exception " << *valeur << endl;
      *valeur += 1;
    }
    cout << "Apres traitement de l'exception " << aiguillage
         << endl;
  }

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Son exécution nous donne les mêmes résultats que la première version

Nous pouvons aussi compléter la ligne d'en-tête d'une fonction en précisant explicitement la liste de toutes les exceptions susceptibles d'être propagées vers l'appelant. Dans notre exemple, en appliquant ce principe l'en-tête de la fonction artificielle prendrait la forme:

```
void artificielle ( unsigned int &param )
          throw ( const char *, const double, unsigned int * )
{ ... }
```

Toutefois nous déconseillons cette possibilité car si une exception non prévue dans la liste survient le programme s'arrêtera comme s'il n'y avait pas de traitement d'exception. N'oubliez pas que si votre fonction appelle d'autres fonctions (et il y a de fortes chances qu'elle le fasse), des exceptions peuvent provenir de ces fonctions appelées! En fait l'arrêt ne survient pas aussi brutalement que cela; une fonction *unexpected* est appelée. Malheureusement son comportement par défaut peut varier, la norme ne le définissant pas clairement. Toutefois le programmeur a la possibilité de définir sa propre fonction à utiliser dans une telle situation. Cette fonction doit respecter les caractéristiques suivantes:

- Elle n'a pas de paramètre.
- Elle peut se terminer en levant une exception (**throw**).
- Elle ne livre pas de résultat (**void**) et n'a pas de **return** donc si elle ne lève pas une exception elle termine brutalement l'application par un appel à *exit* ou *abort*.

Un exemple théorique d'une telle fonction:

```
void testUnexpected ( )
{
  cout << "Dans unexpected\n";
  throw "Vient de unexpected\n";
}
```

Pour remplacer *unexpected* par notre propre fonction nous devons "l'installer" par un appel à *set_unexpected*. Ce qui donnerait avec notre exemple ci-dessus:

```
set_unexpected ( testUnexpected );
```

Si nous désirons préciser que notre fonction ne propage aucune exception (mais en sommes-nous réellement certains?) nous compléterons la ligne d'en-tête de la fonction par le mot réservé **throw** suivi de parenthèses vides, ce qui donnerait dans notre exemple:

```
void artificielle ( unsigned int &param ) throw ( )
{ ... }
```

Si vous utilisez un prototype pour votre fonction sa partie **throw** doit correspondre strictement à ce qui se trouve dans sa définition.

❑ *Compléments*

Bien que la situation ne se présente que rarement, nous pouvons imbriquer les blocs **try**. Dans ce cas le comportement entre bloc imbriqué et englobant correspond à ce qui se passe entre une fonction et son appelant.

Dans un traite-exception (un bloc **catch**) nous pouvons aussi lever une exception, celle qui nous a amené dans ce bloc ou une autre. Cette possibilité s'avère très utile pour commencer un traitement que nous ne sommes pas capables de terminer à l'endroit en question. Si nous désirons propager la même exception que celle nous ayant amené dans le bloc, nous pouvons utiliser une forme simplifiée de l'instruction **throw** sans paramètre; le paramètre initial sera alors transmis.

Le fait qu'un bloc **catch** annonce un paramètre passé par copie ou par référence, de même qu'il soit constant ou pas n'interfère en rien sur sa sélection. De toute façon une copie de la valeur ou de la référence sera transmise. Cela signifie que pour une instruction **throw** transmettant un paramètre de type: *type*, les blocs **catch** suivants:

catch (type val), **catch** (type & val), **catch** (**const** type val) , **catch** (**const** type & val) conviennent parfaitement.

Lorsqu'une fonction lève une exception, ses variables locales sont proprement détruites avant de propager l'exception, comme pour une terminaison normale de la fonction. Il n'en va pas de même pour les variables créées dynamiquement et dont la gestion repose sur l'entière responsabilité du programmeur.

Comme de nombreux autres aspects de C++, en pratique les exceptions sont étroitement liées à la notion de classe. Nous aurons donc l'occasion d'y revenir par la suite.

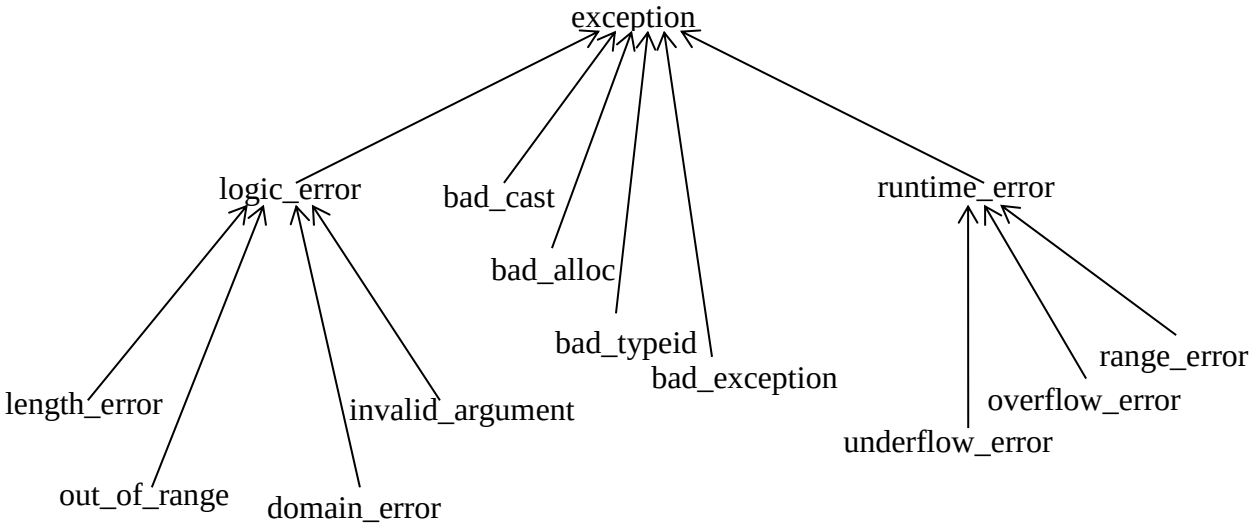
❑ *Classe exception et exceptions prédéfinies*

En réalité il existe une classe exception (mais nous n'avons pas étudié cette notion!). Elle correspond à la définition que nous vous donnons ci-dessous un peu brutalement:

```
class exception
{
  public:
  exception () throw ();
  exception ( const exception & ) throw ();
  exception & operator = ( const exception & ) throw ();
  virtual ~exception () throw ();
  virtual const char * what () const throw ();
};
```

En fait une exception ne devrait jamais être d'un type simple, comme nous l'avons fait pour nos exemples qui précèdent dans ce chapitre.

Toutes les exceptions standard, héritées de la bibliothèque dérivent de cette classe (mais voilà encore une opération que nous ne savons pas faire!). Elles correspondent à la hiérarchie suivante:



Nous pouvons, sans autres, lever ces exceptions dans nos programmes et utiliser les

différents outils associés. Dans ce but, il faut simplement ajouter l'inclusion: *#include <stdexcept>.*

La structure hiérarchique de la définition de ces exceptions présente comme premier avantage la possibilité de pouvoir traiter une exception à différents niveaux et ainsi par exemple de regrouper, dans un même "traite-exceptions", plusieurs exceptions pour un traitement global, avant par exemple de réorienter la suite du traitement dans des branches spécifiques.

Autre point intéressant, la classe *exception* met à disposition la méthode *what* (ne vous inquiétez pas trop pour l'instant, ni de la syntaxe, ni de la sémantique associée au mot **virtual**!). Cette fonction permet d'obtenir, sous forme d'une chaîne de caractères, un message que l'on peut toujours associer à l'exception.

```
/*
  Exemple exception standard
    EXCEPTIONSTD.cpp
*/
#include <cstdlib>
#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;

/* Affiche 1 élément du vecteur transmis en paramètre */
void affiche ( vector<int> v, int i )
{
  cout << v.at(i) << endl;
}

int  main ( )
{
  try
  {
    vector<int> v1 ( 2 );
    v1.at(0) = 5; v1.at(1) = 2;
    cout << "Exception standard" << endl << endl;
    affiche ( v1, 2 );
  }
  catch ( exception &e )
  {
    cout << "*** " << e.what () << endl;
  }
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Qui affiche:

```
Exception standard

*** vector::_M_range_check

Fin du programme...Appuyez sur une touche pour continuer...
```

Note: On obtient le même résultat avec un:

```
  catch ( out_of_range &e ) ...
```

puisque l'exception *out_of_range* dérive indirectement de *exception*!

Ou encore, avec:

```
  catch ( logic_error &e )  ...
```

puisque l'exception *out_of_range* dérive de *logic_error* qui dérive de *exception*!

Autre possibilité intéressante: lorsque nous levons une exception, nous pouvons lui associer un message qui sera récupéré par la méthode *what*. Ceci se fait simplement en complétant l'instruction **throw** par le message désiré, exemple:

```
throw logic_error ( "Message!!" );
```

Modifions quelque peu notre programme exemple ci-dessus pour illustrer ce point.

```
/*
  Exemple exception standard
    EXCEPTIONSTD2.cpp
*/
#include <cstdlib>
#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;

/* Affiche 1 élément du vecteur transmis en paramètre,
   avec propagation d'exception si nécessaire
*/
void affiche ( vector<int> v, int i )
{
  try
  {
    cout << v.at(i) << endl;
  }
  catch ( exception &e )
  {
    cout << "-- " << e.what () << endl;
    throw logic_error ( "Erreur dans affiche!!" );
  }
}


int  main ( )
{
  try
  {
    vector<int> v1 ( 2 );
    v1.at(0) = 5; v1.at(1) = 2;
    cout << "Exception standard" << endl << endl;
    affiche ( v1, 2 );
  }
  catch ( logic_error &e )
  {
    cout << "*** " << e.what () << endl;
  }
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Qui cette fois affiche:

```
Exception standard

-- vector::_M_range_check
*** Erreur dans affiche!!

Fin du programme...Appuyez sur une touche pour continuer...
```

Dans cette version, pour le traite-exceptions dans le programme nous pouvons aussi écrire:

```
  catch ( exception &e ) ...
```

avec le même résultat final. Par contre nous ne pouvons pas utiliser l'exception *out_of_range* puisque nous propageons une exception d'un niveau hiérarchique plus élevé!

Nous arrêtons là cette brève présentation. Par la suite d'autres possibilités s'offriront à nous pour définir nos propres exceptions dérivées/héritées de exception ou de ses descendants

Les espaces de noms: Namespace

Introduction

Nous avons déjà utilisé intuitivement la notion d'espace de noms pour accéder à certains éléments de la bibliothèque. Dans ce but tous nos programmes comportaient jusqu'à maintenant une clause:

```
using namespace std;
```

car tous les éléments de la bibliothèque standard se trouvent définis dans l'espace *std* (pour standard). Nous allons maintenant formaliser quelque peu ces notions.

Lorsque nous développons une application importante en taille, découpées en de nombreux modules, il y a alors une probabilité non négligeable de retrouver des identificateurs globaux de mêmes noms, ce qui provoquera des erreurs lors de l'édition de liens, sauf s'il s'agit de surcharges de fonctions. Les espaces de noms visent à supprimer ces problèmes en offrant à chaque module son propre espace de définitions.

Création d'un espace de noms

Un espace de noms se déclare globalement à une unité. Il s'introduit par le mot réservé **namespace** et prend la forme générale suivante:

```
namespace identificateur
{
   // Vos déclarations / définitions
}
```

identificateur: le nom donné à cet espace.

Le contenu de l'espace de noms (mis entre {}) consiste en déclarations sous leurs formes usuelles de types, de constantes, de variables, de fonctions[1] et éventuellement d'espaces de noms puisqu'ils peuvent s'imbriquer.

Par définition, à l'intérieur d'un espace de noms donné les identificateurs doivent être uniques, sauf éventuelles surcharges de fonctions.

Exemple:

```
namespace espace1
{
  typedef  unsigned uint;
  int   i = 10;
  float f = 5.3;
  int fct ( int );
}
```

Pour les fonctions, 2 possibilités s'offrent au développeur:

- Donner la définition complète de la fonction (son corps) dans la définition de l'espace de noms. Dans ce cas cette définition prend sa forme usuelle.
- Ne donner que la déclaration (le prototype) de la fonction dans l'espace de noms et fournir sa définition à l'extérieur de celui-ci. Dans ce cas la définition doit faire référence de son appartenance à l'espace en question au moyen de l'opérateur de résolution de portée. La fonction *fct* de notre exemple ci-dessus prendrait donc la forme:

  ```
  int espace1::fct ( int p )
  {  ...
    return ...;
  }
  ```

Attention: si vous omettez de préciser l'appartenance à l'espace de noms, vous réalisez alors une autre fonction et celle de votre espace restera indéfinie.

Une unité de compilation peut mettre à disposition plusieurs espaces de noms différents.

[1] Et par la suite de classes.

Extension d'un espace de noms

Au sein d'une unité un espace de noms peut être étendu, en d'autres termes on peut le construire en plusieurs morceaux, mais tout en conservant le principe de globalité. On peut imaginer une structure (volontairement un peu complexe) du genre:

```
namespace espace1
{  ...  }

void sp ( )
{  ...  }

namespace espace2
{  ...  }

namespace espace1
/* Les déclarations qui suivent completent la premiere
    partie de espace1
*/
{  ...  }

int main ( )
{  ...  }
```

En pratique cette possibilité s'avère très utile lors de la réalisation d'une bibliothèque comportant plusieurs fichiers d'inclusion. Chacun de ces fichiers pouvant mettre à disposition de nouveaux éléments dans un espace unique.

Espace anonyme

A première vue étrange, mais en réalité pas tant que cela, un espace de noms peut rester anonyme:

```
namespace
{
   // Vos déclarations / définitions
}
```

Tout se passe de telle sorte que le système nous garantisse un nom spécifique unique pour cet espace. Bien qu'étant globaux (les espaces de noms le sont toujours!) les éléments déclarés dans cet espace de noms garderont une portée locale à l'unité de compilation. En fait nous disposons là d'une alternative intéressante aux variables globales de classe **static**. En effet ces variables ne seront utilisables que dans l'unité où l'on définit l'espace de nom en question.

Chaque unité de compilation a le droit de posséder son propre espace anonyme qui n'interfère pas avec ceux, aussi anonymes, des autres unités de compilation.

❑ Alias

Le nom donné à un espace peut vous sembler: long, compliqué, peu parlant ou tout simplement ne pas vous convenir (n'oubliez pas que ces noms vont très souvent venir d'unités que vous n'avez pas développées vous-même).

Pour améliorer cette situation vous allez tout simplement créer un alias du nom qui ne vous convient pas.

Cette création prend la forme:

```
namespace nom_d_alias = nom_d_espace;
```

Exemple:

```
namespace standard = std;
```

A partir de cette instruction nous pouvons utiliser le nom *standard* à la place de *std*, y compris dans l'instruction **using namespace** que nous utilisons depuis le début de notre cours et dont nous allons clarifier les possibilités dans la suite de ce chapitre.

❑ *Utilisation des objets d'un espace de noms*

Pour utiliser les objets définis dans un espace de noms il suffit de préfixer le nom de l'objet avec celui de l'espace au moyen de l'opérateur de résolution de portée:

nomEspace::nomObjet

Exemple:

```
cout << espace1::i << endl;
```

Ceci permet non seulement d'avoir des identificateurs identiques dans des espaces différents, mais également d'avoir une variable locale possédant le même nom.

```
namespace espace1
{
  int   i = 10;
  float f = 5.3f;
}

int i = 1;   // Variable de l'espace global

int main ( )
{
  int i = 3;
  cout << espace1::i << endl;
  cout << i          << endl;  // Le i local!!
  cout << ::i        << endl;  // Le i global!!
}
```

Une variable globale définie hors espace de noms appartient en fait à l'espace dit "global". On peut la désigner par l'opérateur :: sans nom d'espace[1]

[1] Cette forme sert essentiellement à lever une éventuelle ambiguïté.

Directive using

En pratique il sera certainement fastidieux de toujours devoir préfixer ainsi chaque utilisation d'un objet. Comme nous l'avons toujours fait pour l'espace de noms *std*, nous pouvons "ouvrir" globalement un espace afin de donner un accès direct à l'ensemble de ses éléments.

Une telle opération se réalise par l'intermédiaire de la directive **using namespace**:

```
using namespace nomEspace;
```

Exemple:

```
using namespace espace1;
```

Dans nos exemples nous avons toujours mis de manière globale notre directive:

```
using namespace std;
```

Nous pouvons aussi utiliser la directive localement à un bloc de telle manière à en limiter la portée à celui-ci.

Des conflits peuvent survenir entre 2 espaces de noms pour lesquels nous aurions utilisé une directive **using namespace** et qui comporteraient le même identificateur. Toutefois aucune erreur ne sera signalée à la compilation si nous ne tentons pas d'utiliser un tel objet.

Bien qu'ayant ouvert totalement un espace de noms, rien ne nous empêche de préciser explicitement son appartenance à celui-ci par l'intermédiaire de l'opérateur de résolution de portée. Une telle formulation permet de lever des ambiguïtés!

Une directive **using namespace** peut s'utiliser dans la définition d'un espace de noms. Dans ce cas l'opération est transitive. Donc si dans la définition d'un espace (disons *espace1*) nos introduisons une clause:

```
using namespace espace2;
```

l'unité qui fera un:

```
using namespace espace1;
```

héritera aussi de l'ouverture complète d'espace2, ce qui malheureusement multiplie les risques de conflits.

Instruction using

Une solution intermédiaire entre l'ouverture complète d'un espace de noms et la référence explicite à l'espace pour l'utilisation de chaque objet consiste à utiliser l'instruction **using.** Elle permet de rendre directement accessible un seul objet spécifique de l'espace.

Sa forme générale:

```
using nomEspace::nomObjet;
```

Exemple:

```
using espace1::i;
```

A nouveau nous pouvons rendre une telle instruction locale ou globale suivant la portée désirée.

Avec cette instruction le compilateur signale immédiatement tout conflit. Précisons cette notion de conflit. Si vous utilisez une instruction **using** globale pour un objet (disons *i*), rien ne vous empêche de déclarer une variable locale *i* puisqu'un objet local masque la visibilité d'un objet global de même nom. Par contre si les 2 éléments sont tous 2 globaux ou tous 2 locaux alors il y aura conflit.

Rappel final: les espaces de noms sont réellement utiles dans le contexte de développement d'applications de tailles importantes.

Complément aux entrées/sorties interactives

Introduction

Jusqu'à présent nous n'avons guère accordé d'importance à la mise en forme de nos affichages ni d'ailleurs aux possibilités supplémentaires offertes lors de lectures. Ce chapitre, que nous considérons comme transitoire avant d'aborder de manière générale les fichiers, va nous permettre de préciser quelque peu nos connaissances dans ce domaine. Nous le décomposerons en 2 parties essentielles:

- La première, par l'intermédiaire de manipulateurs, complète les possibilités offertes sur les flux *cin* et *cout* et ils restent donc utilisables uniquement en C++.
- La deuxième nous permettra d'introduire les fonctions de bases dans ce domaine (*prinf* et *scanf*) utilisables, elles, en C et en C++.

Les manipulateurs

Nous avons déjà utilisé l'un d'entre eux dans tous nos programmes:

- endl

qui nous permet de passer à la ligne sur le flux de sortie *cout*. En fait, un manipulateur consiste en un objet qui, envoyé sur un flux, en modifie le comportement.

Nous allons maintenant vous donner une brève description de la majorité de ces manipulateurs.

- oct, dec, hex

Comme leur nom le laisse supposer, ces manipulateurs permettent de fixer en quelle base seront affichées les valeurs entières suivantes:

```
cout << 17 << ' ' << oct << 17 << ' ' << 17
     << ' ' << dec << 17 << ' ' << 17
     << ' ' << hex << 17 << ' ' << 17 << endl;
```

Affiche:

```
17 21 21 17 17 11 11
```

Ils sont également utilisables en entrée. Une exception sera levée si l'utilisateur ne fournit pas une valeur correcte pour la base en question.

Certains de ces manipulateurs possèdent des paramètres. Pour l'utilisation de ceux-ci il faut ajouter à l'unité de compilation le fichier d'inclusion *iomanip.* Il existe une deuxième forme pour fixer la base:

- setbase (**int** base)

où base peut prendre les valeurs 8, 10 et 16 (comportement indéterminé pour les autres valeurs!).

- showbase, noshowbase

Demande d'afficher la base avec la valeur, en d'autres termes les valeurs octales sont précédées d'un 0 et celles en hexadécimal de 0x:

```
cout << showbase << 17 << ' ' << setbase(8) << 17 << ' '
     << 17 << ' ' << setbase(10) << 17 << ' ' << 17
     << ' ' << setbase(16) << 17 << ' ' << 17 << endl;
```

Affichera:

```
17 021 021 17 17 0x11 0x11
```

- uppercase, nouppercase

Les lettres apparaissant dans l'affichage des valeurs hexadécimales le seront respectivement en majuscules ou en minuscules:

```
cout  << hex << 12 << uppercase << 12 << endl;
```

Affiche:

```
cC
```

- boolalpha, noboolalpha

Par défaut les valeurs booléennes s'affichent sous forme respectivement de 0 et de 1 (on les lit aussi sous cette forme). Le premier de ces manipulateurs permet de les afficher (ou de les lire) sous la forme respective de **true**, **false**. Le deuxième nous fait revenir à la forme 0/1.

```
cout  << true << ' ' << false << ' ' << boolalpha
      << true << ' ' << false << endl;
```

Affiche:

```
1 0 true false
```

- setw (**int** val)

Uniquement valable pour la prochaine valeur traitée, *setw* permet de fixer le nombre de colonnes utilisées[1]:

```
for ( int i = 0; i <= 3; i++ )
  cout << '*' << setw ( i*3 ) << i*10 << '*' << endl;
```

Affiche:

```
*0*
* 10*
*    20*
*       30*
```

Note: Si la place demandée se révèle insuffisante, le minimum nécessaire sera de toute façon pris (comme cela se passe normalement par défaut!).

- left, right, internal

En sortie uniquement et étroitement lié à *setw*. Fixe la manière d'ajuster la valeur à afficher dans le champ mis à disposition. Relevez le cas un peu particulier de *internal* qui met le signe (+/-) à gauche (ou la base si demandée), ajuste la valeur à droite et ajoute les éventuels espaces nécessaires entre deux:

```
cout << '*' << setw ( 10 ) << left     << -10.2 << '*' << endl;
cout << '*' << setw ( 10 ) << internal << -10.2 << '*' << endl;
cout << '*' << setw ( 10 ) << right    << -10.2 << '*' << endl;
```

Affiche:

```
*-10.2     *
*-     10.2*
*     -10.2*
```

- showpos, noshowpos

En sortie uniquement, force/supprime l'affichage du signe + devant une valeur numérique positive.

[1] En lecture n'a de sens que pour limiter la longueur d'une chaîne.

- setfill (**char** c)

Spécifie le caractère de remplissage à utiliser à la place de l'espace lorsque c'est nécessaire:

```
cout << setfill ( '*' )
```

pour obtenir des étoiles à la place des espaces.

- fixed, scientific

En sortie, détermine la manière d'afficher les nombres réels:

```
cout << 1.23 << '\t' << fixed << 1.23 << '\t'
     << scientific << 1.23 << endl << fixed << 1.23e9
     << '\t' << scientific << 1.23e9 << endl;
```

Affiche:

```
1.23    1.230000        1.230000e+000
1230000000.000000       1.230000e+009
```

- setprecision (**int** val)

En sortie fixe le nombre de chiffres significatifs à afficher. A utiliser avec modération, certains effets pouvant paraître surprenants:

```
cout << setprecision ( 5 )
     << 1.234 << '\t' << fixed << 1.234 << '\t'
     << scientific << 1.234 << endl << fixed << 1.234e-9
     << '\t' << scientific << 1.234e9 << endl;
```

Affiche:

```
1.234   1.23400 1.23400e+000
0.00000 1.23400e+009
```

- showpoint, noshowpoint

Pour les valeurs réelles ne possédant pas de partie fractionnaire, permet d'afficher ou non le point décimal:

```
cout << 12.0 << '\t' << 12.3 << '\t' << showpoint
     << 12.0 << '\t' << 12.3 << endl;
```

Affiche:

```
12      12.3    12.0000 12.3000
```

- skipws, noskipws

Pour la lecture uniquement: permet de sauter (ce qui se fait par défaut!) ou non les caractères blancs. Dans le cas de valeurs numériques, *noskipws* ne paraît pas très intéressant car pour l'instruction:

```
cin >> i1 >> noskipws >> i2;
```

seule la première valeur sera lue correctement!

Par contre dans le cas de lectures de caractères ou de chaînes, suivant les besoins l'opération peut devenir intéressante.

- ends

Envoie en sortie un caractère de fin de chaîne ('\0') pas très intéressant pour un affichage à l'écran mais pourra le devenir lors de l'écriture dans un fichier.

- flush

Vide le tampon de sortie: envoie physiquement son contenu sur le flux de sortie.

Note: l'utilisation de ces manipulateurs se généralisera dans la suite pour le traitement des fichiers.

Reconnaissons qu'au niveau du formatage ces possibilités ne sont pas forcément très agréables à utiliser. Globalement les anciennes fonctions offertes par C, également utilisables en C++, s'avèrent plus directes mais malheureusement plus difficiles à maîtriser au départ et certainement aussi moins lisibles.

Dans la suite de ce chapitre nous vous présentons les formes de base pour l'affichage à l'écran et la lecture au clavier des fonctions héritées de C. Tout comme pour les flux *cin* et *cout* ces aspects se généraliseront relativement facilement aux fichiers que nous aborderons par la suite.

❑ *printf et scanf*

❑ printf

printf, l'une des multiples fonctions d'entrée/sortie de la bibliothèque, comme les autres, nécessite pour son utilisation d'inclure:

```
#include <cstdio>;  // [1]
```

La fonction *printf*, du point de vue de ses paramètres, se décompose en 2 parties:

```
printf ( "Chaîne de contrôle", liste_de_valeurs );
```

Chaîne de contrôle: une chaîne de caractères représentant le texte que l'on veut afficher avec en plus à l'intérieur, aux endroits désirés dans le texte affiché, des descripteurs de format pour les éléments de la liste de valeurs.

Liste_de_valeurs: 0, 1 ou plusieurs paramètres (expressions) représentant les objets dont on désire afficher les valeurs. Cette liste n'existe pas si l'on désire afficher que du texte:

```
printf ( "\nSalut\n" );
```

La chaîne de contrôle, comme le montre l'exemple, peut elle-même comporter des caractères spéciaux de contrôle afin d'améliorer la mise en page. Dans l'exemple ci-dessus, on passe à une nouvelle ligne (\n), on affiche le texte (Salut) puis on passe à nouveau à la ligne (\n).

[1] En C: #include <stdio.h>

Descripteurs de format

Les descripteurs de format s'introduisent par le caractère % qui n'est donc pas affiché. Les descripteurs possibles sont les suivants:

Descripteur:	**Explication:**
%d	Pour une valeur entière (en base 10)
%i	Pour une valeur entière (en base 10), idem à %d
%c	Pour un caractère
%f	Pour une valeur réelle (virgule flottante)
%e	Pour une valeur réelle en notation scientifique
%E	Idem à e mais avec un E pour introduire la puissance de 10
%g	Choix automatique entre f et e
%G	Idem à g mais avec un E pour la puissance de 10 si nécessaire
%s	Pour une chaîne de caractères
%u	Pour une valeur entière non signée
%o[1]	Pour une valeur entière affichée en octal
%x[2]	Pour une valeur entière affichée en hexadécimal
%X	Idem à x mais avec des lettres majuscules
%%	Pour afficher le caractère % lui-même (1 seul affiché!)

Il suffit de mettre le descripteur désiré à l'endroit désiré dans la chaîne de contrôle:

```
...
int i = 3;
...
printf ( "\n Nous vous offrons du %d %%", i + 2 );
```

[1] %#o force l'affichage du 0 de tête!

[2] %#x force l'affichage du 0x de tête!

Affiche:

```
Nous vous offrons du 5 %
```

Notez qu'il doit y avoir le même nombre de descripteurs de format dans la chaîne de contrôle qu'il y a d'éléments dans la liste de valeurs et qu'ils doivent se correspondre en types, ou tout au moins en types compatibles! Si ce n'est pas le cas, le comportement est indéterminé!

Entre le symbole % et la lettre du descripteur, on peut introduire une constante précisant le champ (nombre de positions) à utiliser pour afficher la valeur:

```
printf ( "%6d", 33 );
```

Affichera la valeur 33 précédée de 4 espaces (en tout 6 positions).

Règles complémentaires:

- Si le champ spécifié est trop petit pour la valeur à afficher, il sera automatiquement étendu à la valeur nécessaire, sauf pour les chaînes de caractères qui elles sont tronquées.
- Si la constante de champ est négative, la valeur affichée sera justifiée à gauche du champ.
- Si un signe "+" est mis après le %, le signe sera toujours mis devant la valeur affichée. Par défaut seul le signe "-" s'affiche!
- Si la constante de champ commence par un zéro, les positions du champ non occupées par la valeur seront complétées par des zéros et non des espaces.

Pour les nombres réels, en plus de la taille du champ, nous pouvons aussi préciser le nombre de chiffres après la virgule en mettant deux constantes séparées par un point:

```
%9.2f
```

Affiche une valeur réelle (**float**) sur 9 positions en tout et avec 2 chiffres après la virgule.

Le caractère * peut remplacer les constantes de formatage. Exemples:

%*.*f %*.5f %2.*f %*d

Pour chaque * présente, la prochaine valeur de la liste n'est pas affichée, mais utilisée comme valeur de format!

Donc:

```
printf ( "%*.*f\n",  12, 3, 9.5 );
```

Affichera 9.500 complété avec des espaces devant.

La première étoile sera remplacée par la valeur 12 et la deuxième par 3. Sous cette forme (avec des constantes) cela ne présente aucun intérêt; en pratique ces constantes seront remplacées par des expressions évaluées à l'exécution, donc le format d'affichage pourra changer en cours d'exécution.

Résumons la syntaxe d'un descripteur de format de la manière suivante:

```
% [ - + ] [ N ] [.n ] [ M ] type
```
[1]

.

Avec:

- **N** et **n**: nombres entiers
- **M**: la lettre "l" dans le cas d'un entier long (ou L pour un **long double**), la lettre "h" pour un entier court
- **type**: l'une des lettres de descripteur de type: d, i, o, u, x, X, c, s, f, e, E, g, G

Notes:

Le descripteur:

```
%10.5s
```

Pour une chaîne de caractères signifie que l'on écrit que les 5 premiers caractères de la chaîne (donc tronquée si elle est plus longue) sur 10 positions donc complétée par des espaces devant!

Et surtout rappelez-vous que malheureusement si vous donnez des éléments incohérents entre format et valeur, il peut se passer n'importe quoi!

[1] Les [ ] désignent des éléments optionels

Exemple

Voici un exemple illustrant certains éléments présentés ci-dessus:

```
/*
   Programme exemple d'affichages
      PRINTF
*/
#include <cstdlib>
#include <cstdio>
using namespace std;
int main ( )
{
  int               i1   = 62, i2 = 'A';
  unsigned int      u    = 27;
  unsigned long int ul   = 27;
  char              sp   = '\12';
  int               hexa = 0xa;
  int               octa = 0123;
  float             f    = 12.5;

  printf ( " Quelques exemples d'affichage\n\n" );
  printf ( " Resultat\n%d\n%c\n%u\n%o\n%d\n%c\n",
                         i1, i1, i1, i1, i2, i2 );
  printf ( "debut\t\\et la suite\b etrange\r***\n" );
  printf ( "Encore %c un test\n",sp );
  printf ( "hexa: %6o, %6x, %6d\n", hexa, hexa, hexa );
  printf ( "octal: %6o, %6x, %6d\n", octa, octa, octa );
  printf ( "Non signe: %u, %lu\n", u, ul );
  printf ( "Float: %f, %14f, %8.2f\n", f, f, f );
  printf ( "Scientifique: %e, %14E, %8.2e\n", f, f, f );
  printf ( "Format g: %g, %14G, %8.2G\n", f, f, f );

  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Ce programme donne le résultat suivant:

```
Quelques exemples d'affichage

 Resultat
62
>
62
76
65
A
***ut   \et la suit etrange
Encore
 un test
hexa:     12,      a,     10
octal:    123,     53,     83
Non signe: 27, 27
Float: 12.500000,      12.500000,    12.50
Scientifique: 1.250000e+001,  1.250000E+001, 1.25e+001
Format g: 12.5,           12.5,       13

Fin du programme...Appuyez sur une touche pour continuer...
```

Notez que dans ce programme nous n'avons pas d'inclusion de *iostream* puisque tous nos affichages se réalisent par l'intermédiaire de la bibliothèque *cstdio*!

Voilà pour les écritures. Passons maintenant à la lecture au clavier, c'est-à-dire la fonction *scanf*.

La valeur d'un pointeur peut être affichée par *printf* à l'aide d'un format *%p*; l'apparence de la sortie dépend de l'environnement; cela n'a que peu d'importance, car quel est l'intérêt d'afficher la valeur d'un pointeur? De plus, avec ce format *%p*, on peut aussi lire la valeur d'un pointeur par *scanf*. Mais si cela n'a que peu de sens d'afficher un pointeur, que dire de sa lecture?!

scanf

La fonction *scanf* est en quelque sorte l'équivalent de *printf* pour les lectures. Sa structure générale correspond à:

```
scanf ( "Chaîne de contrôle", liste_de_lvalues );
```

La chaîne de contrôle contient des descripteurs de format, avec pour l'essentiel les mêmes significations que pour *printf*. Les valeurs de champs sont rarement utilisées dans le cas d'une lecture interactive. Si la chaîne de contrôle comporte des caractères non blancs (autres que: espace, tabulation (horizontale ou verticale), fin de ligne et fin de page), ces caractères sont explicitement attendus en lecture. Si elle comporte des caractères blancs, ceux-ci peuvent être remplacés en lecture par n'importe quel caractère ou suite de caractères blancs.

Notons l'exception des formats *%d* (pour décimal) et *%i* (pour entier). En écriture leur utilisation est absolument interchangeable. Ce n'est pas le cas en lecture où *%d* ne permet de lire que des valeurs fournies effectivement en décimal, alors que *%i* permet de lire des valeurs en octal, décimal ou hexadécimal.

Si après % on trouve le caractère "*", cela signifie que la valeur suivante en entrée (en fonction du format spécifié) doit être ignorée (sautée); donc attention sa signification s'avère différente de celle donnée pour le *printf*.

N'oubliez pas de mettre le & devant les identificateurs de la liste (sauf si c'est déjà une adresse [pointeur/tableau]). Un tel oubli représente une source courante d'erreur!

La fonction *scanf* livre un résultat entier qui représente le nombre d'éléments effectivement lus. Nous savons en principe combien nous devons en lire (le nombre de lvalues de la liste!); si le résultat obtenu ne correspond pas, nous pouvons assurer que nous avons rencontré un problème! Nous disposons là d'un mécanisme, encore rudimentaire il est vrai, de détection d'erreurs.

Attention: *scanf* avec le format %c! Si vous lui donnez une valeur de champ, par exemple:

```
scanf ( "%5c", &c );
```

l'opération va lire 5 caractères et les réduire à partir de l'adresse de *c* dans des octets consécutifs de la mémoire, donc si *c* était réellement de type **char**, vous allez détruire vraisemblablement d'autres objets!

Relevons que le type **double** étant plus utilisé que **float** (la bibliothèque mathématique de base étant par défaut en **double**), le format pour afficher une valeur de ce type est *%lf* avec, éventuellement, des constantes complémentaires pour les tailles!

Pour la lecture d'une chaîne avec le format %s les espaces (tabulations comprises) sont sautés, comme pour la lecture d'une valeur numérique; un tel comportement n'est pas forcément "avantageux" pour l'utilisateur d'autant plus que la lecture s'arrête aussi sur le premier caractère blanc rencontré!

L'instruction:

```
scanf ( "%[0123456789]", chaine );
```

lit des caractères sur l'entrée tant que ceux-ci appartiennent à l'ensemble des chiffres (dans cet exemple) et les réduit dans la chaîne. L'ensemble des caractères à traiter, celui mis entre [ ] peut être quelconque. Si vous désirez savoir combien de caractères ont effectivement été lus, vous pouvez utilisez le format %n associé à une variable entière de la liste de lvalue, variable dans laquelle on retrouvera cette information. Cette fonctionnalité est toujours à disposition dans n'importe quel *scanf* pour savoir combien de caractères ont déjà été traités depuis le début de cet appel à *scanf*. Attention, souvent il reste dans le tampon d'entrée la marque de fin de ligne (\n) de la lecture qui précède; il compte alors aussi dans le nombre de caractères lus!

Vous pouvez également limiter le champ de lecture, par exemple pour ne traiter au maximum que les 5 prochains caractères de l'entrée:

```
scanf ( "%5[0123456789]", chaine );
```

et comme nous l'avions déjà vu, si ces caractères doivent simplement être ignorés (sautés!), on pourra utiliser:

```
scanf ( "%*[0123456789]" );
```

Finalement, la condition peut être inversée, c'est-à-dire qu'on lit/saute tous les caractères ne faisant pas partie de l'ensemble, éventuellement jusqu'à une limite fixée:

```
scanf ( "%*[^0123456789]" );
```

On peut déduire des remarques ci-dessus que le format:

```
"%[^\n]"
```

permet de lire une chaîne de caractères comportant des espaces ou des tabulations. Seule une fin de ligne provoquant l'arrêt de la lecture. Par contre, si dès le départ nous sommes sur une fin de ligne, nous obtenons une chaîne vide. L'adjonction d'un espace devant le % résoudra ce problème (si ceci correspond à l'effet recherché!). De plus pour éviter les débordements, nous pouvons limiter le nombre de caractères lu:

```
"%10[^\n]"
```

Un exemple de programme pour illustrer quelques-unes de ces possibilités:

```
/*
   Programme exemple de lectures
      SCANF
*/
#include <cstdlib>
#include <cstdio>
using namespace std;
int main ( )
{
  int               n, i1, i2;
  char              chaine [20], c;

  printf ( "Quelques exemples de lectures\n\n" );
  printf ( "Donne ton nom: " );
  scanf ( "%s", chaine );
  printf ( "Bonjour %s, on continue...\n", chaine );
  printf ( "%s donne 2 entiers: ", chaine );
  scanf ( " " );  // Vide la fin de ligne!!!
  scanf ( "%i%n%i", &i1, &n, &i2 );
  printf ( "Merci pour ce %i et ce %i\n", i1, i2 );
  printf ( "Le premier nombre comportait %i chiffre(s).\n", n );
  printf ( "%s donne 1 entier de plus de 2 chiffres: ", chaine );
  scanf ( "%2i%i", &i1, &i2 );
  printf ( "Merci pour ce %i et ce %i et oui c'est ainsi!\n", i1, i2 );
  printf ( "%s donne 1 caractere: ", chaine );
  scanf ( " %c", &c );
  printf ( "Merci pour ce %c\n", c );
  printf ( "Donne une chaine commencant par des chiffres "
              "suivie de lettres:\n" );
  n = 0; scanf ( " " );
  scanf ( " %[0123456789]%n", chaine, &n );
  printf ( "Cette chaine commence par %i chiffre%s\n", n-1,
                n > 2 ? "s":"" );
  scanf ( "%*s" );  // Vide le tampon
  printf ( "Donne une autre chaine commencant par des chiffres "
              "suivis de lettres:\n" );
  scanf ( " %*[0123456789]" );
  scanf ( "%s", chaine );
  printf ( "Apres les chiffres vient: %s\n", chaine );


  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Un exemple d'exécution:

```
Quelques exemples de lectures

Donne ton nom: Toto
Bonjour Toto, on continue...
Toto donne 2 entiers:  12 345
Merci pour ce 12 et ce 345
Le premier nombre comportait 2 chiffre(s).
Toto donne 1 entier de plus de 2 chiffres: 12345
Merci pour ce 12 et ce 345 et oui c'est ainsi!
Toto donne 1 caractere: *
Merci pour ce *
Donne une chaine commencant par des chiffres suivie de lettres:
0452bof
Cette chaine commence par 4 chiffres
Donne une autre chaine commencant par des chiffres suivis de lettres:
33 Au revoir
Apres les chiffres vient: Au

Fin du programme...Appuyez sur une touche pour continuer...
```

Quelques remarques complémentaires au sujet de ce programme:

- Relevez tout particulièrement les espaces en début de chaîne de contrôle dans les *scanf* servant à lire un caractère ou une chaîne. Ils ne sont pas là pour l'esthétique, mais permettent de passer sur la marque de fin de ligne que nous obtiendrions sinon comme caractère ou qui provoquerait l'obtention d'une chaîne vide.
- Les lectures pour lesquelles nous précisons un champ, comme dans:

```
scanf ( "%2i%i", &i1, &i2 );
```

- seul le nombre de colonnes demandé est utilisé (dans notre exemple 2) les caractères suivants seront traités lors de la prochaine lecture. Avec notre instruction ci-dessus, si l'utilisateur donne: 1234, *i1* prendra la valeur 12 et *i2* la valeur 34. Cette possibilité permet entre autre d'éviter un débordement lors de la lecture d'une chaîne:

```
scanf ( "%5s", ch );
```

  toutefois n'oubliez pas de prévoir l'octet nul de terminaison de chaine!

- Notez finalement que dans le dernier *scanf*:

```
scanf ( " %*[0123456789]%s", chaine );
```

  la variable *chaine* ne reçoit pas les premiers caractères de la ligne, mais ceux qui viennent après les chiffres.

Fonctions complémentaires

Comme nous l'avons signalé, certains comportements ne se révèlent pas toujours idéals par rapport au traitement que l'on désire réaliser, par exemple l'arrêt de la lecture d'une chaîne sur un caractère blanc. Cette constatation nous amène à compléter très brièvement par quelques indications sur d'autres fonctions permettant aussi de réaliser des entrées/sorties.

```
int getchar ( void );
```

Lecture du prochain caractère sur le périphérique d'entrée (*stdin*[1]). Notez bien qu'elle livre comme résultat un entier, ceci dans (l'unique) but de pouvoir récupérer une valeur *EOF* (définie dans c*stdio*) en cas de tentative de lecture alors que l'on a atteint une fin de fichier.

```
char *gets ( char *string );
```

Lit sur le périphérique d'entrée jusqu'à la rencontre d'une fin de ligne qui est remplacée dans la chaîne lue par un caractère nul. La fonction retourne un pointeur sur la chaîne ou *NULL* en cas d'erreur. Comtrairement au printf avec %s, tous les caractères sont significatifs ("espaces blancs" compris), la lecture se terminant uniquement sur une fin de ligne. Conséquence: si au moment de la lecture on se trouve déjà sur une fin de ligne, nous obtenons une chaîne vide!

```
int putchar ( int c );
```

Ecrit le caractère *c* sur le périphérique de sortie (*stdout*[2]).

```
int puts ( const char *string );
```

Ecrit sur le périphérique de sortie la chaîne *string*, le caractère nul de fin de chaîne est remplacé par une fin de ligne. Retourne le caractère de fin de ligne ou *EOF* en cas d'erreur.

[1] La notion sera traitée au chapitre suivant.

[2] La notion sera traitée au chapitre suivant.

Les fichiers à la C

Introduction

Nous allons maintenant aborder la thématique des fichiers. Ayant terminé le chapitre qui précède sur des éléments hérités de C, nous allons continuer dans cette direction après quelques notions générales liées aux fichiers. Nous n'aborderons les aspects C++ des fichiers que dans le chapitre suivant. Les exemples de programmes que nous donnerons ici seront donc écrits en "C standard". Les bibliothèques restant compatibles, ils sont compilables en C++. Toutefois rappelons les modifications minimales qu'il serait raisonnable d'apporter pour en faire des programmes C++:

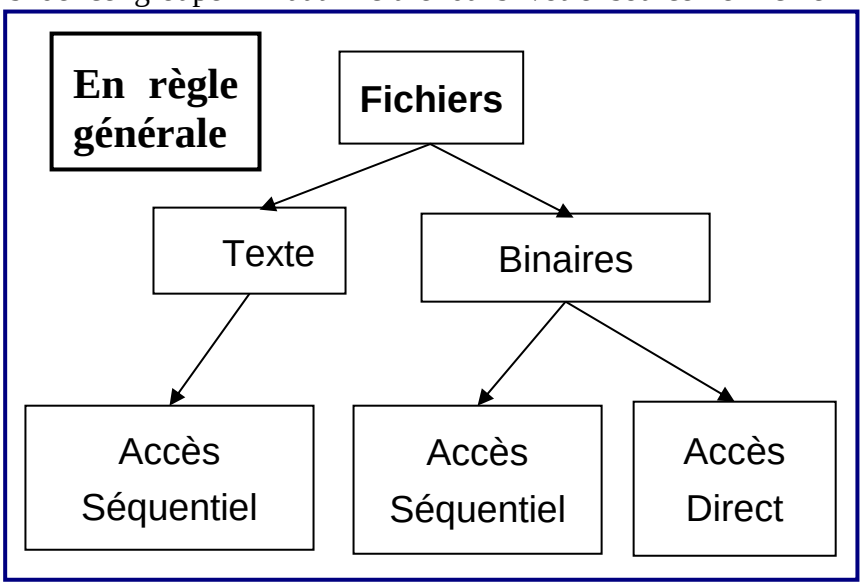
1. Remplacer les "#include xxx.h" par "#include cxxx".
2. Remplacer ***int*** *main (**void**)* par ***int*** *main ( ).*
3. Plus éventuellement remplacer les *fprintf/fscanf* par des flux.

Généralités sur les fichiers

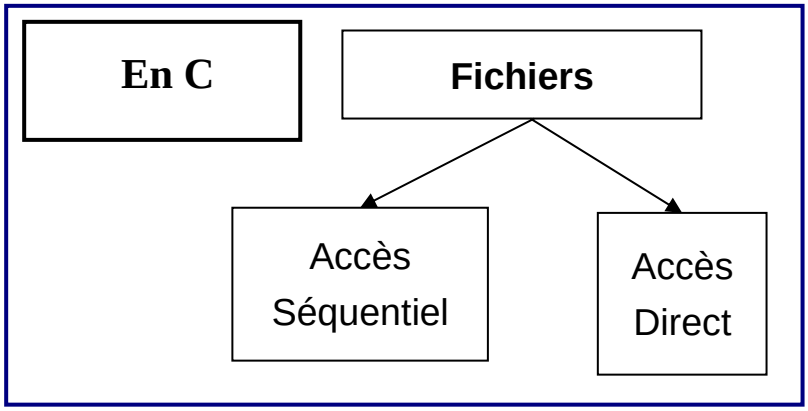
Tous les langages de programmation offrent des possibilités de traitement de fichiers. C, qui possède une riche bibliothèque dans ce domaine, ne fait pas exception.

Pour utiliser les fonctions de ce groupe il faut inclure dans votre source le fichier *<stdio.h>*.

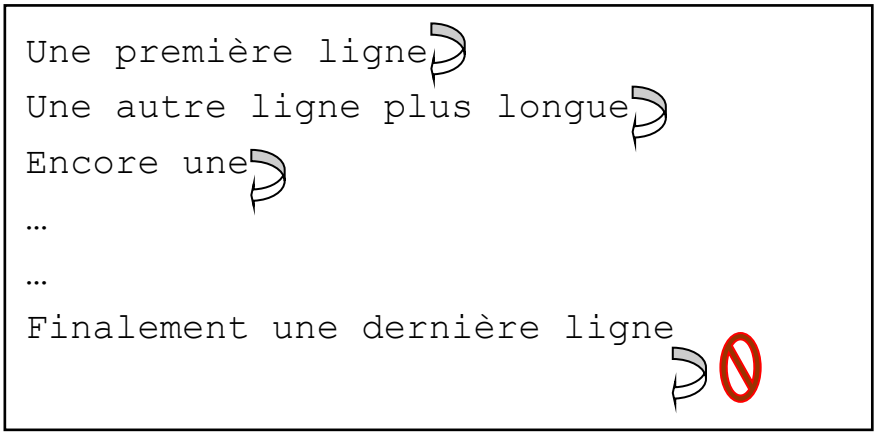
Dans la majorité des langages on distingue clairement entre les fichiers de texte et les fichiers binaires.



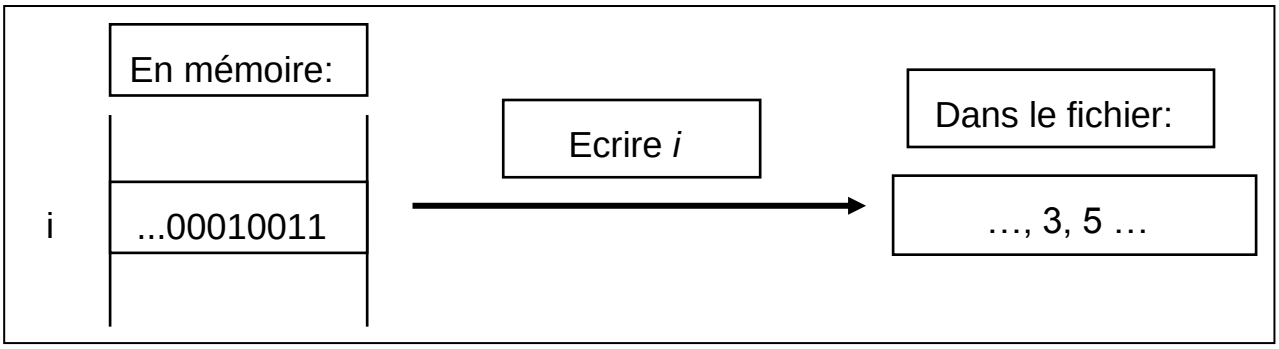
En C/C++ cette distinction est beaucoup plus floue, car pour ce langage tous les fichiers ne sont qu'une suite d'octets que l'on peut traiter sous une forme ou une autre en fonction des sous-programmes utilisés et bien entendu avec les risques que cela comporte.



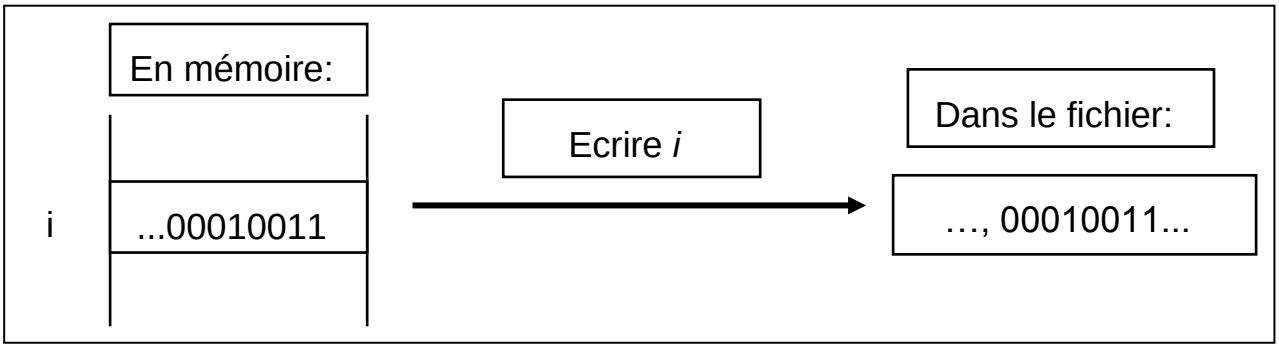
Rappelons qu'un fichier texte est constitué d'une suite de caractères, en principe organisés en lignes, donc avec une indication de fin de ligne ('\n') pour chacune d'elles, mais la nature physique de cette fin de ligne dépend du système!



L'écriture dans un fichier texte implique la conversion de la valeur à écrire en une suite de caractères, ce qui nécessite des instructions pour réaliser cette conversion et donc du temps à l'exécution pour effectuer ces instructions.



L'écriture dans un fichier binaire implique une simple recopie de l'image mémoire (binaire) de la valeur, et par conséquence de meilleures performances que pour un fichier de texte.

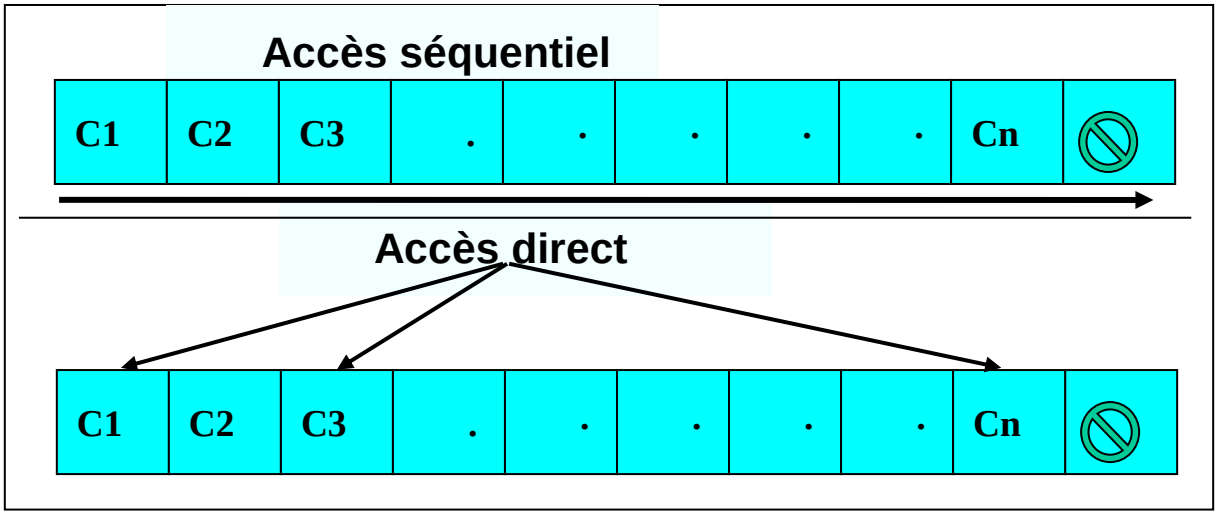


Au niveau de la lecture dans les fichiers, on peut faire les mêmes remarques, simplement la conversion est inverse; nous devons à partir d'une suite de caractères reconstruire la valeur binaire du type de destination.

N'en déduisez pas qu'il faut systématiquement utiliser des fichiers binaires, les objectifs des uns et des autres ne sont pas identiques! Vous ne pourrez pas envoyer sur une imprimante ou afficher à l'écran un fichier binaire; vous ne pourrez pas non plus le reprendre dans un éditeur de texte pour un traitement ultérieur. De plus un fichier texte facilite généralement le transfert des données d'un environnement à un autre, mais ne résout pas tous les problèmes, la codification des caractères pouvant changer de l'un à l'autre!

Dans la majorité des langages on distingue aussi clairement entre l'accès séquentiel et l'accès direct aux fichiers: l'accès séquentiel étant possible aussi bien sur les fichiers de texte que sur les fichiers binaires, alors que l'accès direct est en principe réservé aux fichiers binaires.

En C/C++ cette distinction est bien plus floue, puisque tous les fichiers correspondent à une suite d'octets, on peut théoriquement faire de l'accès direct sur les 2, avec un comportement qui peut partiellement varier en fonction du système sur lequel on travaille. Nous reviendrons sur ce point par la suite.



Généralement, dans tout traitement de fichiers on distingue 3 phases:

- La préparation ou ouverture du fichier qui permet de faire le lien entre le programme et le fichier externe physique.
- Le traitement proprement dit, en d'autres termes les lectures et/ou écritures. Cette phase se déroule généralement dans une boucle.
- La terminaison ou fermeture du fichier qui va rompre le lien entre le programme et le fichier externe, ce qui aura pour conséquence de vider le tampon associé au fichier et de libérer la place qu'il utilise.

❑ *Préparation des fichiers - traitements de base*

Nous allons montrer maintenant quelques possibilités générales de traitement des fichiers.

Nous nous baserons sur un premier exemple, qui consiste simplement à recopier un fichier source dans un fichier destination. Nous le ferons sur un fichier texte à titre d'exemple, mais nous réutiliserons cette notion de copie de fichier pour montrer de nombreuses variantes offertes par la bibliothèque du langage.

Dans ce programme, pour écrire dans le fichier, nous utilisons la fonction *fprintf* qui se comporte comme *printf*, mais appliquée à un fichier. Nous disposons de la même syntaxe que pour *printf* et avec les mêmes conventions pour le formatage des données. Il suffit de lui ajouter un premier paramètre qui est un pointeur sur un fichier (stream/flux).

En entrée, pour la lecture du fichier source, la fonction *fscanf* se comporte elle comme *scanf*, avec les mêmes remarques que ci-dessus.

En pratique:

Tout d'abord pour manipuler des fichiers nous devons déclarer des variables du type pointeur sur *FILE*. *FILE*, dont nous n'avons pas à connaître la structure (elle change d'un environnement à l'autre!), est défini dans *stdio.h*:

```
FILE *fichierSource, *fichierSortie;
```

Ensuite, avant de pouvoir traiter le fichier, il faut le préparer par un appel à la fonction *fopen*:

```
FILE *fopen ( const char *nom, const char *mode );
```

Elle comporte 2 paramètres:

- Le premier représente l'adresse d'une chaîne de caractères (un pointeur) contenant le nom externe du fichier à traiter; ce nom peut comporter un chemin d'accès complet dont la forme dépend du système d'exploitation.
- Le deuxième, contient aussi l'adresse d'une chaîne qui indique le mode de traitement, par exemple pour un fichier ouvert en lecture ("r"), il doit exister; pour un autre ouvert en écriture ("w"), il sera alors créé ou écrasera une ancienne version du fichier s'il existait déjà.

De manière générale, ce paramètre *mode* peut prendre les valeurs suivantes pour les fichiers de texte:

"r"	ouverture en lecture; le fichier doit exister au préalable
"w"	ouverture en écriture; le fichier est créé ou remplace une ancienne version
"a"	ouverture en écriture pour ajouter des éléments en fin de fichier (append). Si le fichier n'existe pas il est créé
"r+"	ouverture en lecture et écriture, le fichier doit exister au préalable
"w+"	ouverture en lecture et écriture, le fichier est créé ou remplace une ancienne version
"a+"	ouverture en lecture et écriture pour ajouter des éléments en fin de fichier (append). Si le fichier n'existe pas il est créé

On ajoute simplement un b pour indiquer qu'il s'agit d'un fichier binaire, soit: *"rb" , "wb", "ab", "rb+" (ou "r+b" ce qui revient au même), "wb+" (ou "w+b") et "ab+" (ou "a+b").*

La fonction *fopen* livre en retour un pointeur (sur un élément de type *FILE*) que nous affectons à nos variables de fichiers pour pouvoir les transmettre ensuite aux autres sous-programmes lors du traitement proprement dit. Si l'opération ne se passe pas correctement (par exemple le fichier ouvert en lecture n'existe pas!) la fonction livre comme résultat un pointeur *NULL*. Ceci explique un test que nous réaliserons souvent dans nos programmes:

```
if (((fichier_source = fopen (nom_du_source, "r"))!= NULL) &&
     ((fichier_sortie = fopen (nom_de_sortie, "w"))!= NULL))
```

car nous ne pouvons pas faire le traitement si les opérations de préparation ne se déroulent pas correctement.

Ce traitement se réalise, comme nous l'avons dit en introduction, par les fonctions:

```
int fscanf  ( FILE *fichier, const char *format, ... );
int fprintf ( FILE *fichier, const char *format, ... );
```

Toutes 2 se comportent respectivement comme *scanf* et *printf*, mais appliquées aux fichiers qui font l'objet de leur premier paramètre.

Après le traitement proprement dit, on ferme les fichiers par des appels à la fonction:

```
int fclose ( FILE *fichier );
```

qui a un paramètre, le pointeur sur le fichier à fermer. Cette opération a pour effet de vider le contenu des tampons et de libérer la place utilisée par ceux-ci.

En cours de traitement nous pouvons utiliser la fonction booléenne:

```
int feof ( FILE *fichier );
```

qui livre vrai si nous avons déjà tenté une lecture alors que nous nous trouvions sur la fin du fichier. Malheureusement il faut avoir tenté cette lecture pour que la fonction livre vrai, ce qui explique la présence d'une boucle infinie que nous utiliserons souvent dans ce cas:

```
while ( 1 )  // Par exemple
```

et qui nous oblige à ajouter un test dans la boucle pour pouvoir en sortir brutalement:

```
if ( feof ( fichier_source ) ) break;
```

Voici un petit programme de démonstration:

```c
/*
  Recopie d'un fichier (caractere par caractere) dans un autre fichier
  COPIEFICHIER1.c
*/
#include <stdio.h>
#include <stdlib.h>
/* Longueur maximale d'un nom de fichier */
#define MAX_NOM 30
int main ( void )
{
  char caractere;
  char nomDuSource [ MAX_NOM ], nomDeSortie [ MAX_NOM ];

  /* Pointeur sur le fichier d'entree, respectivement sortie */
  FILE *fichierSource, *fichierSortie;

  /* Demander le nom des fichiers */
  printf ( "Donnez le nom du fichier source: " );
  scanf  ( "%s" , nomDuSource );
  printf ( "Donnez le nom du fichier de sortie: " );
  scanf  ( "%s" , nomDeSortie );

  /* Ouverture des fichiers avec controle!! */
  if ( ( ( fichierSource = fopen ( nomDuSource, "r" ) )!= NULL ) &&
       ( ( fichierSortie = fopen ( nomDeSortie, "w" ) )!= NULL ) )
  {
    /* Traiter tous les caracteres du fichier */
    while ( 1 )
    {  fscanf  ( fichierSource, "%c", &caractere );
       if ( feof ( fichierSource ) ) break;
       fprintf ( fichierSortie, "%c", caractere  );
    }
    fclose ( fichierSource );
    fclose ( fichierSortie );
  }
  else
    printf ( "Erreur\n" );

  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Note:

Dans cet exemple nous avons traité le fichier caractère par caractère. On pourrait être tenté de le faire en utilisant des chaînes et le format %s. Malheureusement la lecture d'une chaîne sous cette forme s'arrête au premier caractère blanc (espace, tabulation, fin de ligne, …). Ceci rendrait le traitement complexe et peu efficace!

La forme utilisée ci-dessus nous semble peu élégante et nous amène à proposer une deuxième version, d'ailleurs guère plus élégante que la première:

```
/*
  Recopie d'un fichier (caractere par caractere) dans un autre fichier
  COPIEFICHIER2.c
*/
#include <stdio.h>
#include <stdlib.h>
/* Longueur maximale d'un nom de fichier */
#define MAX_NOM 30
int main ( void )
{
  char caractere;
  char nomDuSource [ MAX_NOM ], nomDeSortie [ MAX_NOM ];
  /* Pointeur sur le fichier d'entree, respectivement sortie */
  FILE *fichierSource, *fichierSortie;

  /* Demander le nom des fichiers */
  printf ( "Donnez le nom du fichier source: " );
  scanf  ( "%s" , nomDuSource );
  printf ( "Donnez le nom du fichier de sortie: " );
  scanf  ( "%s" , nomDeSortie );

  /* Ouverture des fichiers avec controle!! */
  if ( ( ( fichierSource = fopen ( nomDuSource, "r" ) )!= NULL ) &&
       ( ( fichierSortie = fopen ( nomDeSortie, "w" ) )!= NULL ) )
  { /* Traiter tous les caracteres du fichier */
    while ( fscanf  ( fichierSource, "%c", &caractere ) == 1 )
       fprintf ( fichierSortie, "%c", caractere  );
    fclose ( fichierSource );
    fclose ( fichierSortie );
  }
  else
    printf ( "Erreur\n" );

  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Nous nous basons simplement ici sur le fait que la fonction *fscanf* (comme *scanf*) livre un résultat entier représentant le nombre d'éléments effectivement lus, et comme nous savons que nous lisons un seul caractère, si ce résultat ne vaut pas 1, c'est que nous avons rencontré un problème ou une fin de fichier.

Tout ce que nous avons dit sur les fonctions autres que *fscanf* et *fprintf* s'applique aussi bien aux fichiers binaires qu'aux fichiers de texte. Et encore, cas limite, les fonctions fprintf et fscanf peuvent aussi s'utiliser sur des fichiers binaires.

❑ *Fonctions auxiliaires générales*

Signalons encore dans la catégorie des fonctions générales de traitement des fichiers:

```
int fflush ( FILE *stream );
```

Elle provoque l'écriture physique du tampon associé au fichier *stream*. Pour la lecture le résultat est en principe indéterminé, toutefois dans plusieurs environnements cela vide le tampon d'entrée, mécanisme que nous pouvons parfois utiliser à la fin des programmes pour provoquer une temporisation avant la fermeture de la fenêtre, ceci avec une séquence du genre[1]:

```
printf ( "Fin du programme, pressez <Enter> pour terminer\n" );
fflush ( stdin );  // [2]
return EXIT_SUCCESS;
```

Si le pointeur transmis à *fflush* est nul, tous les fichiers dans lesquels nous pouvons écrire sont vidés. La fonction livre *EOF* si l'opération a provoqué une erreur et 0 sinon.

```
int remove ( const char *name );
```

Permet d'effacer le fichier qui porte le nom *name*. Si le fichier est ouvert, le résultat dépend de l'implémentation. Livre une valeur différente de 0 en cas d'erreur.

```
int rename ( const char *old, const char *new );
```

Permet de changer le nom du fichier désigné par *old* par le nom désigné par *new*. Si le nouveau nom existe déjà, le comportement dépend de l'implémentation. Livre une valeur différente de 0 en cas d'erreur et dans ce cas le fichier garde son ancien nom.

```
FILE *tmpfile ( void );
```

Crée un fichier temporaire qui sera automatiquement détruit à la fin du programme. Le fichier est ouvert dans le mode *"wb+"*. Livre *NULL* en cas d'échec.

```
char *tmpnam ( char *name );
```

Génère un nom de fichier valide, différent à chaque appel. Si *name* n'est pas *NULL*, le nom est livré dans la chaîne qu'il désigne, sinon une chaîne, qui est automatiquement créée, est livrée comme résultat de la fonction.

---

[1] Mais cela ne fonctionne pas partout!!!

[2] *stdin* correspond au flux d'entrée par défaut, c'est-à-dire à notre clavier !

❑ *Les fichiers texte:*

Comme nous l'avions annoncé, nous allons reprendre notre programme de copie d'un fichier pour illustrer d'autres possibilités de traitement.

Pour notre troisième version, puisque nous ne faisons que lire et écrire un caractère à chaque fois, nous n'avons pas besoin de la complexité de *fscanf* et *fprintf*, complexité qui se ressent dans l'efficacité. Nous allons donc utiliser des fonctions plus simples faites pour lire et écrire un et un seul caractère, à savoir pour la lecture d'un caractère:

```
int fgetc ( FILE *fichier );
```

On lui transmet en paramètre un pointeur sur le fichier, et elle livre, notez le bien, un entier et non pas un caractère; la raison de cette bizarrerie s'explique par le simple fait que c'est la seule manière de pouvoir récupérer l'éventuelle indication de fin de fichier. En effet la fonction livre en retour le caractère lu ou la valeur *EOF* (indicateur de fin de fichier défini dans *stdio.h*).

Pour l'écriture du caractère nous utilisons:

```
int fputc ( int c, FILE *fichier );
```

Elle écrit le caractère transmis comme premier paramètre dans le fichier faisant l'objet du pointeur en deuxième paramètre. Cette fonction livre comme résultat le caractère écrit ou *EOF* s'il y a eu un problème à l'écriture.

Voici donc cette troisième version du programme:

```c
/*
  Recopie d'un fichier (caractere par caractere)dans un autre fichier
  COPIEFICHIER3.c
*/
#include <stdio.h>
#include <stdlib.h>
/* Longueur maximale d'un nom de fichier */
#define MAX_NOM 30
int main ( void )
{
  char caractere;
  char nomDuSource [ MAX_NOM ], nomDeSortie [ MAX_NOM ];

  /* Pointeur sur le fichier d'entree, respectivement sortie */
  FILE *fichierSource, *fichierSortie;

  /* Demander le nom des fichiers */
  printf ( "Donnez le nom du fichier source: " );
  scanf  ( "%s" , nomDuSource );
  printf ( "Donnez le nom du fichier de sortie: " );
  scanf  ( "%s" , nomDeSortie );

  /* Ouverture des fichiers avec controle!! */
  if ( ( ( fichierSource = fopen ( nomDuSource, "r" ) )!= NULL ) &&
        ( ( fichierSortie = fopen ( nomDeSortie, "w" ) )!= NULL ) )
  {
    /* Traiter tous les caracteres du fichier */
    while ( ( caractere = fgetc ( fichierSource ) ) != EOF )
       fputc ( caractere, fichierSortie );
    fclose ( fichierSource );
    fclose ( fichierSortie );
  }
  else
    printf ( "Erreur\n" );

  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Il existe une autre version de ces fonctions *fgetc* et *fputc*, à savoir respectivement *getc* et *putc* dont la syntaxe demeure la même, mais l'implémentation est réalisée sous forme de macros. Il faut juste être prudent et ne pas transmettre un paramètre dont l'évaluation provoquerait un effet de bord, le paramètre pouvant être évalué plusieurs fois. Cette version macro devrait être plus efficace en temps, mais le code global généré pour plusieurs appels plus important en taille.

Voici donc notre quatrième version, en fait la même que la précédente:

```
/*
  Recopie d'un fichier (caractere par caractere) dans un autre fichier
  COPIEFICHIER4.c
*/
#include <stdio.h>
#include <stdlib.h>
/* Longueur maximale d'un nom de fichier */
#define MAX_NOM 30
int main ( void )
{
  char caractere;
  char nomDuSource [ MAX_NOM ], nomDeSortie [ MAX_NOM ];

  /* Pointeur sur le fichier d'entree, respectivement sortie */
  FILE *fichierSource, *fichierSortie;

  /* Demander le nom des fichiers */
  printf ( "Donnez le nom du fichier source: " );
  scanf  ( "%s" , nomDuSource );
  printf ( "Donnez le nom du fichier de sortie: " );
  scanf  ( "%s" , nomDeSortie );

  /* Ouverture des fichiers avec controle!! */
  if ( ( ( fichierSource = fopen ( nomDuSource, "r" ) )!= NULL ) &&
        ( ( fichierSortie = fopen ( nomDeSortie, "w" ) )!= NULL ) )
  {
    /* Traiter tous les caracteres du fichier */
    while ( ( caractere = getc ( fichierSource ) ) != EOF )
       putc ( caractere, fichierSortie );
    fclose ( fichierSource );
    fclose ( fichierSortie );
  }
  else
    printf ( "Erreur\n" );

  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Nous avions dit que le format %s de la fonction *fscanf* s'adaptait mal à notre problème de par son arrêt sur le premier espace blanc rencontré. Par contre la fonction *fgets*, elle, permet de lire une chaîne en ne s'arrêtant que sur une fin de ligne.

Sa syntaxe:

```
char* fgets ( char *chaine, int longueur, FILE *fichier );
```

Ses paramètres:

- Le premier: un pointeur sur la zone mémoire où sera réduite la chaîne lue, l'éventuelle fin de ligne en faisant partie.
- Le deuxième permet de limiter le nombre de caractères lus et réduits dans la chaîne. En fait, c'est un de moins que cette valeur qui sera effectivement traité puisqu'une position est réservée pour introduire le caractère nul. Le code de fin de ligne (\n) est aussi introduit dans la chaîne s'il est rencontré.
- Le troisième: un pointeur sur le fichier dans lequel la lecture se fait.

La fonction livre en retour, si tout se passe bien, le pointeur sur la chaîne (le premier paramètre). Par contre s'il y a eu un problème ou que l'on rencontre la fin de fichier sans avoir lu des caractères, elle livre le pointeur *NULL*. Dans ce cas la chaîne n'est pas modifiée.

Notez que si la ligne est trop longue, la suite sera traitée au prochain cycle de lecture/écriture puisque l'on écrit strictement ce qui a été lu!

En écriture nous utiliserons la fonction *fputs*:

```
int fputs ( const char *chaine, FILE *fichier );
```

Son premier paramètre est la chaîne à écrire (qui comporte ou non une fin de ligne) et le deuxième le pointeur sur le fichier où l'on veut écrire.

Le résultat fourni par la fonction consiste en une valeur positive si tout s'est bien passé ou la valeur *EOF* sinon.

Voici donc la cinquième version de notre programme:

```
/*
  Recopie d'un fichier (ligne par ligne)
  dans un autre fichier
  COPIEFICHIER5.c
*/
#include <stdio.h>
#include <stdlib.h>
/* Longueur maximale d'un nom de fichier */
#define MAX_NOM 30
#define LONGUEUR_LIGNE 120
int main ( void )
{
  char ligne [ LONGUEUR_LIGNE ];
  char nomDuSource [ MAX_NOM ], nomDeSortie [ MAX_NOM ];

  /* Pointeur sur le fichier d'entree, respectivement sortie */
  FILE *fichierSource, *fichierSortie;

  /* Demander le nom des fichiers */
  printf ( "Donnez le nom du fichier source: " );
  scanf  ( "%s" , nomDuSource );
  printf ( "Donnez le nom du fichier de sortie: " );
  scanf  ( "%s" , nomDeSortie );

  /* Ouverture des fichiers avec controle!! */
  if ( ( ( fichierSource = fopen ( nomDuSource, "r" ) )!= NULL ) &&
        ( ( fichierSortie = fopen ( nomDeSortie, "w" ) )!= NULL ) )
  {
    /* Traiter toutes les lignes du fichier */
    while ( ( fgets ( ligne, LONGUEUR_LIGNE, fichierSource ) ) != NULL )
       fputs ( ligne, fichierSortie );
    fclose ( fichierSource );
    fclose ( fichierSortie );
  }
  else
    printf ( "Erreur\n" );

  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Les versions suivantes de ce programme vont mettre en évidence une autre possibilité offerte par *stdio.h*, à savoir le fait de rediriger un flux. Pour cette opération nous disposons de la fonction *freopen*:

```
FILE* freopen ( const char *nom, const char *mode, FILE *fichier );
```

Les paramètres:

- Le premier: un pointeur sur la chaîne de caractères qui contient le nom externe du fichier à utiliser.
- Le deuxième: un pointeur sur la chaîne de caractères qui représente le mode dans lequel on traite le fichier (avec les mêmes conventions que pour *fopen*).
- Le troisième: le pointeur sur le flux que l'on veut rediriger. Le fichier auquel il était associé avant est fermé et une nouvelle ouverture est réalisée avec le nouveau fichier externe et dans le mode désiré.

En retour la fonction livre le pointeur sur le *fichier* si tout se passe bien et *NULL* sinon.

Une utilisation classique de cette fonction consiste à rediriger les flux standards prédéfinis: *stdin, stdout* et *stderr* pour les associer à des fichiers:

- *stdin* correspond à l'entrée standard par défaut, celle utilisée par les fonctions de lecture qui n'ont pas de fichier en paramètre, telle *scanf.*
- *stdout* correspond à la sortie standard, celle utilisée par les fonctions d'écriture qui n'ont pas de fichier en paramètre, telle *printf.*
- *stderr* correspond à la sortie standard réservée pour l'écriture des messages d'erreurs.

Suite à une telle opération, les fonctions d'entrées/sorties sans paramètre peuvent s'utiliser sur des fichiers.

Notez qu'après avoir redirigé un flux standard, nous ne pouvons pas revenir à la situation initiale par un autre appel à *freopen*; par contre, nous pourrons à nouveau le rediriger sur un autre fichier par une telle opération.

La copie proprement dite peut aussi se faire caractère par caractère grâce aux fonctions *getchar* et *putchar*.

getchar correspond pour l'entrée standard à *fgetc* pour les fichiers et *putchar* pour la sortie standard à *fputc*.

```
int getchar ( );
```

La fonction n'a pas de paramètre et livre comme résultat un entier contenant le caractère lu, ceci à nouveau afin de pouvoir récupérer la valeur EOF en cas de tentative de lecture après la fin du fichier, ce qui nous permet de sortir de la boucle dans notre exemple.

```
int putchar ( int c );
```

La fonction *putchar* a un entier comme paramètre d'entrée, le caractère à écrire; elle livre comme résultat le caractère écrit, toujours sous forme d'entier, ou *EOF* en cas d'erreur d'écriture.

Voici encore une nouvelle version du programme de copie de fichiers:

```
/*
  Recopie d'un fichier (caractere par caractere)
  avec reouverture des fichiers standards d'entree et de sortie
  REOPEN1.c
*/
#include <stdio.h>
/* Longueur maximale d'un nom de fichier */
#define MAX_NOM 30
int main ( void )
{
  /* La lecture des caracteres se fait par une variable entière
      pour pouvoir recuperer l'indication de fin de fichier!!
  */
  int  caractereCourant;
  char nomDuSource [ MAX_NOM ], nomDeSortie [ MAX_NOM ];
  /* Demander le nom des fichiers */
  printf ( "Donnez le nom du fichier source: " );
  scanf  ( "%s" , nomDuSource );
  printf ( "Donnez le nom du fichier destination: " );
  scanf  ( "%s" , nomDeSortie );
  /* Reouverture en lecture/ecriture des fichiers standards d'entree */
  /* et sortie, avec controle d'operation reussie!! */
  if ( freopen ( nomDuSource, "r", stdin ) != NULL &&
       freopen ( nomDeSortie, "w", stdout ) != NULL )
  {
    /* Traiter tous les caracteres du fichier */
    while ( ( caractereCourant = getchar ( ) ) != EOF )
      putchar ( caractereCourant );
    fclose ( stdin );
    fclose ( stdout );
  }
  else
    printf ( "Erreur\n" );
  return EXIT_SUCCESS;
}
```

Remarques:

Dans cet exemple nous n'avons pas mis notre "system ("pause");" classique puisque l'entrée standard a été redirigée! Ceci n'a que peu d'importance ici puisque l'utilisateur n'a pas de résultats à lire à l'écran. Pour la même raison nous aurions d'ailleurs aussi pu ne pas le mettre dans les exemples de programmes qui précèdent dans ce chapitre!

Bien entendu nous pouvons utiliser ce principe de la redirection avec les fonctions *scanf* et *printf*; nous ne donnerons pas d'exemple, il semble trivial après ce que nous venons de dire.

Par contre, donnons encore un exemple de redirection avec cette fois les fonctions *gets* et *puts* pour respectivement lire et écrire une chaîne de caractères sur l'entrée et la sortie standards:

```
char *gets ( char *chaine );
```

Lit tous les caractères sur l'entrée standard (qui a pu être redirigée), jusqu'à une fin de ligne (qui contrairement à *fgets* ne fait pas partie de la chaîne lue!) ou la fin de fichier.

Son paramètre: le pointeur sur la zone mémoire où sera réduite la chaîne lue.

La fonction livre en retour, si tout s'est bien passé, le pointeur sur la chaîne (le paramètre). Par contre s'il y a eu un problème ou que l'on rencontre la fin de fichier sans avoir lu des caractères, elle livre le pointeur *NULL*.

Attention: contrairement à *fgets*, la fonction ne possède pas un paramètre pour limiter le nombre de caractères lus et si notre chaîne en mémoire a été prévue trop petite il peut alors se passer n'importe quoi!

```
int puts ( const char *chaine );
```

Ecrit tous les caractères de la chaîne, y compris les éventuelles fins de lignes qui si trouveraient.

Retourne une valeur positive si tout se passe bien et *EOF* sinon.

Attention: contrairement à *fputs*, la fonction ajoute en sortie une fin de ligne!

Voici une version adaptée du programme:

```
/*
  Recopie d'un fichier (ligne par ligne)
  avec reouverture des fichiers standards d'entree et sortie
  REOPEN2.c
*/
#include <stdio.h>
/* Longueur maximale d'un nom de fichier */
#define MAX_NOM 30
#define LONGUEUR_LIGNE 120

int main ( void )
{
  char  ligne [ LONGUEUR_LIGNE ];
  char nomDuSource [ MAX_NOM ], nomDeSortie [ MAX_NOM ];

  /* Demander le nom des fichiers */
  printf ( "Donnez le nom du fichier source: " );
  scanf  ( "%s" , nomDuSource );
  printf ( "Donnez le nom du fichier destination: " );
  scanf  ( "%s" , nomDeSortie );

  /* Reouverture des fichiers standards d'entree et sortie */
  /* avec controle d'operation reussie!! */
  if ( freopen ( nomDuSource, "r", stdin ) != NULL &&
       freopen ( nomDeSortie, "w", stdout ) != NULL )
  {
    /* Traiter tous les caracteres du fichier */
    while ( gets ( ligne ) != NULL )
      puts ( ligne );
    fclose ( stdin );
    fclose ( stdout );
  }
  else
    printf ( "Erreur\n" );
  return EXIT_SUCCESS;
}
```

Voilà, nous n'avons pas tout dit sur les fichiers de texte, mais vous connaissez l'essentiel!

❑ *Les fichiers binaires séquentiels:*

Le principe de traitement de fichiers binaires séquentiels s'avère extrêmement simple; d'abord rappelons les règles générales déjà énoncées.

En premier lieu, il faut préparer le fichier (l'ouvrir); c'est le rôle de la fonction *fopen* (éventuellement *freopen*), comme pour les fichiers de caractères, simplement la lettre b apparaît dans le mode ("rb", "rb+" (ou "r+b"), "wb", "wb+" (ou "w+b"), "ab", "ab+" (ou "a+b").

Après le traitement le fichier doit être fermé, la fonction *close* déjà étudiée joue ce rôle.

Le traitement proprement dit, les lectures et les écritures se réalisent respectivement par l'intermédiaire des fonctions *fread* et *fwrite*:

```
size_t fread(void *buf, size_t size, size_t nmemb, FILE *stream);
```

Lit *nmemb* éléments, chacun de taille *size* (en octets), dans le fichier *stream* et les réduit dans la zone désignée par le pointeur *buf*. La fonction renvoie le nombre d'éléments effectivement lus ou 0 en cas d'erreur.

En pratique on lit dans le fichier des objets d'un certain *type*, donc pour le paramètre *size* on utilise en général **sizeof** (*type*)!

Attention: si la valeur de retour de la fonction est différente de *nmemb*, des blocs ont pu être lus, mais pas tous ceux que l'on désirait, à nous de savoir ce que cela signifie dans la logique de notre problème! En pratique, sauf pour traiter ce cas particulier, il revient au même de lire *nmemb* blocs de taille *size* ou de lire un bloc de taille *nmemb***size*.

```
size_t fwrite(const void *buf,size_t size,size_t nmemb,FILE *stream);
```

Ecrit *nmemb* éléments, chacun de taille *size* (en octets), dans le fichier *stream*, les éléments viennent de la zone désignée par le pointeur *buf*. La fonction renvoie le nombre d'éléments effectivement écrits ou 0 en cas d'erreur.

Les remarques faites pour *fread* restent valables!

Voilà, vous savez tout ou presque, nous pouvons donner un exemple de programme et pourquoi pas toujours le même puisque nous avons dit qu'il n'y a pas de différence fondamentale entre un fichier de texte et un fichier binaire. La copie binaire d'un fichier texte reste strictement conforme à l'original. Elle pourra être reprise par exemple dans un traitement de texte.

Voici ce programme:

```
/*
  Recopie d'un fichier byte a byte (caractere par caractere)
    dans un autre fichier
  COPIEFICHIERBIN.c
*/
#include <stdio.h>
#include <stdlib.h>
/* Longueur maximale d'un nom de fichier */
#define MAX_NOM 30
int main ( void )
{
  char caractere;
  char nomDuSource [ MAX_NOM ], nomDeSortie [ MAX_NOM ];

  /* Pointeur sur le fichier d'entree, respectivement sortie */
  FILE *fichierSource, *fichierSortie;

  /* Demander le nom des fichiers */
  printf ( "Donnez le nom du fichier source: " );
  scanf  ( "%s" , nomDuSource );
  printf ( "Donnez le nom du fichier de sortie: " );
  scanf  ( "%s" , nomDeSortie );

  /* Ouverture des fichiers avec controle!! */
  if ( ( ( fichierSource = fopen ( nomDuSource, "rb" ) )!= NULL ) &&
        ( ( fichierSortie = fopen ( nomDeSortie, "wb" ) )!= NULL ) )
  {
    /* Traiter tous les caracteres (bytes) du fichier */
    while (  fread ( &caractere, sizeof ( char ), 1, fichierSource ) )
       fwrite ( &caractere, sizeof ( char ), 1,  fichierSortie );
    fclose ( fichierSource );
    fclose ( fichierSortie );
  }
  else
    printf ( "Erreur\n" );

  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Donnons un autre exemple, simple, mais réaliste et pratique, un petit programme permettant de faire un "dump" hexadécimal d'un fichier quelconque. Notez l'inclusion de *ctype.h* pour utiliser la fonction *isprint*, qui comme son nom l'indique, permet de déterminer si un caractère est imprimable ou non. Remarquez aussi dans cette première version l'utilisation d'un simple fgetc, bien que nous ayons indiqué à l'ouverture un traitement binaire:

```
/* Realise une "dump" hexadecimal d'un fichier */
/* Fichier: DUMP1.c                            */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
/* Longueur maximale d'un nom de fichier */
#define MAX_NOM 30
#define NOMBRE_PAR_LIGNE 8
int main ( void )
{
  int byte;  // !! logique non …!!
  char nomDuSource [ MAX_NOM ];
  long compteur = 1;

  /* Pointeur sur le fichier d'entree */
  FILE *fichierSource;

  /* Demander le nom du fichier */
  printf ( "Donnez le nom du fichier source: " );
  scanf  ( "%s" , nomDuSource );

  /* Ouverture du fichier avec controle!! */
  if ( ( fichierSource = fopen ( nomDuSource, "rb" ) )!= NULL )
  {
    /* Traiter tous les bytes du fichier */
    while ( ( byte = fgetc ( fichierSource ) ) != EOF )
    {   printf ( "%02hX ", byte );
        /* Est-ce un caractere imprimable? */
       isprint ( byte ) ? printf ( " %c ", byte ) : printf ( "   " );
        /* La ligne est-elle complete? */
       if ( !( compteur++ % NOMBRE_PAR_LIGNE ) ) printf ( "\n" );
    }
    fclose ( fichierSource );
  }
  else
    printf ( "Erreur\n" );

  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Si nous appliquons ce programme à un fichier dont voici le contenu:

```
C: c'est simple!
...
Menteur!!!
```

nous obtenons alors le résultat:

```
Donnez le nom du fichier source: e:\testdump.txt
43  C 3A  : 20    63  c 27  ' 65  e 73  s 74  t
20    73  s 69  i 6D  m 70  p 6C  l 65  e 21  !
0D    0A    2E  . 2E  . 2E  . 0D    0A    4D  M
65  e 6E  n 74  t 65  e 75  u 72  r 21  ! 21  !
21  !
Fin du programme...Appuyez sur une touche pour continuer..
```

Donnons maintenant une deuxième version de ce programme, dans laquelle nous allons lire plusieurs blocs par *fread* et nous baser sur le nombre de blocs effectivement lus pour l'affichage ainsi que pour déterminer la fin du traitement. Cette technique nous permet d'ailleurs de rendre l'affichage plus convivial:

```
/*
  Dump hexa d'un fichier quelconque
Fichier: DUMP2.c
*/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
/* Longueur maximale d'un nom de fichier */
#define MAX_NOM 30
#define NOMBRE_PAR_LIGNE 15
int main ( void )
{
  char bytes [ NOMBRE_PAR_LIGNE ];
  int nbValeursLues, i;
  char nomDuSource [ MAX_NOM ];

  /* Pointeur sur le fichier d'entree */
  FILE *fichierSource;

  /* Demander le nom du fichier */
  printf ( "Donnez le nom du fichier source: " );
  scanf  ( "%s" , nomDuSource );
```

```
  /* Ouverture du fichier avec controle!! */
  if ( ( fichierSource = fopen ( nomDuSource, "rb" ) )!= NULL )
  {
    /* Traiter tous les bytes du fichier */
    while ( 1 )
    {
       nbValeursLues = fread ( bytes, 1, NOMBRE_PAR_LIGNE, fichierSource );
       /* afficher les caracteres imprimables */
       for ( i = 0; i < nbValeursLues; i++ )
         isprint ( bytes[i] ) ? printf ( "%c", bytes[i] ) : printf ( " " );
       printf ( "%*c | ", NOMBRE_PAR_LIGNE - nbValeursLues + 1, ' ' );
       /* afficher les codes */
       for ( i = 0; i < nbValeursLues; i++ )
         printf ( "%02hX ", bytes [i] );
       printf ( "\n" );
       if ( nbValeursLues != NOMBRE_PAR_LIGNE ) break;
    }
    fclose ( fichierSource );
  }
  else
    printf ( "Erreur\n" );

  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Si nous appliquons cette version du *dump* au même fichier que pour la première version nous obtenons:

```
Donnez le nom du fichier source: e:\testdump.txt
C: c'est simple  | 43 3A 20 63 27 65 73 74 20 73 69 6D 70 6C 65
!  ...  Menteur  | 21 0D 0A 2E 2E 2E 0D 0A 4D 65 6E 74 65 75 72
!!!              | 21 21 21

Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *L'accès direct:*

Comme nous l'avons déjà signalé et contrairement à la grande majorité des autres langages de programmation, C permet un traitement en accès direct non seulement aux fichiers binaires, mais aussi (au moins théoriquement) aux fichiers de texte. Toutefois l'intérêt réel de cet accès direct réside bien dans l'utilisation de fichiers binaires et c'est sous cet aspect uniquement que nous allons aborder le problème.

Voici tout d'abord un peu brutalement un programme exemple, nous décrirons les fonctionnalités utilisées après:

```
/*
  Gestion d'un fichier en acces direct; cree un fichier d'entiers,
  le trie dans le fichier et l'affiche
  ================================================
  = C'est un exemple de ce que l'on peut faire,
  = pas de ce que l'on doit faire!!
  ================================================
  ACCESDIRECT.c
*/
#include <stdio.h>
#include <stdlib.h>

/* Affiche le contenu d'un fichier d'entiers */
void afficher ( FILE *fichier );
/* Trie un fichier d'entiers directement dans le fichier */
void trier ( FILE *fichier );

int main ( void )
{
  int valeur;
  /* Pointeur sur le fichier temporaire */
  FILE *fichier;

  /* Preparation d'un fichier temporaire avec controle!! */
  if ( ( fichier = tmpfile ( ) ) != NULL )
  { /* Traiter toutes les valeurs de l'utilisateur */
    while ( 1 )
    {  printf ( "Donnez une valeur entiere (0 pour terminer): " );
       scanf ("%d", &valeur );
       /* sortie si termine! */
       if ( valeur == 0 ) break;
       fwrite ( &valeur, sizeof ( int ), 1,  fichier );
    }
```

```
    printf   ( "\n\nFichier original:\n" );
    afficher ( fichier );
    trier    ( fichier );
    printf   ( "\n\nFichier trie:\n" );
    afficher ( fichier );
    fclose   ( fichier );
  }
  else
    printf ( "Erreur\n" );

  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}

/* Affiche le contenu d'un fichier d'entiers */
void afficher ( FILE *fichier )
{ int valeur;
  rewind ( fichier );
  printf ( "\n\ncontenu du fichier:\n\n" );

  /* Traiter tous les elements du fichier */
  while ( fread ( &valeur, sizeof ( int ), 1, fichier ) )
    printf ( "%d\n", valeur );
}

/* Trie un fichier d'entiers directement dans le fichier */
void trier ( FILE *fichier )
{ int valeurInf, valeurSup;
  int dernierEnregistrements;

  /* Fixer le nombre d'enregistrements */
  fseek ( fichier, 0, SEEK_END );
  dernierEnregistrements = ftell ( fichier ) / sizeof ( int );
  for ( int i = 0; i < dernierEnregistrements - 1; i++ )
    for ( int j = i; j < dernierEnregistrements; j++ )
    { fseek ( fichier, i * sizeof ( int ), SEEK_SET );
      fread ( &valeurInf, sizeof ( int ), 1, fichier );
      fseek ( fichier, j * sizeof ( int ), SEEK_SET );
      fread ( &valeurSup, sizeof ( int ), 1, fichier );
      /* Faut-il croiser les valeurs */
      if ( valeurInf > valeurSup )
      { fseek  ( fichier, i * sizeof ( int ), SEEK_SET );
        fwrite ( &valeurSup, sizeof ( int ), 1, fichier );
        fseek  ( fichier, j * sizeof ( int ), SEEK_SET );
        fwrite ( &valeurInf, sizeof ( int ), 1, fichier );
      }
    }
}
```

Tout d'abord dans ce programme nous utilisons la notion de fichier temporaire, qui rappelons-le, génère automatiquement un nom de fichier, le crée et le détruit lors de sa fermeture:

```
if ( ( fichier = tmpfile ( ) ) != NULL )
```

Ensuite, même si a priori nous avons prévu un traitement en accès direct, rien ne nous empêche de réaliser aussi sur ce fichier un traitement séquentiel. C'est ce que nous faisons en écriture lors de la création initiale du fichier avec les valeurs données par l'utilisateur:

```
fwrite ( &valeur, sizeof ( int ), 1,  fichier );
```

fonction que nous avons déjà introduite au paragraphe qui précède.

Il en va de même pour la lecture dans la fonction afficher:

```
while ( fread ( &valeur, sizeof ( int ), 1, fichier ) )
```

Notez que dans cette boucle nous nous basons sur le fait que la valeur retournée par *fread* doit être ici égale à 1 (donc vraie), sauf lorsque nous aurons atteint la fin de fichier donc que la lecture n'aura pas pu se faire! D'autres erreurs sont peu vraisemblables dans ce contexte!

Dans cette fonction d'affichage, nous avons aussi utilisé la fonction:

```
rewind ( fichier );
```

pour repositionner la fenêtre courante au début du fichier.

Le prototype de cette fonction:

```
void rewind ( FILE *stream );
```

Comme nous allons le voir après, il y a d'autres moyens pour arriver au même résultat!

Pour notre programme exemple l'aspect accès direct est entièrement traité dans la fonction *trier*, qui rappelons-le encore une fois n'est pas un exemple de ce qu'il faut faire mais de ce que l'on peut faire! Tout d'abord comme nous l'avons déjà dit les lectures/écritures se font par l'intermédiaire des fonctions maintenant connues: *fread*/*fwrite*.

C'est à nous de positionner correctement notre fenêtre courante sur le bon enregistrement, donc le premier octet de cet enregistrement puisqu'un fichier en réalité n'est constitué que d'une suite d'octets et non d'enregistrements.

Pour ceci nous disposons de la fonction:

```
int fseek ( FILE *stream, long offset, int method );
```

Elle permet de se positionner en un endroit (octet) précis dans le fichier *stream*.

- Si *method* vaut *SEEK_SET*, la position (*offset*) se donne par rapport au début du fichier.
- Si *method* vaut *SEEK_CUR* la position se donne par rapport à la position courante.
- Si *method* vaut *SEEK_END* la position se donne par rapport à la fin du fichier. L'offset peut donc être négatif. Ces 3 constantes (*SEEK_...*) nous viennent de *stdio.h*.

La fonction retourne 0 si tout va bien et un code d'erreur sinon.

Notez que:

- `fseek ( fichier, 0, SEEK_SET );`

revient au même que de faire:

```
rewind ( fichier );
```

- `fseek ( fichier, 0, SEEK_CUR );`

  revient à ne rien faire (ne pas changer la position de la fenêtre courante) mais sera tout de même utile pour passer du mode de lecture au mode d'écriture ou inversement. En effet, il est indispensable de repositionner explicitement la fenêtre courante lorsque l'on change de mode. Suivant l'effet désiré, ce repositionnement peut aussi s'obtenir par un appel à *rewind* ou a *fsetpos* (c.f. suite!).

- La fonction *fseek* peut théoriquement aussi s'utiliser pour les fichiers de texte, mais ce n'est pas très réaliste car il faut par exemple tenir compte des fins de lignes qui, suivant l'environnement, peuvent représenter un ou deux octets.

Comme nous ne disposons pas de la notion d'enregistrement, c'est à nous de calculer sa position (son premier octet dans le fichier). Ainsi pour se positionner sur le $i^{ème}$ enregistrement:

```
fseek ( fichier, ( i - 1 ) * sizeof ( type ), SEEK_SET );
```

Dans la boucle de notre fonction *trier* nous avons écrit:

```
fseek ( fichier, i * sizeof ( int ), SEEK_SET );
```

car notre boucle partait de 0!

Dans notre fonction nous utilisons un autre "gadget" pour calculer le nombre d'enregistrements du fichier:

```
fseek ( fichier, 0, SEEK_END );
dernierEnregistrements = ftell ( fichier ) / sizeof ( int );
```

Nous nous sommes d'abord positionnés à la fin du fichier par un appel à fseek, puis par un appel à ftell nous obtenons la position de cet octet et finalement en divisant par la taille d'un enregistrement nous obtenons le nombre d'enregistrements qui nous sert ensuite dans la boucle **for** pour en fixer la borne supérieure.

La forme générale de la fonction *ftell* est:

```
long ftell ( FILE *stream );
```

Elle livre comme résultat la position du pointeur courant dans le fichier *stream* ou -1 en cas d'erreur (ce dont nous n'avons pas tenu compte dans notre exemple!).

Signalons encore, dans l'optique de l'accès direct, 2 fonctions que nous n'avons pas utilisées dans l'exemple:

```
int fgetpos ( FILE *stream, fpos_t *pos );
```

Sauve dans *pos* la position courante dans le fichier *stream*. La norme ne précise pas la forme de cette information (le type *fpos_t*), mais elle garantit qu'en l'utilisant ensuite avec la fonction *fsetpos*, on retourne au bon endroit dans le fichier. Elle livre 0 en cas de succès et une autre valeur en cas d'échec.

Son utilité (bien que peu courante, et quand c'est le cas, essentiellement pour des fichiers de texte!) réside dans le fait de pouvoir "marquer" des endroits dans le fichier, positions auxquels on pourra revenir ensuite par un appel à:

```
int fsetpos ( FILE *stream, const fpos_t *pos );
```

Elle permet de retourner dans le fichier *stream* à l'endroit précis spécifié par *pos*, qui lui-même a été initialisé par un appel préalable à *fgetpos*. Elle livre 0 en cas de succès et une autre valeur en cas d'échec.

… et encore, en conclusion:

Voilà, vous ne savez pas tout sur les fichiers, mais l'essentiel; toutefois ajoutons encore brièvement quelques fonctions:

```
int ungetc ( int c, FILE *stream );
```

Remet dans le tampon du fichier *stream* le caractère *c*, qui peut être le dernier caractère effectivement lu ou un autre, ceci de telle manière que la prochaine lecture commencera par ce caractère. L'opération peut s'avérer utile pour décider à quel genre de lecture nous allons procéder en fonction de la nature de ce caractère.

La fonction ne s'applique pas uniquement aux fichiers de caractères puisqu'un octet est réintroduit dans le tampon du fichier (le fichier lui-même n'est pas modifié!). Elle livre comme résultat *EOF* s'il y a eu problème et sinon le caractère *c* lui-même.

Malheureusement la norme dit que le comportement dépend de l'implémentation si l'on réalise plusieurs *ungetc* de suite!

Pour les écritures formatées nous avons introduit les fonctions *printf* (pour *stdout*) et *fprintf* (pour un fichier); il existe encore:

```
int sprintf ( char *string, const char *format, ... );
```

Ecriture formatée dans la chaîne de caractères *string* (à la place d'un fichier); pour le reste tout se passe comme pour les 2 autres fonctions, si ce n'est qu'un caractère nul est ajouté en fin de chaîne; il n'est pas compté dans le nombre de caractères écrits renvoyé par la fonction.

De manière symétrique, il existe aussi pour la lecture:

```
int sscanf ( const char *string, const char *format, ... );
```

Lecture formatée dans la chaîne de caractères *string* (à la place d'un fichier); pour le reste tout se passe comme avec *scanf* ou *fscanf* sauf que le caractère nul de fin de chaîne se comporte comme une fin de fichier.

Flux - les fichiers à la C++

Introduction

Nous abordons maintenant les fichiers "à la mode C++" ce qui veut dire que tout ce que nous présentons dans ce chapitre ne s'applique pas aux programmes C. En fait, plutôt que de parler de fichiers, nous devrions utiliser le terme plus général de flux (ou flots). Il s'agit de canaux permettant respectivement d'envoyer (flux de sortie) ou de recevoir (flux d'entrée) des données.

Depuis nos premiers programmes, nous utilisons deux flux: *cin* et *cout*, pour le dialogue homme/machine. Tout ce que nous avons présenté sur eux reste valable pour tous les autres flux.

En fait *cin* et *cout* sont deux flux prédéfinis qui correspondent pour un programme C classique respectivement à *stdin* et *stdout*. Il existe encore 2 autres flux prédéfinis:

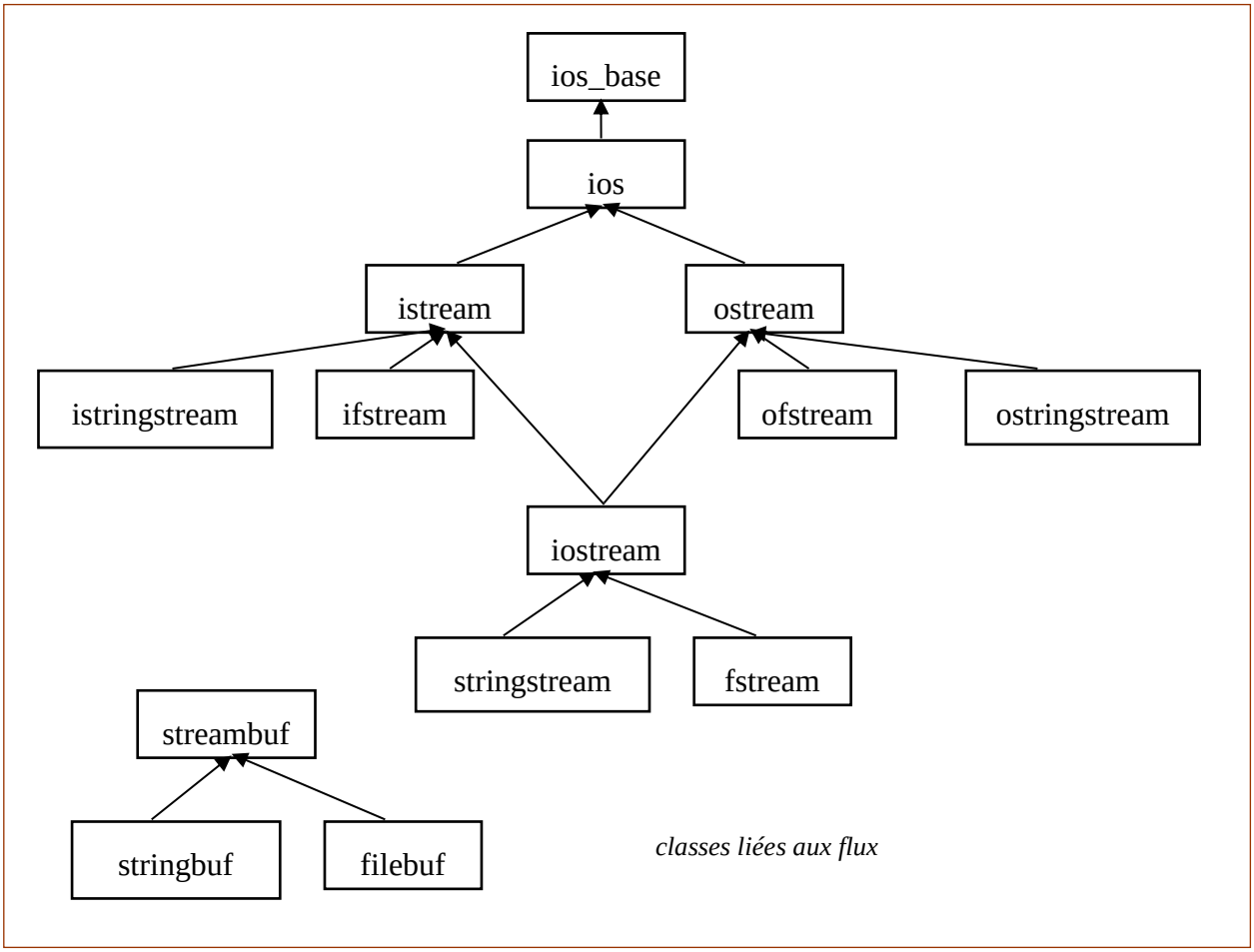
- *cerr*: pour la sortie des erreurs et qui correspond à *stderr* en C.
- *clog*: aussi pour la sortie des erreurs mais sous forme bufferisée.

La correspondance entre les formes C et C++ peut facilement se démontrer. Reprenez l'exemple *Reopen1* du chapitre qui précède; ajoutez y de manière classique l'utilisation de *iostream* et après la redirection du fichier de sortie, introduisez un ou des *cout << ...;* vous verrez que ces ordres d'écriture se retrouvent dans le fichier et non pas à l'écran.

De manière très générale, vous constaterez que même si C++ utilise apparemment une approche syntaxique passablement différente de C, vous retrouverez dans les notions que nous allons introduire un lien étroit avec celles vues pour les fichiers en C.

Comme bien souvent dans la bibliothèque C++, la manipulation des flux fait appel aux notions de programmation objet (classes/héritage/…). Sans aller dans les détails, nous n'aurons guère besoin de connaissances dans ce domaine plus approfondies que celles introduites pour les chaînes de caractères *string*.

Les classes liées au flux sont relativement nombreuses et les liens d'héritage passablement complexes. Nous vous donnons ci-dessous la structure générale de cette hiérarchie, mais ne vous inquiétez pas trop pour l'instant car vous constaterez qu'en pratique peu de choses sont réellement utilisée de manière directe.



Les flux de sortie tel *cout* correspondent à des objets de la classe *ostream*.

Les flux d'entrée tel *cin* correspondent à des objets de la classe *istream*.

La classe *iostream* héritant à la fois des classes *istream* et *ostream*, c'est elle que nous utilisons en pratique.

Nous allons maintenant introduire un certain nombre de notions qui nous permettrons d'écrire dès le début des programmes à peu près propres et robustes!

❑ *Méthodes complémentaires*

Jusqu'à présent nous avons essentiellement utilisé respectivement les opérateurs << (de *ostream*) pour nos écritures et >> (de *istream*) pour les lectures. Rappelons qu'ils sont surchargés pour tous les types de base et comme nous l'avons déjà fait, nous pouvons les surcharger pour nos propres types[1].

Ces opérateurs, bien que simples et pratiques, ne permettent pas de résoudre proprement tous les problèmes qui se posent. Un exemple: la lecture d'une chaîne de caractères par l'opérateur >> s'arrête sur le premier *espace blanc* rencontré (espace, tabulation horizontale ou verticale, fin de ligne, …). Ceci signifie que nous ne pouvons pas par cet opérateur lire toute une ligne, ni même obtenir le(s) caractère(s) de fin de lignes (\n) puisque la lecture d'un caractère ou d'une chaîne commence par sauter les espaces blancs avant d'obtenir et de fournir un caractère effectif!

Rappelons aussi que nous avons déjà vu au chapitre du dialogue homme/machine les possibilités de formatage à l'aide de manipulateurs appliqués aux flux et donc qui seront également applicables aux fichiers.

Nous allons maintenant décrire les principales méthodes complémentaires des classes *ostream* et *istream*, puis nous introduirons une rubrique spécifique au traitement des erreurs. Pour les exemples servant à illustrer nos propos nous continuerons pour l'instant à utiliser les flux prédéfinis *cin* et *cout*. Toutefois ce que nous présentons sera valable pour tous les flux. Nous apprendrons d'ici peu à créer nos propres flux et comment les connecter à des fichiers externes.

❑ Fonctionnalités de *ostream*

Tout d'abord signalons l'existence de la méthode *put*:

```
ostream & put ( char );
```

pas très utile mais qui existe pour des raisons historiques[2]. Elle s'applique à un flux de sortie sur lequel elle envoie le caractère transmis en paramètre.

[1] Par la suite nos classes

[2] Avant la version 3 de la norme, l'opérateur << ne permettait pas d'écrire un caractère, c'est son code (valeur entière) qui était envoyé sur le flux de sortie.

Exemple:

```
cout.put ( 'A' );
```

La méthode livrant en retour le flux sur lequel on l'applique, nous pouvons comme pour l'opérateur << enchaîner les opérations:

```
cout.put ( 'A' ).put ( 'B' ).put ( '\n' );
```

Mais aussi, bien que cela soit peu recommandable en raison du mélange des genres:

```
cout.put ( 'A' ).put ( 'B' ) << endl;
```

Attention toutefois au problème de priorité des opérateurs qui nous obligera à écrire:

```
(cout <<  "Salut").put ( '!' ) << endl;
```

La méthode *write*, quand à elle, paraît plus utile:

```
ostream & write ( const char *, int );
```

Elle s'applique à un flux et possède 2 paramètres. Le premier correspond à l'adresse d'une zone de mémoire à partir de laquelle seront pris et envoyés sur le flux un nombre d'octets fixé par son deuxième paramètre.

Exemple:

```
char salut [] = "Salut Toto";
…
cout.write ( salut, 5 ).put ( '!' ) << endl;
```

Ici seul "Salut!" est envoyé sur le flux de sortie!

De même que le *put*, elle livre en retour le flux auquel on l'applique. On peut donc l'enchaîner à elle-même, à d'autres méthodes et à l'opérateur << comme le montre notre exemple ci-dessus.

Le premier paramètre ne correspond pas forcément à l'adresse d'une chaîne de caractères, et ne sera pas interprétée en tant que telle (il s'agit d'une suite d'octets quelconque!). Le caractère nul ('\0') ne marque pas la fin du traitement et sera envoyé comme les autres octets sur le flux de sortie, comme le montre le mini programme ci-après:

```
/*
  OSTREAM
*/
#include <iostream>
#include <cstdlib>
using namespace std;
int main ( )
{
  int  val = 65*256+66; // "AB"
  char salut [] = "Salut\0 Toto";
  cout << salut << endl;
  cout.write ( salut, 11 ).put ( '!' ) << endl;
  cout.write ( (char *) &val , 2 ).put ( '!' ) << endl;
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution nous affiche les résultats suivants:

```
Salut
Salut  Toto!
BA!                                    1
Appuyez sur une touche pour continuer...
```

Note: nous avons un peu triché avec notre troisième affichage dans ce programme puisque pour l'instant notre flux *cout* correspond à un" fichier texte"! En pratique la méthode *write* sera essentiellement utilisée pour des sorties "brutes", sans aucune interprétation, donc pour des fichiers binaires.

Signalons encore la méthode:

```
ostream & flush ( );
```

Elle permet de vider le tampon associé au flux auquel on l'applique.

Exemple:

```
cout.flush ( );
```

Par la suite nous ajouterons encore 3 méthodes liées à l'accès direct dans un fichier.

[1] Dépend de l'environnement!!

❑ Fonctionnalités de *istream*

Tout d'abord la méthode *get* qui comporte de nombreuses surcharges.

Deux d'entre elles permettent de lire un seul caractère. La première peut paraître étrange, mais l'héritage de C pèse lourd:

```
int get ( );
```

Elle ne possède pas de paramètre et livre le caractère lu sous la forme d'un entier, ceci dans l'unique but de pouvoir récupérer dans l'entier le code de fin de fichier (EOF).

Exemple[1]:

```
int i;
while ( ( i = cin.get ( ) ) != EOF )
     cout.put ( i );
```

La deuxième forme permettant de lire un et un seul caractère correspond à:

```
istream & get ( char & );
```

Le caractère lu est livré dans le paramètre et la méthode en elle-même retourne le flux auquel on l'applique, ce qui permet comme nous l'avons vu pour le *put* d'enchaîner les opérations:

```
char c1, c2;
cin.get ( c1 ).get ( c2 );
```

L'exemple de la recopie de l'entrée standard sur la sortie standard donné pour la première forme du *get* s'écrirait maintenant[2]:

```
char c;
while ( cin.get ( c ))
     cout.put ( c );
```

Contrairement à l'opérateur >>, ces 2 méthodes fournissent tous les caractères, espaces blancs compris!

[1] Dans notre environnement, sur l'entrée standard (clavier) on simule la fin de fichier par la combinaison de touches: CTRL/z suivie de ENTER!

[2] La justification relative à la reconnaissance de fin de fichier dans ce cas sera fournie dans la partie traitant des erreurs.

Il existe aussi une surcharge du *get* permettant de lire une chaîne de caractères. Elle possède les mêmes paramètres (valeur par défaut comprise) que le *getline* que nous allons décrire ci-dessous. La seule différence réside dans son comportement par rapport à la fin de ligne[1] qu'elle ne consomme pas, qui reste donc dans le tampon d'entrée et qui provoquera la lecture d'une chaîne vide si nous opérons un deuxième *get* sous cette forme.

Venons en donc au *getline* dont voici le prototype:

```
istream & getline ( char * chaine, int longueurMax,
                    char delimiteur = '\n' );
```

Elle permet de lire une chaîne dont elle réduit les caractères à partir de l'adresse de *chaine*.

La lecture s'arrête:

- Soit parce que nous avons déjà lu *longueurMax*-1 caractères. Une position reste toujours conservée pour introduire la marque de fin de chaîne ('\0'). Nous disposons là d'un moyen de ne pas écraser des données en lisant une ligne plus longue que la zone réservée pour la contenir! Toutefois, dans ce cas le failbit sera activé (c.f suite: traitement des erreurs).
- Soit parce que le caractère *delimiteur* a été lu, mais il n'est pas conservé dans la chaîne saisie. Raisonnablement ce *delimiteur* correspond par défaut à la marque de fin de ligne.

Exemple:

```
#define L_MAX 4
…
char ch [ L_MAX ];
…
cin.getline ( ch, L_MAX, '\n' );
// idem a: cin.getline ( ch, L_MAX )
```

La méthode:

```
int gcount ( );
```

appliquée à un flux permet d'obtenir le nombre de caractères effectivement lus lors du dernier *get* ou *getline* sur ce flux.

Puisque *get* et *getline* livrent en retour le flux sur lequel nous les appliquons, nous pouvons enchaîner les opérations sous la forme:

```
cout << cin.getline ( ch, L_MAX, '\n' ).gcount ( );
```

[1] En réalité par rapport au caractère choisi comme délimiteur qui par défaut correspond à la marque de fin de ligne ('\n').

La méthode:

```
istream & ignore ( int nombre = 1, int delimiteur = EOF );
```

permet d'ignorer (sauter) sur le flux d'entrée auquel on l'applique un certain *nombre* de caractères ou jusqu'à rencontrer le caractère fixé par *delimiteur*. Par défaut le nombre de caractères est fixé à 1 et le délimiteur correspond à la marque de fin de fichier.

Exemples:

```
cin.ignore ( );          // Saute 1 caractere
cin.ignore ( 10 );       // Saute 10 caracteres
cin.ignore ( 10, '*' ); // Saute 10 caracteres ou jusqu'à *
```

La méthode:

```
int sync ( );
```

synchronise le tampon d'entrée avec le flux externe, en d'autres termes: vide complètement le tampon d'entrée.

La méthode:

```
int peek ( );
```

livre le prochain caractère venant du flux d'entrée auquel on l'applique, mais le caractère reste dans le tampon et il sera effectivement consommé lors de la prochaine lecture. Cette méthode permet par exemple de prendre des décisions en fonction du prochain caractère à venir.

Voici un petit exemple complet:

```
/*
  PEEK
*/
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;
#define L_MAX 80
int main ( )
{
  char ch [ L_MAX ];
  int i, j;

  cout << "Un entier ou un mot: ";
  i = cin.peek ( );
  if ( isdigit ( i ) )
  {
    cin >> j;
    cout << "Votre entier: " << j << endl;
  }
  else
  {
    cin >> ch;
    cout << "Votre mot: " << ch << endl;
  }
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Premier exemple d'exécution:

```
Un entier ou un mot: 123
Votre entier: 123
Appuyez sur une touche pour continuer...
```

Deuxième exemple d'exécution:

```
Un entier ou un mot: Toto
Votre mot: Toto
Appuyez sur une touche pour continuer...
```

La méthode:

```
istream & unget ( );
```

remet dans le tampon du flux le dernier caractère consommé (lu) qui sera donc à nouveau consommé comme premier caractère lors de la prochaine lecture. Donc:

```
char c;
cin.get ( c ).unget ( );
```

revient au même que:

```
c = cin.peek ( );
```

Malheureusement rien ne garantit le comportement de la méthode si on l'applique plusieurs fois de suite.

La méthode:

```
istream & putback ( char c );
```

remet dans le tampon du flux le caractère contenu dans *c* qui vient ainsi remplacer le dernier caractère lu. Il sera à nouveau consommé comme premier caractère lors de la prochaine lecture.

Malheureusement rien ne garantit le comportement de la méthode si on l'applique plusieurs fois de suite.

Pour terminer les méthodes de ce groupe:

```
istream & read ( char * chaine, int nombre );
```

Elle lit *nombre* caractères sur le flux auquel on l'applique et les réduit dans *chaine*. En fait on devrait dire octet et non pas caractère, car cette méthode sera normalement utilisée pour des fichiers binaires. La lecture correspond exactement à *nombre* octets, sans aucune interprétation de ceux-ci et sans adjonction dans *chaine* d'une marque de fin de chaîne ('\0'). On l'utilisera en générale en conjonction avec la méthode *write*!

D'autres fonctionnalités seront introduites avec les fichiers binaires et l'accès direct!

❑ *Traitement des erreurs*

Jusqu'à maintenant nous ne nous sommes guère préoccupés du traitement des erreurs dans nos entrées/sorties. Elles surviennent évidemment essentiellement lors de saisies de données (lectures).

Imaginez le petit programme suivant:

```
/*
  Err0
*/
#include <iostream>
#include <cstdlib>
using namespace std;
int main ( )
{
  int  val = 5;
  do
  {
    cout << "Donnez un entier (0 pour terminer): ";
    cin >> val;
    cout << "Votre valeur: " << val << endl;
  } while ( val );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Tout se passe bien tant que l'utilisateur fournit des valeurs raisonnables dont voici un exemple d'exécution:

```
Donnez un entier (0 pour terminer): 12
Votre valeur: 12
Donnez un entier (0 pour terminer): 3
Votre valeur: 3
Donnez un entier (0 pour terminer): 0
Votre valeur: 0
Appuyez sur une touche pour continuer...
```

Mais maintenant si l'utilisateur a la mauvaise idée de donner une lettre ou une chaîne de caractères à la place d'un entier comme demandé, voici ce qui va se passer:

```
Donnez un entier (0 pour terminer): 3
Votre valeur: 3
Donnez un entier (0 pour terminer): a
Votre valeur: 3
Donnez un entier (0 pour terminer): Votre valeur: 3
Donnez un entier (0 pour terminer): Votre valeur: 3
Donnez un entier (0 pour terminer): Votre valeur: 3
…
```

Cela durera aussi longtemps que vous n'arrêterez pas brutalement votre programme. Que s'est-il passé? L'utilisateur introduit une première valeur correcte. Tout se passe pour elle comme avant. A la demande suivante, l'utilisateur introduit le caractère *a* qui ne permet évidemment pas de construire une valeur entière. Il y a donc erreur de lecture et chose importante le caractère *a*, non consommé, reste dans le tampon d'entrée pour la prochaine lecture! De plus, la lecture n'ayant pas pu se faire, la variable qui devait recevoir la valeur lue ne subit aucune modification; dans notre exemple elle conserve sa valeur 3 fournie lors de la première lecture. Dans le programme, comme nous ne traitons pas les erreurs de lecture, cette ancienne valeur 3 est affichée et l'on recommence la boucle! La prochaine lecture retrouve le caractère *a* dans le tampon, ce qui va évidemment à nouveau provoquer la même erreur et ceci sans attendre une quelconque réponse de l'utilisateur. On recommence la boucle … qui devient joyeusement infinie.

Il nous faudra prendre 3 mesures pour résoudre le problème:

1. Détecter l'existence de l'erreur.
2. Effacer l'indicateur d'erreur.
3. Vider le tampon pour ne pas rester bloqué sur le problème.

Pour détecter la présence de l'erreur nous pouvons travailler à différents niveaux. Le plus simple consiste à se baser sur le fait que les opérateurs *( )* et *!* ont été redéfinis pour les flux dans la classe *ios*.

L'opérateur *( )* livre vrai s'il n'y a pas eu d'erreur de traitement sur le flux en question et faux sinon. Ceci nous permet d'écrire des conditions du genre:

```
if ( cin ) … // [1]
```

[1] Attention à la notation, on aurait tendance à écrire if ((cin)) …; ce ne serait pas faux mais inutile!

Bien entendu la majorité des opérateurs et méthodes applicables au flux livrant comme résultat le flux lui-même, nous pouvons tout à fait écrire:

```
if ( cin >> i ) …
```

Généralement une telle opération se trouvera plutôt dans une boucle:

```
while ( cin >> i ) …
```

De manière similaire, l'opérateur *!* livre vrai s'il y a eu erreur de traitement sur le flux en question et faux sinon.

De plus la classe *ios* met à disposition 4 méthodes à résultat booléen permettant de tester les bits d'état du flux. Il s'agit de:

- *good ( )*

  Livre vrai si aucun problème n'a été détecté sur le flux (donc si aucun bit d'erreur n'est activé!) et faux sinon.

  Ainsi notre exemple:

  ```
  if ( cin ) …
  ```

  peut aussi s'écrire:

  ```
  if ( cin.good ( ) ) …
  ```

- *eof ( )*

  Livre vrai si la fin de fichier a été atteinte et faux sinon.

- *fail ( )*

  Livre vrai si une erreur logique s'est produite et faux sinon.

- bad ( )

  Livre vrai si une erreur physique s'est produite et faux sinon.

De plus il existe aussi la méthode:

- rdstate ( )

  Elle livre sous la forme d'un entier la valeur de l'ensemble des bits d'état du flux.

La classe *ios* met à disposition des constantes représentant la valeur de chacun de ces bits. Il s'agit de: *ios::goodbit, ios::eofbit, ios::failbit, ios::badbit*.

Ces éléments nous permettent de détecter les erreurs. Comme nous l'avons déjà signalé, après sa détection il faudra la traiter. Pour une lecture cela implique entre autres de vider le tampon pour que la prochaine tentative de lecture ne provoque pas à nouveau la même erreur. Cette opération peut se réaliser par exemple en utilisant un *getline*! Reste encore au minimum à

annuler l'indicateur d'erreur, c'est-à-dire à remettre les bits indicateurs d'erreurs à 0. Le plus simple consiste à les remettre tous à 0[1]. C'est le but de la méthode:

```
void clear ( int val = 0 );
```

Exemple:

```
cin.clear ( );  //[2]
```

Lorsque nous réalisons des lectures sur des objets d'un type que nous nous sommes définis (normalement par surcharge de >>) nous devons aussi traiter les erreurs et donc en principe activer en conséquence les bits du mot d'état du flux! Ceci se réalise simplement par la méthode *clear*.

Exemple:

```
cin.clear ( ios::failbit );
```

Dans ce cas, seul le bit spécifié est mis à 1, les autres étant forcés à 0. Si nous désirons mettre un bit à 1 sans modifier la valeur des autres bits du mot d'état nous utiliserons une formulation du genre:

```
cin.clear ( ios::failbit | cin.rdstate ( ) );
```

Voici une version modifiée de notre programme pour la lecture sans erreur d'entiers, prenant en compte la détection et le traitement des erreurs:

[1] Cela ne pose pas de problème avec *goodbit* qui n'est pas réellement un bit!

[2] Attention, le nom de cette méthode est trompeur comme vous allez le constater. En fait il s'agit de mettre la valeur transmise en paramètre dans le mot représentant les bits d'état.

```
/*
  Err1
*/
#include <iostream>
#include <cstdlib>
using namespace std;
int main ( )
{
  int  val = 1;  // =1 en cas d'erreur la 1ere fois

  /* Tant que l'utilisateur le veut… */
  do
  {
    cout << "Donnez un entier (0 pour terminer): ";
    if ( cin >> val )        // [1]
      cout << "Votre valeur: " << val << endl;
    else
    {
      cin.clear ();  // Annule les bits d'erreur
      while ( cin.get ( ) != '\n' );
      cout << "Erreur de donnee, recommencez!" << endl;
    }
  } while ( val );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Cette fois, lors de l'exécution tout se passe comme on pouvait l'espérer, à titre d'exemple:

```
Donnez un entier (0 pour terminer): 3
Votre valeur: 3
Donnez un entier (0 pour terminer): a
Erreur de donnee, recommencez!
Donnez un entier (0 pour terminer): qwertz
Erreur de donnee, recommencez!
Donnez un entier (0 pour terminer): 1
Votre valeur: 1
Donnez un entier (0 pour terminer): 0
Votre valeur: 0
Appuyez sur une touche pour continuer...
```

[1] Dans certains cas il sera préférable de vider le tampon lorsque que la lecture se fait correctement!

❑ *Flux et les fichiers externes*

Il nous faut maintenant faire la liaison avec les fichiers externes. En fait rien de bien compliqué puisque nous pouvons utiliser tout ce que nous connaissons déjà dans ce domaine. Le seul réel problème qui nous reste à résoudre réside dans le lien à faire entre notre programme et un fichier externe. Dans ce but nous devons inclure un nouveau fichier d'en-tête: *fstream*.

Il nous permet tout d'abord de déclarer dans le programme des objets de type "fichier".

Pour un flux d'entrée (traité en lecture uniquement) il s'agit d'un objet de la classe *ifstream*:

```
ifstream fichierSource;
```

Pour un flux de sortie (traité en écriture uniquement) il s'agit d'un objet de la classe *ofstream*:

```
ofstream fichierSortie;
```

Note: les 2 flux pourraient être de la classe *fstream* qui hérite des 2 précédentes (en réalité, de *istream*, respectivement *ostream*). Ceci nous permettra par la suite de traiter le même fichier en lecture et en écriture ce qui sera très utile pour la gestion des fichiers binaires à accès direct:

```
fstream fichier;
```

Une fois l'objet déclaré nous devons ouvrir le flux, c'est-à-dire faire le lien avec le fichier externe. Une telle opération se réalise par l'intermédiaire de la méthode *open* appliquée au flux. Elle possède 2 paramètres:

1. Le nom du fichier externe sous forme d'une chaîne de caractères[1].
2. Le mode dans lequel on désire traiter le flux (le fichier). Les différents modes possibles, dont nous vous donnerons la liste ci-après, sont définis par des constantes de la classe *ios*.

Exemples:

```
fichierSource.open ( "Source.txt", ios::in  );
fichierSortie.open ( "Sortie.txt", ios::out );
fichier.open ( "Donnees.dat", ios::in | ios::out );
```

[1] La forme de cette chaîne dépend du système d'exploitation sur lequel on travaille!

En pratique un constructeur des classes *ifstream*, *ofstream* et *fstream* permet de réaliser l'ouverture (le lien avec le fichier externe) au moment de la création du flux. Ce constructeur possède les 2 mêmes paramètres qu'*open*.

Exemple:

```
ifstream fichierSource ( "Source.txt", ios::in  );
ofstream fichierSortie ( "Sortie.txt", ios::out );
fstream fichierSortie  ( "Donnees",
                         ios::binary | ios::out | ios::in );
```

Nous constatons que les modes peuvent se combiner! En voici la liste complète:

ios:in Ouverture en lecture; interdit pour un flux de classe *ofstream*; obligatoire et valeur par défaut pour un flux de classe *ifstream*.

ios::out Ouverture en écriture; interdit pour un flux de classe *ifstream*; obligatoire et valeur par défaut pour un flux de classe *ofstream*.

ios::trunc Ecrase un ancien fichier de même nom qui existerait déjà. N'a de sens (et est obligatoire si pas *ios::ate* ou *ios::app*) que pour un fichier de sortie (valeur par défaut) ou d'entrée/sortie.

ios::app Ouvre un fichier en "adjonction" de telle sorte que l'on puisse écrire séquentiellement à la suite de son contenu actuel.

ios::ate Ouvre un fichier en "adjonction" de telle sorte que l'on puisse écrire à la suite de son contenu actuel, mais permet aussi un accès direct.

ios::binary Ouvre le fichier en tant que fichier binaire et non pas texte. Ceci implique qu'aucune interprétation des fins de lignes ne doit être faite. Dans plusieurs environnement cette distinction n'est pas nécessaire (mais elle l'est sous Windows!).

Avant de donner des exemples de programmes complets signalons encore l'existence des méthodes:

```
bool is_open ( );
```

Comme son nom le laisse supposer elle permet de tester si un flux est associé ou non à un fichier externe.

```
void close ( );
```

Elle permet de fermer le fichier donc de rompre la liaison avec le fichier externe. Cela vide le tampon associé au flux et éventuellement libère la place qu'il occupe en mémoire. Même si théoriquement les fichiers se ferment automatiquement à la fin de l'exécution du programme, il

faut, pour des raisons pratiques de sécurité et d'efficacité, appeler la méthode *close* dès que l'on a plus besoin d'utiliser le fichier!

Voilà, passons aux exemples et reprenons notre copie de fichier que nous avions largement développée dans le chapitre consacré au traitement des fichiers en C.

```
/*
  Recopie d'un fichier (caractere par caractere) dans un autre fichier
  COPIEFICHIER1a.cpp
*/
#include <iostream>
#include <fstream>
using namespace std;
/* Longueur maximale d'un nom de fichier */
int main ( )
{
  /* Longueur maximale d'un nom de fichier */
  const int MAX_NOM = 30;
  char caractere;
  char nomDuSource [ MAX_NOM ], nomDeSortie [ MAX_NOM ];
  ifstream fichierSource;
  ofstream fichierSortie;

  /* Demander le nom des fichiers */
  cout << "Donnez le nom du fichier source: ";
  cin  >> nomDuSource;
  cout << "Donnez le nom du fichier de sortie: ";
  cin  >> nomDeSortie;

  /* Creation des flux */
  fichierSource.open ( nomDuSource, ios::in );
  fichierSortie.open ( nomDeSortie, ios::out );

  /* Controle!! */
  if ( ( fichierSource ) && ( fichierSortie  ) )
  {
    /* Traiter tous les caracteres du fichier */
    while ( fichierSource.get ( caractere ) )
      fichierSortie.put ( caractere );
    fichierSource.close ( );
    fichierSortie.close ( );
  }
  else
    cout << "Erreur\n";

  cout << "\nFin du traitement...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Notes:

- Dans une véritable application, il sera certainement préférable de contrôler la validité de l'ouverture de chaque fichier immédiatement après chaque demande de nom. Ainsi on pourra recommencer en ne redemandant que ce nom!
- Dans la boucle **while** nous nous basons sur le fait qu'en fin de fichier le *eofbit* est activé, ce qui, au niveau de la condition, nous fait sortir de la boucle à ce moment-là!

Nous pourrions donner de nombreuses variantes de ce programme, comme nous l'avions fait pour les fichiers C. Nous allons ici nous contenter d'une unique deuxième version montrant l'ouverture des fichiers lors de la création des objets ainsi qu'une pseudo copie binaire de notre fichier source:

```
/*
  Recopie d'un fichier (octet par octet) dans un autre fichier
  COPIEFICHIER2.cpp
*/
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
int main ( )
{
  const int MAX_NOM = 30;
  char octet;
  char nomDuSource [ MAX_NOM ], nomDeSortie [ MAX_NOM ];
  /* Demander le nom des fichiers */
  cout << "Donnez le nom du fichier source: ";
  cin  >> setw ( MAX_NOM - 1 ) >> nomDuSource;
  cout << "Donnez le nom du fichier de sortie: ";
  cin  >> setw ( MAX_NOM - 1 ) >> nomDeSortie;
  /* Creation et ouverture des flux */
  ifstream fichierSource ( nomDuSource, ios::binary | ios::in );
  ofstream fichierSortie ( nomDeSortie, ios::binary | ios::out );
  /* Controle!! */
  if ( ( fichierSource ) && ( fichierSortie  ) )
  {
    /* Traiter tous les octets du fichier */
    while ( fichierSource.read ( &octet, 1 ) )
      fichierSortie.write ( &octet, 1 );
    fichierSource.close ( );
    fichierSortie.close ( );
  }
  else
    cout << "Erreur\n";

  cout << "\nFin du traitement...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

❑ *Accès direct*

Comme nous l'avons déjà signalé, en C/C++ un fichier correspond logiquement à une suite d'octets. On ne dispose donc pas a priori de la notion d'enregistrement. A n'importe quel moment nous pouvons nous positionner sur n'importe quel octet du fichier pour que la prochaine lecture ou écriture se fasse à partir de cette position. De plus, un objet de la classe *fstream* pouvant être préparé (ouvert) aussi bien pour des lectures que des écritures, ces opérations peuvent s'enchaîner en alternance sans aucun problème. Toutefois vous ne déduirez pas de l'exemple que nous vous présenterons dans la suite que tout traitement se fait toujours par une alternance de lectures et d'écritures. Le fichier peut très bien n'être ouvert qu'en lecture et le programme y rechercher des données en différents endroits.

Le problème principal réside dans le fait qu'il incombe au programmeur de gérer correctement le positionnement dans le fichier pour y retrouver l'information désirée! Une fois qu'il sait sur quel octet il veut/doit se positionner (certainement suite à un calcul) il dispose de 2 méthodes pour effectuer l'opération puisqu'il existe 2 pointeurs distincts (2 positions courantes), l'un pour les objets *ifstream* l'autre pour ceux de la classe *ofstream*.

Pour la classe *ifstream* la méthode s'appelle: *seekg* et pour *ofstream*: *seekp*. A part cela, les 2 méthodes fonctionnent selon le même principe et avec les mêmes paramètres. Toutes 2 livrent en retour le flux sur lequel on les applique, ce qui permet entre autres d'enchaîner, après un positionnement, une lecture ou une écriture à l'endroit désiré.

Les méthodes possèdent 2 paramètres:

- Le premier, une valeur entière, fixe la grandeur du déplacement à réaliser en nombre d'octets.
- Le deuxième, une constante entière dont les 3 valeurs possibles sont définies dans *ios*. A savoir:
-
    - **ios::beg** le déplacement se fait par rapport au début du fichier (valeur par défaut).
    - **ios::cur** le déplacement se fait par rapport à la position courante dans le fichier (tout dépend donc de l'opération réalisée avant).
    - **ios::end** le déplacement se fait par rapport à la fin du fichier.

Une autre méthode nous permet en tout temps d'obtenir la position courante dans le fichier. En fait à nouveau 2 méthodes, une pour *ifstream* et l'autre pour *ofstream*. Elles correspondent respectivement aux prototypes:

```
int tellg ( );
int tellp ( );
```

Voilà, vous connaissez tout ce qu'il faut savoir sur l'accès direct aux fichiers. Il ne reste plus qu'à mettre en pratique. Vous relèverez particulièrement dans le programme qui suit l'utilisation du positionnement en fin de fichier afin d'en obtenir la taille en octets (elle correspond à cette position!). Il ne reste plus alors qu'à diviser par la taille d'un objet (un enregistrement!) pour en connaître le nombre.

Voici ce code, qui correspond à ce que nous avions déjà fait en C:

```
/*
  Gestion d'un fichier en acces direct. Cree un fichier d'entiers,
  le trie dans le fichier et l'affiche
  ================================================
  = C'est un exemple de ce que l'on peut faire,
  = pas de ce que l'on doit faire!!
  ================================================
  ACCESDIRECT.cpp
*/
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
using namespace std;


/* Affiche le contenu d'un fichier d'entiers */
void afficher ( fstream & fichier );
/* Trie un fichier d'entiers directement dans le fichier */
void trier ( fstream & fichier );

int main ( )
{
  int valeur;
  /* Preparation du fichier */
  fstream fichier ( "temp.tmp",
                    ios::in |ios::out |ios::trunc |ios::binary );
```

```
  /* Si le fichier a ete ouvert correctement */
  if ( fichier )
  { /* Traiter toutes les valeurs de l'utilisateur */
    while ( 1 )
    {
       cout << "Donnez une valeur entiere (0 pour terminer): ";
       cin >> valeur;
       /* sortie si terminé! */
       if ( valeur == 0 ) break;
       fichier.write ( (char *) &valeur, sizeof ( int ) );
    }

    cout << "\n\nFichier original:\n";
    afficher ( fichier );
    trier    ( fichier );
    cout << "\n\nFichier trie:\n";
    afficher ( fichier );
    fichier.close   ( );
  }
  else
    cout << "Erreur\n";

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}



/* Affiche le contenu d'un fichier d'entiers */
void afficher ( fstream & fichier )
{
  int valeur;
  fichier.seekg ( 0, ios::beg );
  cout << "\n\ncontenu du fichier:\n\n";

  /* Traiter tous les elements du fichier */
  while ( fichier.read ( (char *) &valeur, sizeof ( int ) ) )
    cout << valeur << endl;
  fichier.clear ();
}
```

```
/* Trie un fichier d'entiers directement dans le fichier */
void trier ( fstream & fichier )
{
  int valeurInf, valeurSup;

  /* Fixer le nombre d'enregistrements */
  int dernierEnregistrements =
          fichier.seekg ( 0, ios::end ).tellg ( ) / sizeof ( int );
  for ( int i = 0; i < dernierEnregistrements - 1; i++ )
    for ( int j = i; j < dernierEnregistrements; j++ )
    { fichier.seekg ( i * sizeof ( int ), ios::beg )
          .read ( (char *) &valeurInf, sizeof ( int ) );
      fichier.seekg ( j * sizeof ( int ), ios::beg )
          .read ( (char *) &valeurSup, sizeof ( int ) );
      /* Faut-il croiser les valeurs */
      if ( valeurInf > valeurSup )
      { fichier.seekp  ( i * sizeof ( int ), ios::beg )
          .write ( (char *) &valeurSup, sizeof ( int ) );
        fichier.seekp  ( j * sizeof ( int ), ios::beg )
          .write ( (char *) &valeurInf, sizeof ( int ) );
      }
    }
}
```

Les fonction génériques

Introduction

La notion que nous abordons ici n'existe pas en C.

Nous avons choisi le titre de *génériques* par analogie avec d'autres langages (Ada). En C++ on utilise généralement soit le terme anglais: *template* ou l'une de ses traductions françaises: *patrons* ou *modèles*.

Dans ce chapitre nous n'étudierons que des fonctions génériques, mais cette notion s'applique aussi aux classes! Dans le cas des classes les possibilités seront plus étendues que pour les fonctions, mais une très grande partie de ce que nous présentons ici restera valable.

Nous savons déjà que nous pouvons surcharger une fonction, c'est-à-dire que plusieurs fonctions peuvent avoir le même nom pour autant que leur signature diffère (nombre et/ou type des paramètres). Toutefois cette possibilité nous oblige à réécrire le code complet de ces fonctions, même s'il reste strictement identique (même algorithme!) seul le type des paramètres ayant changé. La généricité va nous permettre, elle, de paramétrer une fonction de telle sorte que son code qui reste formellement le même puisse s'adapter à des types différents.

Considérons une fonction générique comme un moule (donc non directement utilisable!) à partir duquel nous créons par instanciation des fonctions effectives adaptées aux paramètres désirés. Ces paramètres représentent des types, ceux que l'on veut appliquer dans la fonction réelle. Les fonctions ainsi instanciées (créées) s'utilisent comme n'importe quelle autre fonction classique.

❑ *Fonctions génériques et paramètres*

La définition d'une fonction se réalise de manière simple. Devant la déclaration normale d'une fonction usuelle vous ajoutez le mot réservé **template** suivi entre < > d'un ou de plusieurs paramètres génériques formels, soit un identificateur (qui représentera un type) précédé du mot réservé **class**. Ce mot **class** n'est pas heureux! Il peut effectivement s'agir, au moment de l'instanciation, d'un type défini sous forme d'une classe, mais aussi d'un type de base quelconque ou encore d'un type défini par l'utilisateur (**enum**, **struct**, **union** ou champ de bits). La version 3 de la norme a voulu rectifier la situation en introduisant la possibilité de mettre le nouveau mot réservé **typename** à la place de **class**. Les 2 formes sont donc théoriquement possibles, mais les habitudes se prennent rapidement et (presque) tout le monde continue à utiliser **class**. A regret nous ferons donc de même.

Voici un premier exemple. Des règles plus précises suivront:

```
template <class t> void echange ( t  &v1, t &v2 )
{
  t temp = v1;
  v1 = v2;
  v2 = temp;
}
```

Pour les fonctions, l'instanciation se fera automatiquement par un simple appel usuel[1], le type des paramètres effectifs de la fonction appelée déterminant le lien à réaliser avec chaque paramètre de la généricité:

```
long int  i1, i2;
char c1, c2;
...
/* cree une fonction echange pour des entiers longs */
echange ( i1, i2 );
...
/* cree une fonction echange pour des caracteres */
echange ( c1, c2 );
...
```

Nous avons dit qu'une fonction générique peut comporter plusieurs paramètres de

[1] Nous traiterons de situations ambiguës par la suite.

généricité:

```
template <class t1, class t2> void f ( t1 v1, t2 v2 )
...
```

Chaque type donné comme paramètre générique doit apparaître au moins une fois dans les paramètres formels de la fonction pour réaliser une instanciation automatique (non forcée !). Sauf mention explicite que nous étudierons par la suite, ce sont eux qui permettent d'attribuer un type explicite aux paramètres de généricité.

De plus les paramètres de généricité peuvent aussi s'utiliser:

- Pour fixer le type du résultat de la fonction.
- Pour fixer le type de déclarations locales à la fonction.
- Dans les instructions de la fonction par exemple comme opérateurs de conversion (cast) ou pour en fixer la taille (**sizeof**).

La fonction générique peut comporter d'autres paramètres que ceux d'un type générique. Généralement on parle alors de paramètres "expressions". Il s'agit là de paramètres usuels de fonctions. Les paramètres effectifs doivent pour ceux-ci correspondre au type annoncé, ou tout au moins à un type permettant une conversion automatique vers le type du paramètre formel.

Donnons maintenant un exemple de programme complet. Nous ajouterons quelques remarques complémentaires à sa suite.

```
/*
  Premier exemple de generique
    GENERIQUE1
*/
#include <cstdlib>
#include <iostream>
using namespace std;
/* Definition du type vecteur */
struct vecteur
{
  float dx;
  float dy;
};
/* Addition de 2 vecteurs */
vecteur operator + ( const vecteur &, const vecteur & );
/* Affichage d'un vecteur */
void affiche ( const vecteur & );
```

```
/*
  Fonction generique livrant la somme des
    elements d'un tableau
*/
template <class t> t somme ( t tab [], int n );


int main ( )
{
  int t1 [3]     = { 1, 2, 3 };
  float t2 [4]   = { 0.0, 0.1, 0.2, 0.3 };
  vecteur t3 [3] = { {1,2},{3,4},{5,6} };
  cout << "Somme d'un tableau d'entiers:" << endl;
  cout << somme ( t1, 3 ) << endl;
  cout << "Somme d'un tableau de reels:" << endl;
  cout << somme ( t2, 4 ) << endl;
  cout << "Somme d'un tableau de vecteurs:" << endl;
  affiche ( somme ( t3, 3 ) );
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}


template <class t> t somme ( t tab [], int n )
{
  t temp = tab [0];
  for ( int i = 1; i < n; i++ )
    temp = temp + tab [i];
  return temp;
}


/* Addition de 2 vecteurs */
vecteur operator + ( const vecteur &v1, const vecteur &v2 )
{
  vecteur temp = { v1.dx + v2.dx, v1.dy + v2.dy };
  return temp;
}


/* Affichage d'un vecteur */
void affiche ( const vecteur &v )
{
  cout << "( " << v.dx << ", " << v.dy << " )";
}
```

L'exécution de ce programme livre comme résultat:

```
Somme d'un tableau d'entiers:
6
Somme d'un tableau de reels:
0.6
Somme d'un tableau de vecteurs:
( 9, 12 )
Fin du programme...Appuyez sur une touche pour continuer...
```

Relevons encore sur notre fonction générique *somme* les points spécifiques suivants:

- Nous initialisons le calcul de la *somme* avec le premier élément du tableau. Nous ne pouvons pas l'initialiser à 0, ce qui serait correcte pour des tableaux d'éléments de type **int** ou **float** mais pas pour un tableau de *vecteur*.
- Pour la même raison, au moment de faire la somme nous ne pouvons pas utiliser l'opérateur +=. Par contre nous pouvons utiliser l'opérateur plus car nous l'avons surchargé pour le type *vecteur*.
- De manière générale, nous ne pouvons pas utiliser notre fonction générique *somme* sur un tableau d'éléments d'un type que nous nous serions défini et pour lequel nous n'aurions pas donné une surcharge de l'opérateur +.

Ceci nous montre que nous pouvons réaliser des fonctions génériques suffisamment souples et s'appliquant à de nombreux types. Cette technique nous évite de réécrire de nombreuses fois le même code de fonctions où seuls les types changent, mais elle nous oblige aussi à être attentifs à certains détails si nous voulons effectivement que les génériques soient utilisables dans de nombreuses situations. N'oubliez pas que souvent c'est après coup que nous pensons à utiliser telle ou telle partie de code déjà réalisée.

❑ *Quelle instance pour quel paramètre?*

Nous avons dit que les paramètres transmis au moment de l'appel de la fonction déterminent la nature de l'instanciation à effectuer. Cela signifie entre autre que les paramètres effectifs doivent correspondre strictement au type sélectionné. En d'autres termes, il n'y a pas, comme pour les fonctions usuelles, de conversion implicite.

Exemple:

```
template <class t> void fonction ( t v1, t v2 )...
```

Avec des déclarations du genre:

```
int   i1   =  1, i2 = 2, i3 = 3;
int   *pi1 =  &i1, *pi2 = &i2;
char  c1   =  'a', c2 = 'z';
float f1   =  1.6, f2 = 2.6;
```

Nous pouvons très bien appeler notre fonction:

```
fonction ( i1, i2 );
```

qui correspond à une fonction réelle (une instanciation) basée sur le type **int**. Pour l'appel:

```
fonction ( f1, f2 );
```

elle correspond à une fonction réelle (une instanciation) basée sur le type **float**.

Par contre les appels suivants:

```
fonction ( i1, f2 );  // Erreur!!!
fonction ( f1, i2 );  // Erreur!!!
fonction ( i1, c2 );  // Erreur!!!
```

provoquent tous des erreurs à la compilation. Remarquez que c'est même le cas entre caractère et entier!

Forcer une instance spécifique

Si nous avons de bonnes raisons de vouloir une instance spécifique, nous pouvons au moment de l'appel forcer les types. Pour cela il suffit lors de cet appel de faire suivre le nom de la fonction (avant d'en donner les paramètres effectifs) du ou des type(s) désiré(s) mis entre < >.

Exemple avec notre *fonction* ci-dessus:

```
fonction<int> ( i1, i2 );
```

L'exemple ci-dessus ne présente pas un grand intérêt puisqu'il correspond à ce qui se passe par défaut. Par contre l'appel:

```
fonction<int> ( i1, c2 );
```

nous permet effectivement d'obtenir une instance basée sur les entiers mais à laquelle nous avons le droit de transmettre un paramètre de type **int** et un autre de type **char**.

Bien qu'un peu surprenant, l'appel:

```
fonction<float> ( i1, i2 );
```

devient tout à fait correct.

De même que:

```
fonction<float> ( f1, i2 );
```

qui ne pose aucun problème particulier!

Par contre:

```
fonction<int> ( i1, f2 );
```

ou

```
fonction<int> ( f1, f2 );
```

provoquent certainement un avertissement du compilateur puisqu'il va y avoir une conversion dégradante potentiellement dangereuse du type **float** vers **int**!

Les principes et les problèmes restent les mêmes pour une fonction possédant plusieurs paramètres de généricité:

```
template <class t1, class t1>
        void fct ( t1 v1, t2 v2 )...
```

Exemple d'appel forcé:

```
fct<int, float> ( ... );
```

Les 2 paramètres de généricité peuvent posséder le même type:

```
fct<int, int> ( ... );
```

ce qui d'ailleurs peut aussi se produire sous la forme d'un appel normal:

```
fct ( f1, f2 );
```

Toutefois nous pouvons ne forcer que le premier paramètre, l'autre se basant sur les règles usuelles en fonction du paramètre effectif transmis lors de l'appel:

```
fct<int> ( ... );
```

Le mécanisme se généralise à plus de 2 paramètres!

Si une fonction générique possède des paramètres de généricité qui n'apparaissent pas dans les paramètres formels de la fonction, son instanciation devra se faire de manière forcée selon le mécanisme présenté ci-dessus.

Spécialisation

Il n'existe guère de possibilités de restreindre la nature des paramètres effectifs donnés lors d'une instance de générique! Prenons la fonction:

```
template <class t> bool fonction ( t v1, t v2 )...
```

Imaginons maintenant que dans les instructions de cette fonction nous réalisons l'addition de ces 2 paramètres. Tout se passe bien si nous instancions avec des **int** ou des **float** par exemple! Si nous instancions avec des **int** * ou des **float** *, les choses se gâtent car le compilateur ne connaît pas l'addition de 2 pointeurs! L'erreur sera signalée dès la compilation! Par contre si dans le corps de la fonction nous comparons les 2 paramètres pour livrer vrai si le premier a une valeur plus grande que le deuxième, le compilateur peut générer une comparaison sur 2 pointeurs. Mais la fonction fait-elle ce que l'on désire? Certainement pas! Il va comparer les 2 adresses et non pas les 2 objets pointés.

La spécialisation des fonctions va nous aider à résoudre ce problème. Tout d'abord la fonction:

```
bool fonction ( int *v1, int *v2 )
{
  return *v1 > *v2;
}
```

peut cohabiter dans le même programme avec la fonction générique. Elle sera utilisée automatiquement lorsque l'on appelle *fonction* avec comme paramètres 2 pointeurs sur des entiers. Par contre dans le cas de paramètres d'un autre type nous aboutirons, dans la mesure du possible, à une instantiation de la fonction générique.

Cette solution toutefois présente un inconvénient majeur: elle ne fonctionne que pour des pointeurs sur des entiers! Nous aimerions certainement qu'elle puisse se généraliser pour des pointeurs sur un type quelconque. Remplaçons la fonction "standard" proposée par une deuxième fonction générique qui prend la forme:

```
template <class t> bool fonction ( t *v1, t *v2 )
{
  return *v1 > *v2;
}
```

Elle aussi demeure compatible avec notre première fonction générique et sera instanciée dès qu'un appel comporte 2 pointeurs sur un même type.

Surcharge

Une fonction générique, comme une fonction usuelle peut être surchargée, cela ne pose en fait aucun problème particulier.

Exemple:

```
template <class t> bool fonction ( t v1, t v2 ) ...
et
template <class t> bool fonction ( t v1, t v2, t v3 ) ...
```

La surcharge ne se fait pas forcément que sur des paramètres génériques:

```
template <class t> bool fonction ( t v1, t v2, float v3 )
```

Dans ce cas, les règles de substitution/promotion usuelles s'appliquent pour le troisième paramètre de cette nouvelle surcharge.

Attention toutefois, les 2 dernières surcharges ci-dessus, qui peuvent en tant que définition cohabiter dans le même module, peuvent aussi provoquer des situations ambiguës suivant la nature du paramètre de généricité.

❑ *Exportation*[1]

A priori une fonction générique n'est visible (utilisable) que dans l'unité de compilation de sa définition. Redonner dans chaque module une telle définition va à l'encontre de l'objectif visé! Pour cette raison la norme a introduit le mot réservé **export**:

```
export template <class t1 > void f ( t1 v1) ...
```

On fait précéder la définition de la fonction générique de **export**, ce qui la rend globalement disponible pour d'autres unités.

Le module désirant utiliser cette fonction en donnera une simple déclaration (l'équivalent d'un prototype usuel!):

```
template <class t1 > void f ( t1 v1);
```

En pratique le prototype sera certainement mis à disposition par l'intermédiaire d'un fichier d'inclusion.

Vous trouverez à la page suivante un exemple artificiel résumant l'essentiel des points présentés ci-dessus.

[1] Attention, de nombreux environnements ne supportent pas (encore!?) cette possibilité. C'est malheureusement aussi le cas de notre environnement de travail!

```
/*
  Exemple de generiques
    GENERIQUE2
*/
#include <cstdlib>
#include <iostream>
using namespace std;


/*
  Fonction generique realisant l'addition de ses parametres
  F1
*/
template <class t> t oper ( t v1, t v2 )
{
  return v1 + v2;
}


/*
  Fonction generique realisant l'addition de ses parametres
  Specialisation de la fonction somme
  F2
*/
template <class t> t oper ( t *v1, t *v2 )
{
  return *v1 + *v2;
}


//int oper ( int *v1, int *v2 )
//{
//  return *v1 - *v2;
//}


/*
  Fonction generique realisant l'addition de ses parametres
  Surcharge/surdefinition de la fonction somme
  F3
*/
template <class t> t oper ( t v1, t v2, t v3 )
{
  return v1 + v2 + v3;
}
```

```
int main ( )
{
  int   i1   =  1, i2 = 2, i3 = 3;
  int   *pi1 =  &i1, *pi2 = &i2;
  char  c1   =  'a', c2 = 'z';
  float f1   =  1.6, f2 = 2.6;
  float *pf1 =  &f1, *pf2 = &f2;

  cout << "Somme 2 entiers (F1): " << oper ( i1, i2 )
       << endl;  // Instance de F1
  cout << "Somme 2 pointeurs sur des entiers (F2): "
       << oper ( pi1, pi2 ) << endl;    // Instance de F2
  cout << "Somme 3 entiers (F3): " << oper ( i1, i2, i3 )
       << endl;  // Instance de F3
  cout << "Somme 2 reels (F1): " << oper ( f1, f2 )
       << endl;  // Instance de F1
  cout << "Somme 2 pointeurs sur des reels (F2): "
       << oper ( pf1, pf2 ) << endl;   // Instance de F2
  cout << "Somme 2 reels forces int (F1): " << oper <int> ( f1, f2 )
       << endl;  // Instance de F1
  cout << "Somme 1 entier et 1 caractere force int (F1): "
       << oper <int> ( i1, c2 ) << endl; // Instance de F1

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution de ce programme nous donne l'affichage suivant:

```
Somme 2 entiers (F1): 3
Somme 2 pointeurs sur des entiers (F2): 3
Somme 3 entiers (F3): 6
Somme 2 reels (F1): 4.2
Somme 2 pointeurs sur des reels (F2): 4.2
Somme 2 reels force int (F1): 3
Somme 1 entier et 1 caractere force int (F1): 123

Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *Inférence de type avec auto et decltype (C++ 11)*

Le mot-clé **auto** indique au compilateur de déterminer lui-même le type d'une variable, dans le cas d'une déclaration de variable avec initialisation. Le compilateur donne à la variable automatiquement le type de l'expression utilisée pour l'initialiser (on parle d'inférence de type). Ceci peut réduire l'effort d'écriture et on évite des fautes de frappe dans le cas de types complexes.

Exemples :

```
auto a = 2; // a est du type int
auto b = 2.9; // b est du type double
```

Le programme suivant montre différentes applications de **auto** avec des types de base et des types définis par l'utilisateur :

```
/*
  Exemple: Utilisation de auto
*/
#include <iostream>
#include <vector>  // type vector
#include <string>  // type string
#include <cstddef> // size_t
using namespace std;

struct Point {
  int x;
  int y;
};

int main() {
  Point p1 = { 100, 200 };
  auto p2 = p1;  // p2 est du type Point
  cout << "p2.x= " << p2.x << " p2.y= " << p2.y << endl;

  vector<double> v1 = { 1.1, 2.2, 3.3, 4.4, 5.5 };
  auto v2 = v1;  // v2 est du type vector<double>
  for (size_t i = 0; i < v2.size(); ++i) {
    cout << i << ": " << v2[i] << endl;
  }

  string s1("Fin!");
  auto s2 = s1;  // s2 est du type string
  cout << s2 << endl;

  return EXIT_SUCCESS;
```

```
}
```

Un mécanisme similaire à **auto** permet de laisser le compilateur déterminer le type d'une variable à partir d'une *expression*. Pour cela, on déclare les variables en utilisant le mot-clé **decltype** suivi de l'expression mise entre parenthèses.

Exemples :

```
int n;
double p;
decltype (n*p) pTotal; // pTotal sera du type n*p,
                       // soit ici double

vector<int> v1;
decltype (v1) v2;  // v2 sera du type vector<int>
```

Les exemples vus jusqu'ici ont pour seul but d'illustrer le fonctionnement de l'inférence de type et utilisent volontairement des types simples. Ce ne sont pas de bons exemples d'utilisation car pour un développeur qui lit le code le type de la variable est plus difficile à déterminer. Il est conseillé de réserver l'inférence de type aux variables qui ont une très petite portée, comme par exemple les variables de boucle (voir la section ci-après « Instruction for généralisée (C++11) »).

Il y a pourtant des problèmes (voir exemple ci-dessous) où l'utilisation de **auto** et **decltype** est la seule solution possible. Supposons qu'un programme ait besoin d'initialiser des vecteurs avec une séquence de valeurs croissante. Il y a une fonction pour créer et initialiser un vecteur de **int** et une autre pour un vecteur de **float**. Chacune des fonctions prend un argument qui définit la séquence à créer et qui est du même type que les éléments du vecteur. Les deux fonctions sont surchargées.

```
vector<int> creerSequence(int valeur) {
    vector<int> v(5);
    for (int i = 0; i < 5; i++) {
      v[i] = valeur;
      valeur += valeur;
    }
    return v;
}

vector<float> creerSequence(float valeur) {
    vector<float> v(5);
    for (int i = 0; i < 5; i++) {
      v[i] = valeur;
      valeur += valeur;
    }
    return v;
}
```

Supposons maintenant que nous voulons écrire une fonction générique pour créer une

séquence qui utilise les fonctions précédentes. La fonction peut être appelée aussi bien avec une valeur du type **int** pour retourner une séquence de **int** ou une valeur du type **float** pour retourner une séquence de **float**.

```
template <class T> ??? sequence(T valeur) {
    return creerSequence(valeur);
}
```

Se pose alors le problème de comment déclarer le type de la valeur de retour (marqué dans l'exemple avec « ??? »). Ce problème peut être résolu avec **auto**, **decltype** et une notation dite « retardée » pour déclarer le type de la valeur de retour après l'en-tête de la fonction :

```
template <class T> auto sequence(T valeur)
-> decltype(creerSequence(valeur)) {
    return creerSequence(valeur);
}
```

Cette déclaration dit au compilateur que la valeur de retour est du même type que l'expression `creerSequence(valeur)`, et le compilateur résout la surcharge des fonctions avec le type du paramètre `valeur`.

❑ *Instruction for généralisée (C++11)*

Une forme compacte pour écrire une boucle for permet d'itérer sur un objet disposant d'un itérateur, ceci en améliorant la lisibilité du programme. Ansi avec la déclaration suivante :

```
int t[200];
```

on peut écrire de manière succincte :

```
for (int val : t) { // traite val }
```

au lieu d'une boucle traditionnelle :

```
for (int i = 0 ; i < 200 ; i++) {
    int val = t[i];
    // traite val
}
```

Ceci met à disposition la valeur de chaque élément dans la variable `val` et la création et gestion de la variable d'itération sont prises en charge par le compilateur. Notons que la variable `val` est une copie de l'élément et non pas l'élément lui-même. Il est également possible d'utiliser comme variable de boucle une référence ce qui permet d'accéder directement aux éléments et les modifier en écriture :

```
for (int& elem : t) {
    elem = 0;
}
```

Si l'objet est un container on peut en plus utiliser l'inférence de type avec **auto** et écrire élégamment :

```
vector<int> v;
for (auto elem : v) { // traite elem }
```

au lieu du traditionnel :

```
for (vector<int>::iterator it = v.begin();
     it != v.end();
     it++)  {
    int elem = *it;
    // traite elem
}
```

❑ *Classes génériques*

Bien que cela ne soit pas l'objet de ce chapitre et que nous ne traitions pas réellement des techniques orientées objets dans ce cours, nous avons déjà utilisé des éléments de la bibliothèque standard C++ (*vector* par exemple). Nous serons amenés dans la suite à parler de plusieurs classes mises à disposition par ces bibliothèques. La majorité de ces classes sont génériques et fournissent des fonctions également génériques.

Signalons simplement qu'une classe générique s'instancie de manière explicite comme une fonction générique, Nous l'avons d'ailleurs déjà fait! Pour nous et pour l'instant cela se traduit essentiellement par spécifier les paramètres lors de la déclaration des variables de la classe. Par exemple pour obtenir un *vecteur* initialement vide d'entiers:

```
vector<int> vecteur;
```

Le plus souvent le ou les paramètre(s) correspondent à des types! Toutefois dans le cas des classes, un paramètre générique peut aussi correspondre à une valeur (grandeur) d'un type approprié. Ce sera le cas par exemple lorsque nous traiterons de la classe *bitset* qui permet de manipuler des ensembles de bits et à laquelle il faut spécifier le nombre de ces bits. Nous déclarerons alors des variables du genre:

```
bitset<16> flag;
```

pour obtenir un objet de 16 bits utilisables individuellement ou globalement suivant les

fonctionnalités utilisées.

Bibliothèque héritée de C

Introduction

Dans ce chapitre nous allons présenter brièvement quelques éléments de la bibliothèque hérités de C. Ils nécessitent l'inclusion d'un fichier *cxxx* pour C++ ou du fichier correspondant *xxx.h* pour C.

Exemple: #include <cmath> // En C++
#include <math.h> // En C

Voici tout d'abord la liste complète des éléments de bibliothèque C utilisables en C++:

cassert	**cctype**	**cerrno**	**cfloat**	**ciso646**
climits	**clocale**	**cmath**	**csetjmp**	**csignal**
cstdarg	**cstddef**	**cstdio**	**cstdlib**	**cstring**
ctime	**cwchar**	**cwtype**		

Nous décrirons partiellement certains d'entre eux sans pour autant vouloir couvrir l'ensemble de la matière. Quelques-uns, déjà abordés dans d'autres chapitres ne seront pas repris ici, ou alors partiellement pour introduire de nouvelles informations.

Donnons encore au préalable la liste des éléments de la bibliothèque purement C++ donc non utilisables en C. Certains d'entre eux seront décrits dans des chapitres ultérieurs:

algorithm	**bitset**	**complex**	**deque**	**exception**
fstream	**functional**	**iomanip**	**ios**	**iosfwd**
iostream	**istream**	**iterator**	**limits**	**list**
locale	**map**	**memory**	**new**	**numeric**
ostream	**queue**	**set**	**sstream**	**stack**
stdexcept	**streambuf**	**string**	**typeinfo**	**utility**
valarray	**vector**			

❑ *Commande système: cstdlib*

Depuis le début de nos exemples nous utilisons la commande:

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

pour terminer l'exécution de nos programmes sans que la fenêtre se referme automatiquement. Nous n'allons pas revenir là-dessus, le but ici consistant simplement à formaliser une fois cette utilisation.

Un programme peut exécuter une commande système, donc entre autres exécuter un programme quelconque.

Pour ceci, *cstdlib* (*stdlib.h*) met à disposition:

```
int system ( const char * commande );
```

Le paramètre contient l'adresse d'une chaîne de caractères qui représente la commande système (valable dans l'environnement courant!) qui sera exécutée. La valeur de retour (un entier) dépend de l'environnement.

❑ *Nombre variable de paramètres: cstdarg*

Nous avions dit, sans en décrire les mécanismes, qu'il était possible de définir et d'utiliser des fonctions possédant un nombre variable de paramètres, comme le permettent les fonctions *scanf*, *printf* et d'autres de la bibliothèque. Précisons maintenant cette possibilité.

Tout d'abord rappelons que le prototype d'une telle fonction prend la forme suivante:

```
int maximum ( int nombre, ... );
```

Les "…" indiquant qu'il peut y avoir d'autres paramètres, mais on en fixe ni le nombre ni le type, donc le compilateur ne réalise aucun contrôle sur eux (il ne le peut pas!). Une telle fonction doit toujours avoir au moins un paramètre fixe de type connu, qui s'utilise(nt) comme nous en avons l'habitude.

Commençons par donner un petit exemple, une fonction livrant la plus grande valeur parmi ses paramètres de type **int**:

```
/*
   Demonstration de l'utilisation d'une fonction a un
    nombre variable de paramètres
    PARAMETRESVARIABLES.CPP
*/
#include <cstdarg>
#include <cstdlib>
#include <iostream>
using namespace std;
int maximum ( int nombre, ... ); // prototype!

int main ( )
{
  cout << "Fonction avec nombre de parametres variable\n\n";
  cout << "Maximum de 1, 7, 5, 2 est: "
       << maximum (  4, 1, 7, 5, 2 ) << endl;
  cout << "Maximum de 9, 8, 7 est: "
       << maximum (  3, 9, 8, 7 )  << endl;
  cout << "Fin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

```
/*  Fonction livrant comme resultat la plus grande
      valeur de ses parametres variables
*/
int maximum ( int nombre, ... )
{
  int max, courant;
  va_list param;  // Pour gerer la liste de parametres variables

  va_start ( param, nombre ); // Initialise le processus
  max = va_arg ( param, int ); nombre--;
  /* Traiter tous les parametres */
  while ( nombre-- )
  {  courant = va_arg ( param, int );
     max = max < courant ? courant : max;
  }
  va_end ( param );
  return max;
}
```

Qui donne comme résultats:

```
Fonction avec nombre de parametres variable

Maximum de 1, 7, 5, 2 est: 7
Maximum de 9, 8, 7 est: 9
Fin du programme...Appuyez sur une touche pour continuer...
```

La définition des outils nécessaires (type et macros) se trouve dans *cstdarg (stdarg.h)*, qu'il faut donc inclure dans le fichier source.

On y trouve tout d'abord la définition du type *va_list*; il permet d'accéder à la liste de paramètres. Nous devons dans le sous-programme déclarer une variable de ce type:

```
va_list param;  // Dans notre exemple
```

Le processus de récupération des paramètres de la liste variable[1] s'initialise par l'appel de la macro : *va_start* qui a 2 paramètres :

- Le premier: la variable de type *va_list* déclarée dans notre fonction. C'est elle en fait qui va être initialisée.
- Le deuxième: le nom du dernier paramètre fixe (le premier paramètre s'il n'y en a qu'un!). En fait cela permet de pointer sur le paramètre suivant, le premier de la liste de paramètres variables.

[1] Pas pour le ou les premiers paramètres fixes qui eux s'utilisent de manière usuelle!

Dans notre exemple cela correspond à:

```
va_start ( param, nombre ); // Initialise le processus
```

Ensuite, et c'est là le cœur du traitement proprement dit, à chaque "appel" de la macro *va_arg* nous obtenons la valeur du paramètre suivant. Cette macro a 2 paramètres:

- Le premier: la variable de type *va_list* initialisée par *va_start* et éventuellement déjà modifiée par des appels précédents à *va_arg* (mais en aucun cas modifiée par un autre mécanisme!).
- Le deuxième: le nom du type du paramètre qui va être fourni. Ceci signifie que d'une manière ou d'une autre ce type doit être connu. Avec notre programme, ce n'est pas trop compliqué puisqu'ils sont tous de type **int**.

Dans notre exemple ce traitement est réalisé dans la boucle par:

```
courant = va_arg ( param, int );
```

Il faut bien comprendre qu'à chaque appel c'est le paramètre suivant qui est traité!

Oui, mais combien de fois devons-nous faire cette boucle (plus généralement appeler la macro *va_arg*)? Nous nous sommes basés dans notre exemple sur un premier paramètre nous permettant de déterminer combien d'éléments nous avions à traiter. Ce mécanisme n'est de loin pas le seul possible. Nous pouvons utiliser le principe de la sentinelle, à savoir: mettre en dernière position une valeur dont on sait qu'elle n'apparaît pas autre part dans la liste. Autre possibilité, celle des fonctions du genre *printf*, qui utilisent le contenu du premier paramètre (plus précisément les informations de formatage de la chaîne) pour déterminer le nombre et le type des autres paramètres!

Ce qu'il faut bien comprendre c'est que l'appelant (l'utilisateur de la fonction) aura quelque chose à faire pour que l'appelé puisse déterminer le dernier paramètre de la liste et peut-être même des choses pour transmettre la nature (le type) des paramètres.

Phase finale et obligatoire du traitement, l'appel de la macro *va_end*, qui n'a qu'un seul paramètre: l'objet de type *va_list*. Si nous ne faisons pas cet appel, le comportement du programme devient indéterminé!

Pour notre fonction *maximum*, en partant du principe qu'il y aura toujours au moins un paramètre dans la liste des valeurs parmi lesquelles on recherche le maximum (ce qui semble tout à fait raisonnable comme hypothèse!) nous pouvons même, moyennant quelques petites adaptions, rendre la fonction générique. Ce qui nous donnerait le code suivant:

```
/*  Fonction generique livrant comme resultat la plus grande
      valeur de ses parametres variables
*/
template <class T> T maximum ( int nombre, T p1, ... )
{
  T max = p1, courant;
  va_list param;  // Pour gerer la liste de parametres variables

  va_start ( param, p1 ); // Initialise le processus
  /* Traiter tous les parametres */
  while ( nombre-- )
  {  courant = va_arg ( param, T );
     max = max < courant ? courant : max;
  }
  va_end ( param );
  return max;
}
```

Les modifications:

- Le deuxième paramètre fixe (la première des valeurs à comparer) est du type de la généricité: *T*.
- Partout où nous utilisions le type **int**, sauf pour le premier paramètre qui représente le nombre de valeurs à traiter, nous trouvons maintenant le type générique *T*.
- La variable de travail *max* est directement initialisée lors de sa déclaration avec la première valeur à traiter.

N'oubliez pas que le compilateur ne peut réaliser aucun contrôle sur ces paramètres, ce qui implique un traitement potentiellement dangereux. Dans la mesure du possible nous éviterons une telle situation. Par exemple, s'il s'agit d'un programme C++ et non pas C, et que de plus nous savons que le nombre de paramètres ne dépassera pas 3, nous pouvons envisager d'utiliser des surcharges de la fonction.

❑ *Paramètres du programme*

Bien que cela ne fasse pas directement partie de la bibliothèque nous allons décrire ici, pour son lien avec ce qui précède, une possibilité qui permet de récupérer des paramètres transmis au programme lors de son appel. Ce mécanisme peut avoir des comportements partiellement variables en fonction de l'environnement de travail.

Dans les conditions qui nous intéressent la ligne d'en-tête d'un programme prendra la forme:

```
int main ( int argc, char *argv [] )
```

Le premier paramètre de type **int** contient comme valeur le nombre de chaînes disponibles dans le deuxième paramètre qui est un tableau de pointeurs sur des chaînes dont la taille correspond au nombre de paramètres transmis. Plus précisément *argv [0]* contient le nom du programme (ou éventuellement une chaîne vide si l'environnement n'est pas capable de le fournir), les éléments suivants: *argv [1]*, *argv [2]*, … représentent les arguments s'ils existent (un par mot).

Voici un petit exemple purement formel illustrant l'utilisation de ce mécanisme:

```
/*
  Programme recuperant ses parametres
    ARGCARGV.cpp
*/
#include <cstdlib>
#include <iostream>
using namespace std;
int  main ( int argc, char *argv []  )
{
  int numero = 1;
  cout << "Demonstration de \"argc/argv\":\n\n";
  cout << "Nom du programme: " << *argv << endl;
  while ( --argc > 0 )
    cout << "Parametre " << numero++ << " = " << *++argv << endl;

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Pour bien comprendre lançons ce programme dans une fenêtre "DOS":

```
C:\>argcargv p1 p2 3 ... \n
Demonstration de "argc/argv":

Nom du programme: argcargv
Parametre 1 = p1
Parametre 2 = p2
Parametre 3 = 3
Parametre 4 = ...
Parametre 5 = \n
Appuyez sur une touche pour continuer...
```

❑ *Caractéristiques des entiers et des réels*

De nombreux points varient d'un système à l'autre soit par leur comportement, soit par les valeurs que peuvent prendre des objets d'un type donné.

Nombre de ces caractéristiques font l'objet d'une définition spécifique dans les fichiers:

- *climits* (*limits.h*): essentiellement des grandeurs propres aux différents types entiers
- *cfloat* (*float.h*) : essentiellement des grandeurs propres aux différents types flottants

Nous vous conseillons donc de lire attentivement le contenu de ces fichiers ou tout au moins de regarder les résultats des 2 programmes ci-dessous qui vous permettent de connaître les caractéristiques essentielles valables dans notre environnement. Ces programmes très simples peuvent d'ailleurs vous être utiles en les faisant tourner dans un nouvel environnement sur lequel vous seriez amené à travailler.

Tout d'abord les caractéristiques principales données par *climits*:

```
/*
   Programme exemple: Demontrant certaines limites propres
    a l'environnement de travail
   LIMITES.cpp
*/
#include <climits>  // Valeurs dépendantes de l'implementation
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  cout << "\n\tGrandeurs en octets des types de base\n\n";
  cout << "sizeof(char)\t\t= "      << sizeof(char)        << endl;
  cout << "sizeof(short)\t\t= "     << sizeof(short)       << endl;
  cout << "sizeof(int)\t\t= "       << sizeof(int)         << endl;
  cout << "sizeof(long)\t\t= "      << sizeof(long)        << endl;
  cout << "sizeof(float)\t\t= "     << sizeof(float)       << endl;
  cout << "sizeof(double)\t\t= "    << sizeof(double)      << endl;
  cout << "sizeof(long double)\t= " << sizeof(long double) << endl;

  cout << "\n\tComportement de l'operateur modulo\n\n" << endl;
  cout << " 12 %  5\t= " << 12  % 5  << endl;
  cout << " 12 % -5\t= " << 12  % -5 << endl;
  cout << "-12 %  5\t= " << -12 % 5  << endl;
  cout << "-12 % -5\t= " << -12 % -5 << endl;

  cout << "\n\tCaracteristiques fixees par climits (limits.h)\n\n";
  cout << "CHAR_BIT:   nombre de bits d'un char\t\t= "
       << CHAR_BIT << endl;
  cout << "SCHAR_MIN:  valeur minimale d'un signed char\t= "
       << SCHAR_MIN << endl;
  cout << "SCHAR_MAX:  valeur maximale d'un signed char\t= "
       << SCHAR_MAX << endl;
  cout << "UCHAR_MAX:  valeur maximale d'un unsigned char\t= "
       << UCHAR_MAX << endl;
  cout << "CHAR_MIN:   valeur minimale d'un char\t\t= "
       << CHAR_MIN << endl;
  cout << "CHAR_MAX:   valeur maximale d'un char\t\t= "
       << CHAR_MAX << endl;
  cout << "MB_LEN_MAX: max. # octets d'un char multiple\t= "
       << MB_LEN_MAX << endl;
  cout << "SHRT_MIN:   valeur minimale d'un signed short\t= "
       << SHRT_MIN << endl;
  cout << "SHRT_MAX:   valeur maximale d'un signed short\t= "
       << SHRT_MAX << endl;
  cout << "USHRT_MAX:  valeur maximale d'un unsigned short\t= "
       << USHRT_MAX << endl;
```

```
 cout << "INT_MIN:    valeur minimale d'un int \t\t= "
      << INT_MIN << endl;
  cout << "INT_MAX:   valeur maximale d'un signed int\t= "
      << INT_MAX << endl;

  cout << "UINT_MAX:   valeur maximale d'un unsigned int\t= "
      << UINT_MAX << endl;
  cout << "LONG_MIN:   valeur minimale d'un signed long\t= "
      << LONG_MIN << endl;
  cout << "LONG_MAX:   valeur maximale d'un signed long\t= "
      << LONG_MAX << endl;
  cout << "ULONG_MAX:  valeur maximale d'un unsigned long\t= "
      << ULONG_MAX << endl;
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Son exécution donne les résultats suivants dans notre environnement:

```
        Grandeurs en octets des types de base

sizeof(char)            = 1
sizeof(short)           = 2
sizeof(int)             = 4
sizeof(long)            = 4
sizeof(float)           = 4
sizeof(double)          = 8
sizeof(long double)     = 12

        Comportement de l'operateur modulo


 12 %%  5       = 2
 12 %% -5       = 2
-12 %%  5       = -2
-12 %% -5       = -2

        Caracteristiques fixees par climits (limits.h)

CHAR_BIT:   nombre de bits d'un char            = 8
SCHAR_MIN:  valeur minimale d'un signed char    = -128
SCHAR_MAX:  valeur maximale d'un signed char    = 127
UCHAR_MAX:  valeur maximale d'un unsigned char  = 255
CHAR_MIN:   valeur minimale d'un char           = -128
CHAR_MAX:   valeur maximale d'un char           = 127
MB_LEN_MAX: max. # octets d'un char multiple    = 2
SHRT_MIN:   valeur minimale d'un signed short   = -32768
SHRT_MAX:   valeur maximale d'un signed short   = 32767
```

```
USHRT_MAX:  valeur maximale d'un unsigned short = 65535
INT_MIN:    valeur minimale d'un int            = -2147483648
INT_MAX:    valeur maximale d'un signed int     = 2147483647
UINT_MAX:   valeur maximale d'un unsigned int   = 4294967295
LONG_MIN:   valeur minimale d'un signed long    = -2147483648
LONG_MAX:   valeur maximale d'un signed long    = 2147483647
ULONG_MAX:  valeur maximale d'un unsigned long  = 4294967295
Appuyez sur une touche pour continuer...
```

Relevez aussi le nom donné à ces identificateurs!

Maintenant les caractéristiques obtenues de *cfloat (float.h)*:

```
/*
   Programme exemple: Demontrant certaines caracteristiques
    des Float
   FLOAT.cpp
*/
#include <cfloat>  // Valeurs dépendantes de l'implementation
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  cout << "Caracteristiques des types float\n\n";
  cout << "\nBase utilisee: FLT_RADIX\t\t= " << FLT_RADIX << endl;

  cout << "\nTaille en bits de la mantisse pour:" << endl;
  cout << "float: \t\tFLT_MANT_DIG\t= "  << FLT_MANT_DIG
       << " ==> " << FLT_DIG << " chiffres significatifs" << endl;
  cout << "double: \tDBL_MANT_DIG\t= " << DBL_MANT_DIG
       << " ==> " << DBL_DIG << " chiffres significatifs" << endl;
  cout << "long double: \tLDBL_MANT_DIG\t= " << LDBL_MANT_DIG
       << " ==> " << LDBL_DIG << " chiffres significatifs" << endl;

  cout << "\nExposant en puissance de 2 (MAX/MIN) pour:" << endl;
  cout << "float: \t\tFLT_MAX_EXP\t= "       << FLT_MAX_EXP  << endl;
  cout << "float: \t\tFLT_MIN_EXP\t= "       << FLT_MIN_EXP  << endl;
  cout << "double: \tDBL_MAX_EXP\t= "        << DBL_MAX_EXP  << endl;
  cout << "double: \tDBL_MIN_EXP\t= "        << DBL_MIN_EXP  << endl;
  cout << "long double: \tLDBL_MAX_EXP\t= "  << LDBL_MAX_EXP << endl;
  cout << "long double: \tLDBL_MIN_EXP\t= "  << LDBL_MIN_EXP << endl;
  cout << "\nCe qui represente en puissance de 10 (MAX/MIN) pour:"
       << endl;
  cout << "float: \t\tFLT_MAX_10_EXP\t= "      << FLT_MAX_10_EXP
       << endl;
  cout << "float: \t\tFLT_MIN_10_EXP\t= "      << FLT_MIN_10_EXP
       << endl;
```

```
  cout << "double: \tDBL_MAX_10_EXP\t= "       << DBL_MAX_10_EXP
       << endl;
  cout << "double: \tDBL_MIN_10_EXP\t= "       << DBL_MIN_10_EXP
       << endl;
  cout << "long double: \tLDBL_MAX_10_EXP\t= " << LDBL_MAX_10_EXP
       << endl;
  cout << "long double: \tLDBL_MIN_10_EXP\t= " << LDBL_MIN_10_EXP
       << endl;

  cout << "\nLa plus grande, respectivement petite, valeur pour:\n";
  cout << "float: \t\tFLT_MAX\t = "     << FLT_MAX
       << "\t FLT_MIN  = "              << FLT_MIN  << endl;
  cout << "double: \tDBL_MAX\t =  "     << DBL_MAX
       << " DBL_MIN  = "                << DBL_MIN  << endl;
  cout << "long double: \tLDBL_MAX = "  << LDBL_MAX
       << "\t LDBL_MIN = "              << LDBL_MIN << endl;

  cout << "\nEcart entre 1 et la valeur suivante pour:" << endl;
  cout << "float: \t\tFLT_EPSILON  = "      << FLT_EPSILON  << endl;
  cout << "double: \tDBL_EPSILON  = "       << DBL_EPSILON  << endl;
  cout << "long double: \tLDBL_EPSILON = "  << LDBL_EPSILON << endl;

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Son exécution nous fournit les résultats suivants dans notre environnement:

```
Caracteristiques des types float


Base utilisee: FLT_RADIX               = 2

Taille en bits de la mantisse pour:
float:          FLT_MANT_DIG    = 24 ==> 6 chiffres significatifs
double:         DBL_MANT_DIG    = 53 ==> 15 chiffres significatifs
long double:    LDBL_MANT_DIG   = 64 ==> 18 chiffres significatifs

Exposant en puissance de 2 (MAX/MIN) pour:
float:          FLT_MAX_EXP     = 128
float:          FLT_MIN_EXP     = -125
double:         DBL_MAX_EXP     = 1024
double:         DBL_MIN_EXP     = -1021
long double:    LDBL_MAX_EXP    = 16384
long double:    LDBL_MIN_EXP    = -16381

Ce qui represente en puissance de 10 (MAX/MIN) pour:
float:          FLT_MAX_10_EXP  = 38
float:          FLT_MIN_10_EXP  = -37
```

```
double:         DBL_MAX_10_EXP  = 308
double:         DBL_MIN_10_EXP  = -307
long double:    LDBL_MAX_10_EXP = 4932
long double:    LDBL_MIN_10_EXP = -4931

La plus grande, respectivement petite, valeur pour:
float:          FLT_MAX  = 3.40282e+038  FLT_MIN  = 1.17549e-038
double:         DBL_MAX  =  1.79769e+308 DBL_MIN  = 2.22507e-308
long double:    LDBL_MAX = 1.#INF        LDBL_MIN = 0

Ecart entre 1 et la valeur suivante pour:
float:          FLT_EPSILON  = 1.19209e-007
double:         DBL_EPSILON  = 2.22045e-016
long double:    LDBL_EPSILON = 1.0842e-019
Appuyez sur une touche pour continuer...
```

❑ *Génération de nombres aléatoires: cstdlib*

La génération de nombres aléatoires ne pose aucun problème particulier. Elle se base sur 2 fonctions définies dans *cstdlib* (*stdlib.h*) qu'il faut inclure dans votre programme.

Il s'agit de:

```
void srand ( unsigned int seed );
```

Initialise la série pseudo aléatoire générée. Chaque fois que nous réinitialisons le système avec la même valeur, nous obtenons la même série (il s'agit de valeurs pseudo aléatoires!). Si l'on n'initialise pas la série, tout se passe comme si l'opération avait été faite avec la valeur 1.[1]

```
int rand ( void );
```

Génère un entier pseudo aléatoire compris entre 0 et *RAND_MAX*, constante également définie dans *cstdlib*. Elle peut changer de valeur d'un environnement à l'autre, mais doit être supérieure ou égale à 32767.

Voici un exemple d'utilisation, qui simule le jet d'un dé. Les résultats affichés pour chaque face nous permettent grossièrement de contrôler si la distribution des valeurs semble uniforme:

[1] Pour une meilleure initialisation, se référer à l'utilisation de *time*, présnetée plus loin dans ce chapitre.

```
/*
  But       : Demonstration de la generation de nombres aleatoires
                  simulation d'un lancement de de
      DE.cpp
*/
#include <cstdlib>
#include <iostream>
using namespace std;
#define NOMBRE_DE_FACES 6

int main ( )
{
  int nombreDeSimulations;
  static int leDe [NOMBRE_DE_FACES];
  //  srand ( 17 );  // A titre d'exemple!!!
  cout << "Simulation d'un jeu de de\n\n";
  cout << "Combien de jets a simuler: ";
  cin >> nombreDeSimulations;

  /* Realiser toutes les simulations demandees */
  for ( int i = 1; i <= nombreDeSimulations; i++ )
    leDe [ rand () % NOMBRE_DE_FACES ]++;

  /* Afficher les resultats */
  for ( int i = 1; i <= NOMBRE_DE_FACES; i++ )
    cout << "La face " << i << " est apparue " << leDe [ i - 1 ]
         << " fois\n";
  cout << "\n\nPour info RAND_MAX = " << RAND_MAX << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Une exécution:

```
Simulation d'un jeu de de

Combien de jets a simuler: 100000
La face 1 est apparue 16628 fois
La face 2 est apparue 16719 fois
La face 3 est apparue 16458 fois
La face 4 est apparue 16752 fois
La face 5 est apparue 16769 fois
La face 6 est apparue 16674 fois

Pour info RAND_MAX = 32767

Fin du programme...Appuyez sur une touche pour continuer...
```

La bibliothèque mathématique: cmath

Les fonctions de ce groupe sont définies dans le fichier *cmath* (*math.h*). Nous en avons déjà utilisé quelques-unes. Dans l'ensemble, leur définition et utilisation ne posent aucun problème aussi nous ne les décrirons que brièvement!

Nous décrirons ci-dessous les fonctions travaillant sur des objets de type **double** qui livrent des résultats du même type. Nous disposons des même fonctionnalités sur des objets de type **float** en ajoutant la lettre "f" à la fin du nom de chacune des fonctions décrites et sur des objets de type **long double** en ajoutant la lettre "l".

Exemple : la fonction *frexp* pour les **double** devient *frexpf* pour les **float** et *frexpl* pour les **long double**.

Voici ces fonctions:

Sur les réels

`double acos ( double x );` L'arc cosinus.

`double asin ( double x );` L'arc sinus.

`double atan( double x );` L'arc tangente : une valeur appartenant à l'intervalle [-π/2, π/2].

`double atan2 ( double x, double y );` L'arc tangente de y / x : une valeur appartenant à l'intervalle [-π, π].

`double ceil ( double x );` Arrondi entier par excès (partie entière sous forme d'un réel double). Attention toutefois à la représentation interne approchée des valeurs réelles; si 3.000000000000001 donnera vraisemblablement 4, 3.0000000000000001 donnera peut être 3, dépend de l'environnement!

`double cos ( double x )` Le cosinus.

`double cosh ( double x );` Le cosinus hyperbolique.

double exp (**double** x); L'exponentielle : e^x.

double fabs (**double** x); La valeur absolue.

double floor (**double** x); Arrondi entier par troncature; avec le même genre de remarque que pour *ceil*: si 3.999999999999999 donnera vraisemblablement 3, une valeur théorique de 3.9999999999999999 elle fournira peut être 4, dépend de l'environnement!

double fmod (**double** x, **double** y); Livre l'équivalent du reste de la division entière, mais pour des réels. Vaut x-i*y avec i tel que x-i*y = signe(x)*a et 0<=a<|y|. Le signe du résultat correspond à celui du premier opérande. Si le deuxième opérande vaut 0, le résultat est indéterminé.

double frexp (**double** x, **int** *exp); Elle livre la mantisse normalisée comme résultat et l'exposant (puissance de 2) dans le deuxième paramètre. Cela permet de connaître la représentation interne de valeurs réelles en double précision.

double ldexp (**double** x, **int** exp); Livre comme résultat $x*2^{exp}$.

double log (**double** x); Le logarithme naturel.

double log10 (**double** x); Le logarithme en base 10.

double modf (**double** x, **double** *ip); Livre comme résultat la partie décimale de x et dans le deuxième paramètre sa partie entière ; les 2 valeurs ont le même signe que x. A ne pas confondre avec *fmod*!!

double pow (**double** x, **double** y); Livre comme résultat x puissance y.

double sin (**double** x); Le sinus.

double sinh (**double** x); Le sinus hyperbolique.

double sqrt (**double** x); La racine carrée.

double tan (**double** x); La tangente.

double tanh (**double** x); La tangente hyperbolique.

Comme indiqué, nous n'avons pas décrit toutes les fonctions de la bibliothèque mathématique. Signalons encore simplement, mais cela devrait être une évidence en informatique, que les fonctions trigonométriques (sin, cos, …) travaillent toutes avec des valeurs exprimées en radians.

Un exemple qui nous amènera une petite réflexion que les débutants ont parfois un peu de peine à assimiler:

```
/*
  Utilisation de cmath
  MATH.cpp
*/
#include <cmath>
#include <iostream>
#include <iomanip>
using namespace std;

int main ( )
{
  cout << "Utilisation de CMATH\n\n";
  cout << setprecision (12) << fixed;
  for ( int i = 1; i <= 10; i++ )
    cout << i << '\t' << setw (15) << sqrtf ( float ( i ) )
         << setw (15) << sqrt ( double ( i ) ) << endl;

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Qui nous donne lors de l'exécution l'affichage (peu raisonnable!) suivant:

```
Utilisation de CMATH

1       1.000000000000 1.000000000000
2       1.414213538170 1.414213562373
3       1.732050776482 1.732050807569
4       2.000000000000 2.000000000000
5       2.236068010330 2.236067977500
6       2.449489831924 2.449489742783
7       2.645751237869 2.645751311065
8       2.828427076340 2.828427124746
9       3.000000000000 3.000000000000
10      3.162277698517 3.162277660168
Appuyez sur une touche pour continuer...
```

Pourquoi avons-nous dit "*l'affichage peu raisonnable*"? Simplement parce qu'il ne sert à rien d'afficher des valeurs avec une précision de 12 chiffres alors que le type de la variable n'en permet que 6! Toutefois si nous le demandons, le code généré permettra de le faire mais le résultat n'aura aucun sens. Il est même trompeur, donc dangereux, puisque l'on peut croire que ces chiffres sont effectivement significatifs!

Toutefois faisons tout de même attention à divers aspects:

- L'affichage par défaut qui parfois ne tient pas compte des possibilités effectives du type traité.
- Le fait que les calculs se font effectivement avec une précision correspondant au type **double**.

Pour illustrer ce dernier point, modifions un peu le programme ci-dessus et observons les conséquences sur les résultats affichés:

```
/*
  Utilisation de cmath
  MATH2.cpp
*/
#include <cmath>
#include <iostream>
#include <iomanip>
using namespace std;

int main ( )
{
  cout << "Utilisation de CMATH\n\n";
  cout << setprecision (12) << fixed;
  for ( int i = 1; i <= 10; i++ )
    cout << i << '\t' << setw (15) << (double) sqrtf ( float ( i ) )
         << setw (15) << sqrt ( double ( i ) ) << endl;

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Cette fois les résultats affichés sont les suivants:

```
Utilisation de CMATH

1       1.000000000000 1.000000000000
2       1.414213562373 1.414213562373
3       1.732050807569 1.732050807569
4       2.000000000000 2.000000000000
5       2.236067977500 2.236067977500
6       2.449489742783 2.449489742783
7       2.645751311065 2.645751311065
8       2.828427124746 2.828427124746
9       3.000000000000 3.000000000000
10      3.162277660168 3.162277660168
Appuyez sur une touche pour continuer...
```

Comparez les résultats de ces 2 exécutions et tirez-en les conclusions qui s'imposent!

Sur les entiers: cstdlib

Puisque nous sommes dans les fonctions mathématiques, relevons aussi qu'il existe quelques fonctions travaillant sur les entiers, mais là nous quittons *cmath* (*math.h*) pour revenir à *cstdlib* (*stdlib.h*) dont nous avons déjà parlé.

Parmi elles, 2 permettent d'obtenir la valeur absolue respectivement d'un **int** et d'un **long**:

```
int  abs  ( int i );
long labs ( long j );
```

Deux autres apparaissent plus particulières:

```
div_t div ( int numer, int denom );
```

Calcule le quotient et le reste de la division entière de *numer* par *denom*; *div_t* est un type structure comportant 2 champs **int** et défini dans *cstdlib* (*stdlib.h*) sous la forme:

```
typedef struct { int quot, rem; } div_t;
```

et la fonction:

```
ldiv_t ldiv ( long numer, long denom );
```

Calcule le quotient et le reste de la division entière de *numer* par *denom*; *ldiv_t* est un type structure comportant 2 champs de type **long int** correspondant à la définition:

```
typedef struct { long quot, rem; } ldiv_t;
```

Dans ce groupe viennent aussi les fonctions de génération de nombres aléatoires dont nous avons déjà parlé!

❑ *Tri et recherche: cstdlib*

Restons dans *cstdlib* (*stdlib.h*) (qui met des outils très variés à disposition) pour décrire l'utilisation de 2 fonctions générales, l'une de tri, l'autre de recherche.

La première, pour le tri:

```
void qsort ( void *array, size_t nmemb, size_t size,
       int ( *compar ) ( const void *, const void * ) );
```

Réalise le tri (quicksort) d'un tableau *array*, comportant *nmemb* éléments, chacun étant de taille *size* (en unités mémoire). Le dernier paramètre (pointeur) fournit la fonction de comparaison; celle-ci doit livrer:

- Une valeur entière négative si son premier paramètre est plus petit que le deuxième.
- 0 si ces 2 paramètres ont une valeur égale.
- Une valeur entière positive si son premier paramètre est plus grand que le deuxième.

C'est une fonction très générale, pouvant s'adapter à tout contexte, donc à un tableau de taille quelconque et possédant des éléments de n'importe quel type. Elle modifie le contenu du tableau puisque c'est le but du tri.

La deuxième, pour la recherche:

```
void *bsearch
  ( const void *key, const void *array, size_t nmemb,
    size_t size, int ( *compar )( const void *, const void * ));
```

Recherche dichotomique de l'élément *key* dans le tableau *array*, comportant *nmemb* éléments, chacun étant de taille *size* (en unités mémoire). Le dernier paramètre (pointeur) fournit la fonction de comparaison avec les mêmes conventions que pour *qsort*. C'est donc une fonction très générale pouvant s'adapter à tout contexte. La fonction livre en retour un pointeur (neutre) contenant l'adresse de l'élément recherché si on l'a trouvé et la valeur *NULL* sinon.

Note: une recherche dichotomique se fait par définition sur un tableau trié; signalons toutefois qu'il n'est pas nécessaire qu'il le soit entièrement, la condition minimale suffisante est que toutes les valeurs plus petites que la clé recherchée se trouve "avant" et par conséquence, toutes les valeurs plus grandes après!

Donnons d'abord un exemple. Nous fournirons ensuite des explications complémentaires. Une fois n'est pas coutume, étant essentiellement dans un chapitre propre à la bibliothèque C, nous donnerons d'abord une version purement C (si on peut dire) de ce programme, puis une adaptation C++ tout en gardant l'utilisation de la bibliothèque de base. Notez toutefois que la version C peut se "compiler en C++" moyennant il est vrai une conversion explicite des types pointeurs, que nous aurions aussi pu réaliser en C mais sans que cela soit une obligation. Dans cet exemple nous allons construire un tableau dont les éléments sont d'un type structure offrant deux champs de type entier. Dans le premier champ nous enregistrons une valeur générée aléatoirement; dans le deuxième nous gardons simplement le numéro d'ordre de génération de la valeur, ou si vous préférez la position dans le tableau (en partant de 1!). Nous affichons le tableau pour en montrer l'état actuel. Ensuite, par la fonction *qsort* nous trions les éléments de ce tableau en fonction de la valeur des premiers membres de la structure et nous affichons à nouveau le tableau pour visualiser le résultat. Finalement, tant que l'utilisateur le désire, nous lui demandons une valeur entière, nous lui indiquons grâce au résultat de la fonction *bsearch* si cette valeur se trouve dans le tableau et si c'est le cas, à quelle position initiale elle se trouvait lors de la génération des valeurs.

```
/*
  But       : Demonstration de l'utilisation des fonctions de
              tri et de recherche dichotomique
      TRIRECHERCHE.C
*/
#include <stdio.h>
#include <stdlib.h>
typedef struct element { int valeur; int position; } t_element;
/* Afficher les valeurs d'un tableau */
void afficher ( t_element tableau [], int nombre );
/* Fonction de comparaison pour qsort et bsearch */
int compare ( const void *v1, const void *v2 );

int main ( void )
{
  int nombreDeValeurs;
  t_element *tableau, *ou;
  int valeur;
  srand ( 17 );  // Initialisation du generateur aleatoire
  printf ( "Tri et recherche\n\n" );
  printf ( "Combien de valeurs a traiter: " );
  scanf  ( "%d", &nombreDeValeurs );
  tableau = calloc ( nombreDeValeurs, sizeof ( t_element ) );
  /* Generer toutes les valeurs */
  for ( int i = 0; i < nombreDeValeurs; i++ )
  { tableau [ i ].valeur   = rand ();
    tableau [ i ].position = i + 1;
  }
  /* Afficher l'etat du tableau */
  printf ( "\n\nTableau avant le tri:\n" );
  afficher ( tableau, nombreDeValeurs );
```

```
  /* Trier le tableau */
  qsort ( tableau, nombreDeValeurs, sizeof ( t_element ), compare );
  /* Afficher l'etat du tableau */
  printf ( "\n\nTableau apres le tri:\n" );
  afficher ( tableau, nombreDeValeurs );

  /* Recherche de valeurs */
  while ( 1 )
  { printf ( "Quelle valeur desirez-vous (0 pour terminer): " );
    scanf ( "%d", &valeur );
    /* Veut-on arreter? */
    if ( !valeur ) break;

    ou = bsearch ( &valeur, tableau, nombreDeValeurs,
                           sizeof ( t_element ), compare );
    /* La valeur se trouve-t-elle dans le tableau? */
    if ( ou )
      /* Oui, afficher sa position initiale */
      printf ( "\nLa valeur existe, introduite a la position: %d\n",
                   (*ou).position );
    else
      printf ( "\nLa valeur n'existe pas\n" );
  }
  printf ( "\nFin du programme..." );
  system ( "pause" );
  return EXIT_SUCCESS;
}

/* Afficher les valeurs d'un tableau */
void afficher ( t_element tableau [], int nombre )
#define NB_VALEUR_LIGNE 4
{
  for ( int i = 0; i < nombre; )
  {  printf ( " %7i, %4i ", tableau [ i ].valeur,
                            tableau [ i ].position );
     /* Faut-il passer a la ligne? */
     if ( !( ++i % NB_VALEUR_LIGNE ) ) printf ( "\n" );
  }
  printf ( "\n" );
}
```

```
/* Fonction de comparaison pour qsort et bsearch */
/*
    Attention, dans notre contexte on sait
     qu'il ne peut pas y avoir debordement
*/
int compare ( const void *v1, const void *v2 )
{
  return (*(t_element *)v1).valeur - (*(t_element *)v2).valeur;
}
```

Les résultats d'une exécution:

```
Tri et recherche

Combien de valeurs a traiter: 22


Tableau avant le tri:
     94   1  26602   2  30017   3  18297   4
  20363   5  13015   6  28509   7  15290   8
  29003   9  24399  10   3339  11  28849  12
  17055  13  19424  14   4588  15  15756  16
   6098  17  11834  18   1351  19  21383  20
  18431  21    155  22


Tableau apres le tri:
      94,    1      155,   22     1351,   19     3339,   11
    4588,   15     6098,   17    11834,   18    13015,    6
   15290,    8    15756,   16    17055,   13    18297,    4
   18431,   21    19424,   14    20363,    5    21383,   20
   24399,   10    26602,    2    28509,    7    28849,   12
   29003,    9    30017,    3
Quelle valeur desirez-vous (0 pour terminer): 6098

La valeur existe, introduite a la position: 17
Quelle valeur desirez-vous (0 pour terminer): 33

La valeur n'existe pas
Quelle valeur desirez-vous (0 pour terminer): 0

Fin du programme...Appuyez sur une touche pour continuer...
```

La fonction de comparaison (dans notre exemple *compare*) possède 2 paramètres qui sont les pointeurs (neutres selon la définition du prototype) sur les éléments à comparer. Ceci explique les conversions de type que nous avons appliquées:

```
return (*(t_element *)v1).valeur - (*(t_element *)v2).valeur;
```

ici nous jouons sur le fait que toutes les valeurs à traiter sont positives, donc que le résultat de la soustraction nous donne directement une valeur raisonnable pour notre fonction.

Notez qu'il suffit simplement d'inverser le résultat de la fonction de comparaison pour que *qsort* livre un tableau trié dans un ordre décroissant au lieu d'un ordre croissant!

Voici maintenant la version C++ du même programme et qui donne évidemment les mêmes résultats:

```
/*
  But       : Demonstration de l'utilisation des fonctions de
              tri et de recherche dichotomique
      TRIRECHERCHE.cpp
*/
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef struct { int valeur; int position; } element;
/* Afficher les valeurs d'un tableau */
void afficher ( element tableau [], int nombre );
/* Fonction de comparaison pour qsort et bsearch */
int compare ( const void *v1, const void *v2 );

int main ( )
{
  int nombreDeValeurs;
  element *tableau, *ou;
  int valeur;

  srand ( 17 );  // Initialisation du generateur aleatoire
  cout << "Tri et recherche\n\n";
  cout << "Combien de valeurs a traiter: ";
  cin >> nombreDeValeurs;

  tableau = new element [ nombreDeValeurs ];
  /* Generer toutes les valeurs */
  for ( int i = 0; i < nombreDeValeurs; i++ )
  { tableau [ i ].valeur   = rand ();
    tableau [ i ].position = i + 1;
  }
```

```
  /* Afficher l'etat du tableau */
  cout << "\n\nTableau avant le tri:\n";
  afficher ( tableau, nombreDeValeurs );

  /* Trier le tableau */
  qsort ( tableau, nombreDeValeurs, sizeof ( element ), compare );

  /* Afficher l'etat du tableau */
  cout << "\n\nTableau apres le tri:\n";
  afficher ( tableau, nombreDeValeurs );

  /* Recherche de valeurs */
  while ( 1 )
  {
    cout << "Quelle valeur desirez-vous (0 pour terminer): ";
    cin >> valeur;
    /* Veut-on arreter? */
    if ( !valeur ) break;

    // Attention: en C++ la conversion explicite est necessaire!
    ou = (element *) bsearch ( &valeur, tableau, nombreDeValeurs,
                               sizeof ( element ), compare );
    /* La valeur se trouve-t-elle dans le tableau? */
    if ( ou )
      /* Oui, afficher sa position initiale */
      cout << "\nLa valeur existe, introduite a la position: "
           << ou->position << endl;
    else
      cout << "\nLa valeur n'existe pas\n";
  }

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

```
/* Afficher les valeurs d'un tableau */
void afficher ( element tableau [], int nombre )
{
  const int NB_VALEUR_LIGNE = 4;
  for ( int i = 0; i < nombre; )
  {
     cout << setw(7) << tableau [ i ].valeur
          << setw(4) << tableau [ i ].position;
     /* Faut-il passer a la ligne? */
     if ( !( ++i % NB_VALEUR_LIGNE ) ) cout << endl;
  }
  cout << endl;
}


/* Fonction de comparaison pour qsort et bsearch */
/*
    Attention, dans notre contexte on sait
     qu'il ne peut pas y avoir debordement
*/
int compare ( const void *v1, const void *v2 )
{
  return ((element *)v1)->valeur - ((element *)v2)->valeur;
}
```

Gestion du temps: ctime

Le fichier *ctime* (*time.h*) contient les définitions nécessaires pour résoudre les problèmes de gestion du temps. Attention toutefois lors de son utilisation, certains comportements peuvent varier en fonction de l'environnement!

Donnons tout d'abord un exemple qui utilise une grande partie de ces possibilités, puis nous en expliquerons les éléments principaux.

```
/*
  But       : Demonstration de l'utilisation des fonctions
              de gestion du temps
      TEMPS.CPP
*/
#include <ctime>
#include <iostream>
using namespace std;
#define LONGUEUR_MAX 80

int main ( )
{
  time_t depart;
  char   affichage [ LONGUEUR_MAX + 1 ];
  struct tm copieDepart;
  int    valeur; // Utilisee juste pour faire durer!

  /* Prendre l'heure de depart */
  time ( &depart );
  cout << "Gestion du temps\n\n";

  /* Afficher date et heure locale */
  cout << "Nous sommes (local): " << ctime ( &depart ) << endl;
  /* Afficher date et heure en temps universel */
  cout << "Nous sommes (universel): "
       << asctime ( gmtime ( &depart ) ) << endl;
```

```
  /* Changement de format mais pas de valeur */
  copieDepart = *localtime ( &depart );
  /* Afficher date et heure, equivalent ctime */
  cout << "Nous sommes: " << asctime ( &copieDepart ) << endl;
  /* Decomposition de la structure "tm" */
  cout << "Les secondes: " << copieDepart.tm_sec << endl;
  cout << "Les minutes: " << copieDepart.tm_min << endl;
  cout << "Les heures: " << copieDepart.tm_hour << endl;
  cout << "Le jour du mois: " << copieDepart.tm_mday << endl;
  cout << "Le mois: " << copieDepart.tm_mon + 1 << endl;
  cout << "L'annee: " << copieDepart.tm_year + 1900 << endl;
  cout << "C'est le jour " << copieDepart.tm_wday + 1
       << " de la semaine\n";
  cout << "...et le jour numero " << copieDepart.tm_yday
       << " de l'annee\n";
  /* Y a-t-il un decalage local? */
  if ( copieDepart.tm_isdst > 0 )
    cout << "Il y a un decalage local!\n";
  else if ( copieDepart.tm_isdst == 0 )
    cout << "Il n'y a pas de decalage local!\n";
  else
    cout << "On ne sait pas s'il y a decalage local!\n";

  /* Les possibilites de mise en forme */
  strftime ( affichage, LONGUEUR_MAX,
      "Jour: %A (%a), mois: %b (%b) annee %y (%Y)\n", &copieDepart );
  cout << affichage << endl;
  strftime ( affichage, LONGUEUR_MAX,
             "Heure: %H (%I - %p)\n", &copieDepart );
  cout << affichage << endl;

  cout << "\n\nDonnez une valeur entiere (faire attendre!): ";
  cin  >> valeur;
  cout << "\nVous avez donne: " << valeur << endl;

  cout << "\nNous travaillons depuis: "
       << difftime ( time ( NULL ), depart )
       << " secondes\n";
  cout << "\nNous avons utilise:" << (float) clock() / CLOCKS_PER_SEC
       << " secondes\n",

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Résultat d'une exécution:

```
Gestion du temps

Nous sommes (local): Sun Dec 09 17:21:13 2007

Nous sommes (universel): Sun Dec 09 16:21:13 2007

Nous sommes: Sun Dec 09 17:21:13 2007

Les secondes: 13
Les minutes: 21
Les heures: 17
Le jour du mois: 9
Le mois: 12
L'annee: 2007
C'est le jour 1 de la semaine
...et le jour numero 342 de l'annee
Il n'y a pas de decalage local!
Jour: Sunday (Sun), mois: Dec (Dec) annee 07 (2007)

Heure: 17 (05 - PM)



Donnez une valeur entiere (faire attendre!): 22

Vous avez donne: 22

Nous travaillons depuis: 20 secondes

Nous avons utilise:20.299 secondes

Fin du programme...Appuyez sur une touche pour continuer...
```

Tout d'abord, en plus du type *size_t*, déjà vu et défini dans d'autres fichiers d'inclusion, *ctime (time.h)* nous met à disposition les types:

- ***time_t***: un type entier, permettant de représenter une durée selon un codage interne.
- ***clock_t***: un autre type entier, permettant de représenter un temps *t* selon un codage interne.

- ***struct tm***: un type structure, permettant de représenter un temps en différentes parties; le type comporte les membres suivants:
  - **int** *tm_sec*: le nombre de secondes (de 0 à 59)
  - **int** *tm_min*: le nombre de minutes (de 0 à 59)
  - **int** *tm_hour*: le nombre d'heures (de 0 à 23)
  - **int** *tm_mday*: le jour du mois (de 1 à 31)
  - **int** *tm_mon*: le numéro d'ordre du mois (de 0 à 11)
  - **int** *tm_year*: l'année (0 correspond à 1900)
  - **int** *tm_wday*: le numéro du jour dans la semaine (0 = dimanche)
  - **int** *tm_yday*: le numéro du jour dans l'année (0 à 365)
  - **int** *tm_isdst*: indique s'il y a un décalage horaire local (<0: information inconnue, =0: il n'y a pas de décalage, >0 il y a un décalage)

Dans notre exemple nous avons démontré ces possibilités par les différents affichages regroupés sous le commentaire */* Decomposition de la structure "tm" */*.

En plus de la constante *NULL* également définie dans d'autres fichiers d'inclusion, *ctime* (*time.h*) met également à disposition la constante ***CLOCKS_PER_SEC*** qui représente le nombre d'unités d'horloge dans une seconde.

Notre programme pour ses besoins déclare une variable du type *time_t* (*depart*) et une autre du type *tm* (*copieDepart*).

Au début de l'exécution du programme nous sauvons les valeurs courantes de l'horloge dans la structure *depart* de type *time_t* par un appel à la fonction:

```
time_t time ( time_t *caltime );
```

qui livre la date et l'heure courante comme résultat, ainsi que dans l'objet désigné par *caltime* si ce pointeur n'est pas *NULL*. Dans notre cas, nous ne faisons rien de particulier du résultat de la fonction.

De ce fait, nous pouvons remplacer cette instruction (*time (&depart)*) par:

```
depart = time ( NULL );
```

Avec la fonction:

```
char *ctime ( const time_t *caltime );
```

qui livre un pointeur sur une chaîne de caractères contenant le résultat de la transformation du temps transmis en paramètre (de type *time_t*), nous affichons la date et l'heure. En fait cette fonction correspond à: *asctime (localtime (timer))* c.f. suite.

La fonction:

```
struc tm *gmtime ( const time_t *timer );
```

transforme un temps calendrier dont la valeur se trouve dans l'objet désigné par le pointeur *timer* en un temps sous forme locale de type *tm*, exprimé en heure universelle. Dans notre exemple nous constatons une heure de décalage, puisque nous utilisons ensuite la fonction:

```
char *asctime ( const struct tm *bdtime );
```

qui transforme la valeur contenue dans la structure pointée par *bdtime* de type *tm* en une chaîne de caractères, pour afficher cette valeur.

Avec l'appel à:

```
struct tm *localtime ( const time_t *caltime );
```

nous faisons une simple transformation de format du type *time_t* au type *tm*, ce qui nous permet d'affecter à notre variable *copieDepart* (de type *tm*) la valeur de *depart* qui, elle, est de type *time_t*.

La fonction:

```
size_t strftime ( char * string, size_t size,
                  const char * format,
                  const struct tm * bdtime );
```

range dans la chaîne désignée par *string* différentes informations relatives au temps enregistré dans la structure désignée par *bdtime* et ceci selon le *format* donné par une chaîne de caractères. Nous pouvons utiliser les différents formats de manière indépendante ou groupée

selon l'ordre que l'on désire. Ces formats correspondent en quelque sorte aux formats de la fonction *printf*. Si la chaîne résultante de toutes les demandes (de tous les formats) est de taille plus petite que *size*, elle est réduite à l'adresse désignée par *string* en étant complétée par un octet nul, la valeur livrée par la fonction est alors la taille de cette chaîne; par contre si la taille de la chaîne résultante est plus grande ou égale à *size*, la fonction livre 0 et le contenu de la chaîne désignée par *string* est alors indéterminé!

Le tableau suivant fournit la liste des différents formats utilisables:

%A	nom du jour
%a	abréviation du nom du jour
%B	nom du mois
%b	abréviation du nom du mois
%c	la date et l'heure au format local
%d	1 à 31: le jour du mois
%H	0 à 23: l'heure sur 24 heures
%I	0 à 12: l'heure sur 12 heures (I = i majuscule!!)
%j	1 à 366: numéro du jour de l'année
%m	1 à 12: numéro du mois de l'année
%M	0 à 59: les minutes
%p	indicateur (AM/PM) pour l'heure sur 12 heures
%S	0 à 59: les secondes
%U	0 à 53: numéro de la semaine dans l'année, la semaine numéro 1 commençant le premier dimanche de l'année
%W	0 à 53: numéro de la semaine dans l'année, la semaine numéro 1 commençant le premier lundi de l'année
%w	0 à 6: numéro du jour dans la semaine, le dimanche correspondant à 0
%X	l'heure en forme locale
%x	la date en forme locale
%Y	l'année sous forme complète (4 chiffres)
%y	l'année sous forme réduite (2 chiffres)
%Z	une indication (forme variable!) spécifiant la zone locale (vide si le système ne peut donner une telle information)
%%	le caractère % (comme pour *printf*!)

Dans notre exemple nous avons illustré l'utilisation des ces possibilités par plusieurs affichages consécutifs.

Toujours dans notre exemple, la demande d'une valeur entière et son affichage ne sont là que pour permettre au temps de s'écouler.

Ensuite nous avons utilisé la fonction:

```
double difftime ( time_t time1, time_t time0 );
```

qui livre comme résultat (exprimé en double précision) le nombre de secondes obtenues de la soustraction du temps *time0* au temps *time1*. Ceci représente dans notre exemple le temps écoulé depuis le lancement du programme. Notez l'appel à *time* dans le programme, avec un paramètre *NULL* pour utiliser directement le résultat dans la soustraction.

Dernière fonction de cette catégorie utilisée dans le programme:

```
clock_t clock ( void );
```

qui livre théoriquement le temps utilisé par le programme, sur une base définie par l'implémentation. Pour obtenir le temps en secondes, il faut diviser le résultat obtenu par: *CLOCKS_PER_SEC*. Nous pouvons constater que dans notre environnement en tous les cas, ce n'est pas le temps CPU utilisé qui est livré, mais simplement le temps "horloge", d'où les valeurs proches des 2 derniers affichages!

Il existe encore une fonction de cette catégorie que nous n'avons pas utilisée et que nous ne détaillerons guère:

```
time_t mktime ( struct tm *bdtime );
```

qui transforme l'heure locale enregistrée dans la structure pointée par *bdtime* en une valeur encodée de type *time_t*. Dans la structure, les plages de validité fixées pour les différents champs n'ont pas besoin d'être respectées, la fonction les mettra automatiquement à jour. De même les champs *tm_wday* et *tm_yday* ne sont pas significatifs en entrée et sont mis à jour en sortie.

Générateurs aléatoires: le retour

Revenons brièvement sur les générateurs de nombres aléatoires. Nous avions constaté que les séries de valeurs générées étaient toujours les mêmes. Cette possibilité s'avère souvent utile afin de pouvoir reproduire le comportement d'un programme. Dans d'autres circonstances cela devient un inconvénient majeur! L'horloge s'utilise souvent dans ce genre de situations afin de permettre une initialisation différenciée du générateur à chaque utilisation. Reprenons à titre d'illustration notre exemple du *dé* et adaptons-le dans ce sens:

```
/*
  But       : Demonstration de la generation de nombres aleatoires
                  simulation d'un lancement de de
      DE2.CPP
*/
#include <cstdlib>
#include <iostream>
#include <ctime>
using namespace std;
#define NOMBRE_DE_FACES 6

int main ( )
{
  int nombreDeSimulations;
  static int leDe [NOMBRE_DE_FACES];

  /* Pour initialiser le generateur */
  srand ( time ( NULL ) );

  cout << "Simulation d'un jeu de de\n\n";
  cout << "Combien de jets a simuler: ";
  cin >> nombreDeSimulations;

  /* Realiser toutes les simulations demandees */
  for ( int i = 1; i <= nombreDeSimulations; i++ )
    leDe [ rand () % NOMBRE_DE_FACES ]++;

  /* Afficher les resultats */
  for ( int i = 1; i <= NOMBRE_DE_FACES; i++ )
    cout << "La face " << i << " est apparue " << leDe [ i - 1 ]
         << " fois\n";

  cout << "\n\nPour info RAND_MAX = " << RAND_MAX << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Assert: cassert

Le but premier de cette bibliothèque consiste à offrir, en phase de développement, un outil d'aide au dépannage. Il doit en partie remplacer d'éventuelles instructions d'impression qui nous permettraient de suivre à la trace notre programme. Pour accéder à l'unique fonctionnalité de cette bibliothèque, une unité de compilation doit inclure le fichier *cassert (assert.h)*.

Une macro est mise à disposition, elle correspond à la définition:

```
void assert ( expression_scalaire );
```

Exemple d'utilisation:

```
assert ( i < 0 );
```

Si l'expression est vraie au moment de l'exécution de *l'assert*, tout se passe comme si cet *assert* n'existait pas; par contre si l'expression est fausse (vaut 0!) le déroulement normal du programme s'arrête, les informations venant __FILE__, __LINE__ et __func__ ainsi qu'éventuellement d'autres spécifiques à l'environnement sont affichées sur la sortie standard des erreurs, ensuite le programme s'arrête brutalement par un appel à la fonction *abort* de *cstdlib (stdlib.h)* qui arrête brutalement et définitivement l'exécution du programme pour redonner le contrôle au système.

Note complémentaire: l'effet des *assert* est annulé si au moment de l'inclusion du fichier *cassert (assert.h)* le symbole *NDEBUG* est défini. Ceci permet donc de générer le code en phase de développement et de supprimer ce code aux différents endroits où il est généré par la définition d'un seul et unique symbole.

Donnons maintenant un petit programme exemple, un peu limite, mais qui montre la manière d'utiliser cette possibilité. Nous reprenons simplement la fonction calculant récursivement les nombres de Fibonacci. Nous y avons remplacé le type du paramètre et du résultat de la fonction (anciennement **unsigned int**) par **int** et si l'utilisateur appelle avec une valeur <0 le *assert* prendra effet. Notez que nous avons mis le #define NDEBUG en commentaire, il n'a donc pour l'instant pas d'effet, mais pourra facilement être modifié!

Voici ce code:

```
/* #define NDEBUG
  Utilisation de assert
  ASSERT.cpp
*/
#include <cassert>
#include <iostream>
using namespace std;
/* Fonction calculant le nombre de Fibonacci numero i  */
int fibo ( int i )
{
  assert ( i >= 0 );  // Arret brutal si i < 0
  /* Cas limite simple? */
  if ( i <= 1 )
    /* Oui, retourner la valeur 1 */
    return 1;
  else
    /* Non, appels recursifs */
    return fibo ( i - 1 ) + fibo ( i - 2 );
} /* fibo */

int main ( )
{
  int valeur; // La valeur donnee par l'utilisateur
  cout << "Les nombres de Fibonacci\n\n";
  cout << "Pour quelle valeur voulez-vous le nombre de Fibonacci: ";
  cin >> valeur;
  cout << "fibo : " << valeur << "= " << fibo ( valeur ) << endl;
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Exemples d'exécution:

```
Les nombres de Fibonacci

Pour quelle valeur voulez-vous le nombre de Fibonacci: -3
Assertion failed: i >= 0, file
C:\JMT\Polycop\CPP\Partiel\BibliothequeC\Assert.cpp, line 14

This application has requested the Runtime to terminate it in an
unusual way.
Please contact the application's support team for more information.
```

Bien sûr si l'on donne une valeur positive, tout se passe correctement sans problème:

```
Les nombres de Fibonacci

Pour quelle valeur voulez-vous le nombre de Fibonacci: 4
fibo : 4= 5
Appuyez sur une touche pour continuer...
```

Les autres bibliothèques C

Nous allons donner ici une très briève indication sur les bibliothèques non encore présentées.

errno.h / cerrno

Définit essentiellement *errno* censée représenter un code d'erreur. Toutefois une seule valeur (variable ou macro) pour gérer toutes les erreurs semble un peu léger, d'autant plus que les différents composants de la bibliothèque ne la maintiennent pas systématiquement à jour!

Elle fournit aussi 2 constantes représentant les valeurs que peut prendre *errno*, à savoir:

- *EDOM* lorsqu'un paramètre prend une valeur hors de sa plage de validité!
- *ERANGE* lorsqu'un résultat dépasse la capacité du type utilisé!

locale.h / clocale

Permet de fixer certaines caractéristiques nationales utilisées par différentes fonctionnalités de la bibliothèque. Il s'agit entre autres:

- Du jeu de caractères étendus utilisé.
- L'ordre des caractères.
- La manière d'écrire les nombres pour les lectures et les écritures.
- La manière de formater les valeurs monétaires.

On y trouve entre autres la fonction (non détaillée ici):

```
char *setlocale ( int cat, const char *locale );
```

iso646.h / ciso646

Permet d'influencer le jeu de caractères utilisé!

setjmp.h / csetjmp

Les éléments ci-dessous ne sont à utiliser qu'avec une très grande prudence, et souvent dans un contexte de gestion des interruptions. Le type *jmp_buf* est un tableau d'éléments pouvant contenir les informations nécessaires à un contexte d'exécution, notion évidemment variable en fonction du matériel et du logiciel système.

```
int setjmp ( jmp_buf env );
```

Permet de sauver dans *env* l'environnement courant pour une future utilisation en conjonction avec *longjmp*.

```
void longjmp ( jmp_buf env, int val );
```

Réutilise le contexte sauvé par *setjmp*.

signal.h / csignal

L'objectif de cette bibliothèque consiste à offrir un mécanisme de bas niveau permettant d'implémenter un traitement d'exceptions/interruptions. Toutefois c'est à vous de le réaliser, les fonctions ci-dessous vous aidant dans cette tâche:

```
void ( *signal ( int sig, void (*func) ( int ) ) ) ( int );
```

Fixe la fonction à appeler lorsque l'erreur *sig* se produira.

```
int raise ( int sig );
```

Provoque l'erreur (l'exception) *sig*.

stddef.h / cstddef

Définition de caractéristiques générales et partagées que l'on retrouve aussi souvent dans d'autres fichiers d'inclusion.

A titre d'exemple les types *size_t* et *wchar_t* ou la constante *NULL*;

wchar.h / cwchar

Gestion des caractères étendus.

wtype.h / cwtype

Comme *ctype* mais pour les caractères étendus.

Les classes, 1ière

Introduction

Avertissement:

Dans ce chapitre et ceux qui suivent nous ne présentons plus que des possibilités propres à C++; nous ne le préciserons plus dans la suite!!

Avant d'aborder la bibliothèque propre à C++ nous avons besoin de quelques notions de base liées à la programmation objet. Sans le savoir nous connaissons déjà un certain nombre de choses liées aux classes car tout ce que nous avons dit à propos des structures (et des unions) reste valable pour les classes. On peut dire a priori que structures et classes sont équivalentes à une différence importante de visibilité près. Dans une structure les membres (données ou fonctions) sont par défaut visibles du monde extérieur à la structure alors que pour la classe ils restent par défaut internes à celle-ci. Toutefois, autant dans une classe que dans une structure, ces règles de visibilités appliquées par défaut peuvent être forcées par le développeur dans le sens qu'il désire!

A titre d'exemple reprenons notre type *vecteur* défini dans le chapitre des structures, dans sa version avec des fonctions membres. Le fichier d'inclusion (de déclaration) prenait la forme suivante:

```
/*   Outils de base pour la gestion de vecteurs    */
struct vecteur
{
  float dx;
  float dy;
  /* Addition de 2 vecteurs */
  vecteur operator + ( const vecteur & );
  /* Affichage d'un vecteur */
  void affiche ( );
  /* Lecture d'un vecteur */
  void lire ( );
};
```

Pour l'instant le principal inconvénient de ce type *vecteur* réside dans le fait que l'utilisateur de la structure a accès à tout:

- Les fonctions membres ce qui en règle générale est tout à fait raisonnable!
- Les membres "données" (*dx*, *dy*), ce qui pourra très vite devenir dangereux!!

Nous devons pour des raisons de sécurité cacher les données: réaliser **l'encapsulation** de ces données!

Comme nous l'avons dit, les bases des structures et des classes sont identiques. Si dans le fichier d'inclusion proposé ci-dessus nous remplaçons le mot **struct** par **class**, la seule différence (de taille tout de même!) réside dans le fait que tous les membres sont par défaut privés, donc inutilisables en dehors de la classe, ce qui convenons-le ne sert pas à grand-chose!

Aussi bien dans les classes que dans les structures, le développeur peut décider des éléments qu'il désire cacher ou rendre accessibles aux utilisateurs de la classe. Dans cette optique, il peut définir des zones commençant par ***public***, respectivement ***private***. Une zone privée ou publique se termine avec la fin de la définition de la classe ou par le début d'une nouvelle zone commençant par l'un de ces mots réservés.

Ainsi nous obtenons le même résultat en remplaçant **struct** par **class** dans la définition des vecteurs et en ajoutant au début de la définition **public**:[1]

```
/*   Outils de base pour la gestion de vecteurs */
class vecteur
{
public:
  float dx;
  float dy;
  /* Affichage d'un vecteur */
  void affiche ( );
  /* Lecture d'un vecteur */
  void lire ( );
};
```

[1] L'opérateur + a volontairement été omis des exemples qui suivent!

Toutefois selon le bon principe d'encapsulation des données, il serait préférable de ne rendre publiques que les fonctions membres (et peut-être même pas toutes par la suite):

```
class vecteur
{
  float dx;
  float dy;
public:
  /* Affichage d'un vecteur */
  void affiche ( );
  /* Lecture d'un vecteur */
  void lire ( );
};
```

Nous obtenons le même résultat avec la forme structure en rendant explicitement privées les données:

```
struct vecteur
{
private:
  float dx;
  float dy;
public:
  /* Affichage d'un vecteur */
  void affiche ( );
  /* Lecture d'un vecteur */
  void lire ( );
};
```

Que nous pouvons aussi écrire:

```
struct vecteur
{
  /* Affichage d'un vecteur */
  void affiche ( );
  /* Lecture d'un vecteur */
  void lire ( );
private:
  float dx;
  float dy;
};
```

Le déplacement des données en fin de définition ne pose aucun problème pratique quel que soit le contenu des éléments qui précèdent. A l'intérieur d'une classe/structure[1] nous avons le droit d'utiliser ses membres avant leur définition.

Dans la définition d'une classe, le type en cours de définition peut s'utiliser:

- pour fixer le type de paramètres ou du résultat livré par une fonction:

```
vecteur addition ( vecteur v );
```

- comme pointeur sur un objet de ce type pour construire par exemple des structures chaînées:

```
vecteur *unAmi;
```

❑ *Utilisation des classes*

Une variable de type classe (un objet, une instance) se déclare comme n'importe quel autre variable[2]:

```
vecteur monVecteur; //[3]
```

Comme pour les variables simples usuelles, un tel objet appartient à la catégorie **auto.** Ses membres "données" contiennent une valeur indéterminée[4]. Volontairement nous n'utilisons plus ici le terme de classe de déclaration qui devient ambigu dans ce contexte!

Nous pouvons évidemment la forcer comme étant **static**:

```
static vecteur monVecteur;
```

---

[1] A partir de maintenant nous ne parlerons plus que de classe et construirons nos exemples sur ce modèle, tout en sachant qu'ils s'appliquent aussi aux structures et aux unions!

[2] Toutefois cette opération implique d'autres considérations que nous aborderons en traitant la notion de constructeurs!

[3] Cette déclaration peut aussi s'écrire:

```
vecteur monVecteur = vecteur ( );
```

Vous en comprendrez la raison lorsque nous traiterons des constructeurs.

[4] La situation pourra changer dès que nous aurons introduit la notion de constructeur.

Dans ce cas, ses membres "données" contiennent des octets à 0[1].

Comme toujours, la déclaration d'un objet peut se faire sous la forme d'une constante:

```
const vecteur vecteurConstant; //[2]
```

L'utilisateur de la classe a le droit de déclarer des pointeurs sur des éléments de type classe:

```
vecteur * adresseVecteur;
```

Dans les instructions, un objet dynamique se crée par l'intermédiaire de l'opérateur **new**[3]:

```
adresseVecteur = new vecteur; //[4]
```

Dans la philosophie objet, l'utilisation d'une méthode (l'appel d'une fonction membre) revient à envoyer un message à l'objet auquel on l'applique, en lui transmettant éventuellement des paramètres. En pratique cela consiste à donner d'abord le nom de l'objet, puis, par la notation pointée, la méthode à lui appliquer (le message à lui envoyer):

```
monVecteur.affiche ( );
```

Par contre, si l'opération à réaliser a été définie sous forme d'un opérateur, son utilisation se fait de manière usuelle:

```
vecteur v1, v2, v3;
…
v3 = v1 + v2;
```

Si l'objet est désigné par un pointeur nous pouvons utiliser la notation usuelle pour celui-ci:

```
adresseVecteur -> affiche ( );
```

ou

```
( *adresseVecteur ).affiche ( );
```

[1] Ce qui ne sera pas le cas des membres d'une classe déclarés **static**, c.f. *Membres statiques*.

[2] Nous pouvons le faire sous cette forme que s'il existe un constructeur par défaut fournissant une valeur aux membres "données" de l'objet. Toutefois, avec les constructeurs nous pourrons aussi donner les valeurs spécifiques désirées pour chacun des membres.

[3] Dans ce contexte purement C++ il faut oublier les fonctions traditionnelles C de la famille *alloc*, *malloc*.

[4] Là aussi la situation changera avec l'introduction des constructeurs.

Bien entendu, dans notre exemple de la classe *vecteur*, l'utilisateur ne peut pas écrire:

```
v1.dx = 12.0; // ERREUR
```

les membres *dx* et *dy* étant privés!

Si l'utilisateur final a besoin de connaître ces valeurs nous devons définir dans la classe des fonctions les fournissant, exemple:

```
float abscisse ( ) const;  //[1]
```

dont la définition correspondra simplement à:

```
float vecteur::abscisse ( ) const
{
  return dx;
}
```

❑ *Fonctions membres const*

Nous savons depuis longtemps que pour une fonction ordinaire nous pouvons lors de sa déclaration/définition préciser certains de ses paramètres comme étant **const**. Ce qui devrait garantir qu'ils ne subissent pas de modification dans la fonction et peuvent donc être appelés avec des constantes comme paramètres effectifs.

La situation demeure la même pour les classes et le compilateur C++ ne laissera pas appeler la fonction avec une constante comme paramètre effectif si le paramètre formel n'a pas été spécifié **const**.

Bien, mais qu'en est-il du paramètre implicite qui représente l'objet auquel on applique la méthode? Dans cette optique, nous complétons sa ligne d'en-tête en lui ajoutant le mot **const** tant dans sa déclaration que dans sa définition, comme nous l'avons fait ci-dessus pour la fonction abscisse.

Il en va de même pour une surcharge d'opérateur comme nous le verrons dans le programme complet qui va suivre!

[1] Pour le **const** se référer au paragraphe qui suit!

Rappelons-le, comme nous l'avions vu pour les structures, si la définition d'une fonction membre se trouve dans une unité compilée séparément, nous devons faire référence à sa classe par l'intermédiaire de l'opérateur de résolution de portée (::), sans quoi nous créons une autre fonction indépendante de la classe: vecteur::abscisse.

Rappelons aussi que les fonctions membres ont directement accès aux membres "données" de l'objet auquel on les applique. De plus l'objet en question se désigne par le pointeur **this**. Ainsi le **return** de la fonction *abscisse* aurait pu s'écrire:

```
return this->dx;
```

ou encore:

```
return (*this).dx;
```

Nous voilà prêts pour donner un premier exemple de code compilable:

```
/*
  Outils de base pour la gestion de vecteurs
   CLASS0.cpp
*/
#include <cstdlib>
#include <iostream>
using namespace std;

/* Definition du type vecteur */
class vecteur
{
public:
  /* Addition de 2 vecteurs */
  vecteur operator + ( const vecteur & ) const;
  /* Affichage d'un vecteur */
  void affiche ( ) const;
  /* Lecture d'un vecteur */
  void lire ( );
  /* Livre la valeur du membre dx */
  float abscisse ( ) const;
private:
  float dx;
  float dy;
};
```

```
/*
  Outils de base pour la gestion de vecteurs
*/
/* Addition de 2 vecteurs */
vecteur vecteur::operator + ( const vecteur & v ) const
{
  vecteur temp;  // On ameliorera ceci par la suite
  temp.dx = v.dx + dx;
  temp.dy = v.dy + dy;
  return temp;
}

/* Affichage d'un vecteur */
void vecteur::affiche ( ) const
{
  /* Cette forme "compliquee" n'est la que pour montrer
      les possibilites. En pratique on utiliserait:
        cout << "{ " << dx << ", " << dy << " }";
  */
  cout << "{ " << this -> dx << ", " << (*this).dy << " }";
}

/* Lecture d'un vecteur  */
void vecteur::lire ( )
{
  cout << "\nDonnez la valeur de dx: ";
  cin >> dx;
  cout << "Donnez la valeur de dy: ";
  cin >> dy;
}

/* Livre la valeur du membre dx */
float vecteur::abscisse ( ) const
{
  return dx;
}




/*
  Exemple: Utilisation d'une classe vecteur
*/
int main ( )
{
  /* Declaration des variables de travail  */
  static vecteur v1;
  vecteur v2 , v3;
  vecteur *ptv1 = &v1, *ptv2 = new vecteur;
```

```
  cout << "Exemple de classe\n\n";
  cout << "Vecteur statique avant traitement: ";
  v1.affiche ( ); cout << endl;
  cout << "Vecteur auto avant traitement: ";
  v2.affiche ( ); cout << endl;
  cout << " Donnez un premier vecteur: ";
  v1.lire ();
  cout << " Donnez un deuxieme vecteur: ";
  v2.lire ();

  cout << "Somme des 2 vecteurs = ";
  v3 = v1 + v2;
  v3.affiche ( ); cout << endl;

  cout << "V1 accede par pointeur ";
  ptv1 -> affiche(); cout << endl;

  *ptv2 = v2;  // On dispose de l'affectation
  cout << "Copie de V2 accede par pointeur ";
  ptv2 -> affiche(); cout << endl;
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution de ce code donne les résultats suivants:

```
Exemple de classe

Vecteur statique avant traitement: { 0, 0 }
Vecteur auto avant traitement: { 6.5861e-044, 2.8026e-045 }
 Donnez un premier vecteur:
Donnez la valeur de dx: 1
Donnez la valeur de dy: 2
 Donnez un deuxieme vecteur:
Donnez la valeur de dx: 3
Donnez la valeur de dy: 4
Somme des 2 vecteurs = { 4, 6 }
V1 accede par pointeur { 1, 2 }
Copie de V2 accede par pointeur { 3, 4 }

Fin du programme...Appuyez sur une touche pour continuer...
```

Comme nous l'avons déjà fait remarquer plus d'une fois, le découpage proposé ci-dessus n'est de loin pas idéal puisque nous avons mis l'ensemble de l'application dans un seul fichier. Solution d'autant mois raisonnable que l'objectif premier des classes vise à écrire du code réutilisable, ce qui ne sera certainement pas le cas en procédant ainsi! Nous le faisons ici uniquement pour montrer cette possibilité, une classe pouvant même se définir localement dans une fonction.

Dès le prochain exemple complet nous procéderons de manière plus propre avec un fichier d'inclusion pour la déclaration de la classe, un fichier incluant le premier pour la définition des fonctions de cette classe et le(s) fichier(s) l'utilisant.

❑ *Où définir les fonctions membres?*

Jusqu'à présent nous sommes partis du principe que les définitions des fonctions membres se trouvaient hors de la déclaration de la classe (d'où l'utilisation de l'opérateur de résolution de portée). Ceci n'est en aucun cas une obligation. Nous pouvons définir les fonctions membres directement à l'intérieur de la classe. Exemple avec la fonction abscisse déjà introduite:

```
class vecteur
{
  float dx;
  float dy;
public:
  float abscisse ( ) const
    { return dx };
  ...
};
```

Par définition une fonction définie à l'intérieur d'une classe est **inline**. Raisonnablement il s'agira de fonctions courtes ne générant que peu de code! Notez toutefois que même si elle n'est pas définie dans la classe, rien ne vous empêche de la spécifier explicitement comme **inline**.

Les constructeurs

La création d'un objet de type classe implique automatiquement l'appel d'un constructeur. Si, comme ce fut le cas dans nos exemples jusqu'à maintenant nous n'en définissons pas pour une classe donnée, un constructeur par défaut, ne faisant rien, est généré par le compilateur.

Un constructeur consiste en une fonction membre quelque peu particulière:

- Elle possède le même nom que la classe.
- Elle ne livre aucune valeur de retour, son en-tête ne comporte aucune indication dans ce sens (pas de **void**).
- En conséquence, elle ne comporte aucune instruction **return**.

Une classe peut fournir plusieurs constructeurs, il s'agit même d'une situation courante. Ils doivent toutefois posséder chacun une signature différente!

Normalement les constructeurs correspondent à des opérations très courtes. Pour cette raison, on les définit souvent dans la classe elle-même, sans toutefois que cela soit une obligation. Pour notre classe *vecteur* on peut imaginer par exemple:

```
vecteur ( float x, float y )
{
   dx = x;
   dy = y;
}


vecteur ( )
{
  dx = dy = 0.0;
}
```

Nous avons choisi ici et pour l'instant 2 constructeurs. Le premier, comportant 2 paramètres de type **float**, permet de donner explicitement une valeur spécifique à chacun des membres "données" de l'objet créé. Il permet donc de déclarer des variables sous la forme:

```
vecteur unVecteur ( 3.0, 5.0 );
```

Le deuxième, ne possédant aucun paramètre, devient le constructeur par défaut. Dès que l'on fournit un constructeur pour une classe le compilateur n'en génère plus un par défaut. Donc si nous n'en fournissons pas un de manière explicite il ne sera plus possible de déclarer des objets sans donner des valeurs d'initialisation, donc plus possible d'écrire simplement:

```
vecteur unVecteur;
```

Toutefois nuançons cette remarque, les paramètres d'un constructeur tout comme ceux des autres fonctions peuvent posséder des valeurs par défaut. L'appelant (ici le fait de déclarer un objet du type classe en question) n'a plus l'obligation de transmettre des paramètres effectifs lors de l'appel (ici des valeurs d'initialisation!). Donc si nous avions défini notre constructeur avec des valeurs par défaut pour ses paramètres, par exemple:

```
vecteur ( float x = 0.0, float y = 0.0 )
{
  dx = x;
  dy = y;
}
```

nous ne devons plus fournir notre deuxième constructeur, celui ci-dessus à lui seul permettant de déclarer:

```
vecteur v1, v2 ( 3.0 ), v3 ( 1.0, 5.0 );
```

Nous aurions aussi pu imaginer un constructeur de la forme:

```
vecteur ( float val = 0.0 )
{
  dx = dy = val;
}
```

Lui aussi peut servir de constructeur par défaut. Toutefois attention, sa cohabitation avec la version qui précède créerait des situations ambiguës que le compilateur n'admet évidemment pas!

Avec un constructeur à un seul paramètre (et uniquement dans ce cas) comme celui-ci-dessus pour *vecteur*, nous pouvons déclarer une variable sous la forme normale:

```
vecteur unVecteur ( 3.0 );
```

Mais aussi sous la forme:

```
vecteur unVecteur = 3.0; //[1]
```

Relevons aussi, ceci pour tous les constructeurs (qu'ils possèdent un ou plusieurs paramètres) que nous pouvons écrire une déclaration du genre:

```
vecteur unVecteur = vecteur (3.0);
```

Cette forme impliquera l'appel de 2 constructeurs: un pour la création de l'objet *unVecteur* et un deuxième appel pour la création d'un vecteur temporaire qui après son affectation à *unVecteur* sera détruit automatiquement[2].

La déclaration de tableaux dont les éléments sont d'un type classe se fait de manière usuelle, comme pour les types simples:

```
vecteur tabVecteur [ 3 ];
```

Dans ce cas le constructeur est appelé pour chacun des éléments du tableau. Un tableau peut aussi s'initialiser lors de sa déclaration:

```
vecteur v1 ( 1.0,3.0 ), v2 ( 1.1, 2.2 ), v3;
...
vecteur tab1 [ 3 ] = { v1, v2, v3 };
vecteur tab2 [ 2 ] = { vecteur (2.0, 3.0), vecteur
(1.0) };
```

L'objet de type classe désigné par un pointeur peut se créer lors de la déclaration du pointeur:

```
vecteur * pt0 = new vecteur;
```

Si désiré, un tel objet s'initialise lors de la déclaration du pointeur:

```
vecteur * pt1 = new vecteur ( 3.3, 7.7 );
```

Un tableau de pointeurs se construit aussi dynamiquement:

```
vecteur * pt2 = new vecteur [ 7 ];
```

[1] La raison principale de cette double formulation réside essentiellement dans le fait qu'elle permet de créer des classes génériques applicables aussi bien sur des types de base que sur des classes.

[2] Se référer à la partie *destructeurs* ci-après!

Constructeurs par copie en profondeur

Comme son nom le laisse supposer, le but d'un tel constructeur consiste à initialiser un objet de type classe lors de sa déclaration (construction) avec la valeur d'un autre objet du même type. On qualifie généralement une telle opération de clonage.

Un constructeur par copie possède une forme particulière: son premier paramètre (généralement il n'en a qu'un et si ce n'est pas le cas les autres doivent posséder une valeur par défaut!) sera transmis comme une référence.

Un tel constructeur pour notre classe *vecteur* pourrait prendre la forme:

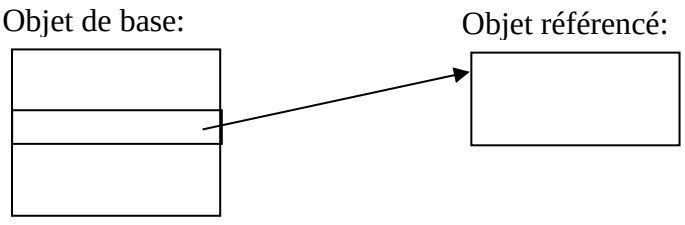
```
vecteur ( const vecteur & v )
{
  dx = v.dx;
  dy = v.dy;
}
```

Toutefois sous cette forme nous n'avons pas besoin de le fournir. En cas d'absence le compilateur en génère un qui réalise une copie membre à membre de la source dans la destination. C'est ce que l'on appelle une copie triviale ou superficielle!

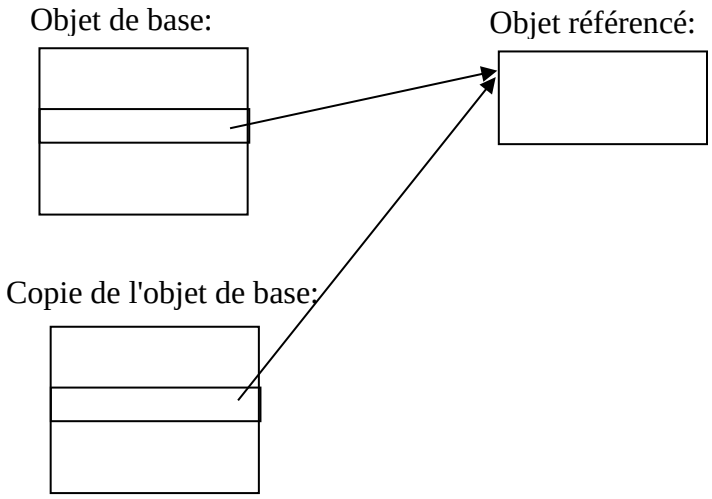
Relevez la présence du qualificatif **const** qui nous permet éventuellement de transmettre une constante *vecteur* comme valeur d'initialisation!

L'utilisation de ces constructeurs par copie se révèle bien plus fréquente qu'il n'y paraît à première vue. Rappelez-vous que chaque fois que vous transmettez un paramètre par valeur à une fonction, en pratique une copie du paramètre est transmise, d'où l'utilisation implicite d'un tel constructeur.

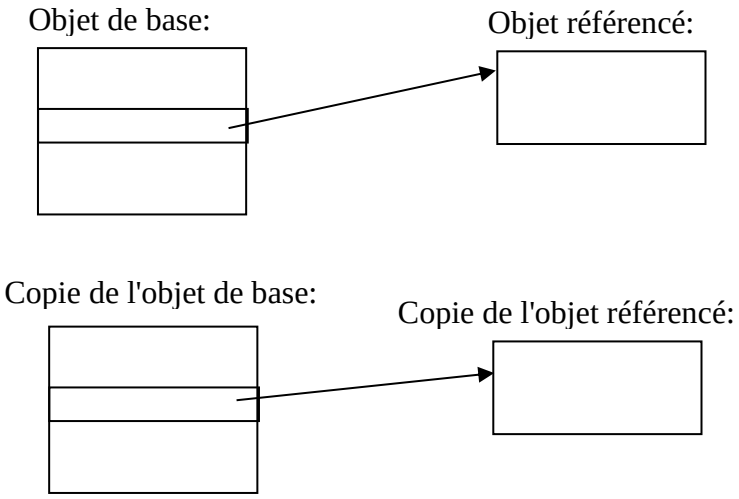
Cette copie superficielle par clonage se révèle parfois insuffisante pour aboutir à l'objectif visé. Les membres d'une classe peuvent être de n'importe quel type, y compris un type pointeur. Ce membre contiendra alors l'adresse d'un autre objet. On peut représenter la situation de la manière suivante:



Maintenant que se passe-t-il avec une copie triviale par clonage? Nous obtenons la situation suivante:



Cela peut être voulu et raisonnable, mais certainement pas dans tous les cas! Souvent nous avons besoin d'obtenir aussi une copie de l'objet référencé, soit la situation[1]:



Pour obtenir ce résultat nous devons dans le constructeur par copie créer explicitement un nouvel objet du type de celui qui est référencé, et affecter à cet objet l'information désirée, selon les besoins de l'application.

[1] La même problématique se pose avec l'opérateur d'affectation (=) que nous pourrons aussi surcharger!

Si nous admettons qu'un objet de notre classe *vecteur* comporte maintenant un membre "donnée" de plus, du genre:

```
float dx;
float dy;
int *identifiant;
```

où le membre *identifiant* servirait à contenir l'adresse d'un élément créé dynamiquement[1].

Un constructeur "normal" avec valeurs par défaut pourrait prendre la forme:

```
vecteur ( float x = 0.0, float y = 0.0, int id = 0 )
{
  dx = x;
  dy = y;
  identifiant  = new int ( id );
  *identifiant = id;
}
```

Par contre, un constructeur par copie prendrait lui une forme du genre:

```
vecteur ( const vecteur & v, int id = 0 )
{
  dx = v.dx;
  dy = v.dy;
}
```

Il crée donc un nouvel objet pointé et sauve son adresse dans *identifiant* puis il met à jour selon les besoins[2] la variable en question.

Ce même mécanisme se répercute à différents niveaux de profondeur pour autant que dans toute la hiérarchie chaque classe possède un constructeur adapté.

[1] Nous avons choisi ici un champ de type **int** pour simplifier les écritures. Dans la réalité l'objet ainsi référencé sera certainement plus complexe et vraisemblablement lui-même d'un type classe.

[2] Ici nous avons choisi une solution triviale.

Les destructeurs

Un destructeur consiste lui aussi en une fonction membre un peu particulière:

- Elle possède obligatoirement le même nom que la classe précédé du caractère ~.
- Elle ne possède pas de paramètres.
- Elle ne livre aucune valeur de retour, son en-tête ne comporte aucune indication dans ce sens (pas de **void**).
- En conséquence, elle ne comporte aucune instruction **return**.

Exemple: `~vecteur ( );`

Un tel destructeur est appelé automatiquement lorsqu'un objet de type classe disparaît:

- A la fin de son bloc de déclaration pour une variable **auto**.
- A l'appel de l'opérateur **delete** pour une variable dynamique créée par un **new**.
- A la fin de l'exécution de l'application pour des variables **static**.

S'il était logique suivant l'application de fournir plusieurs constructeurs pour une classe donnée, il n'en va pas de même pour le destructeur qui, lui, doit être unique[1].

S'il semble logique pour les constructeurs (ou tout au moins pour la majorité d'entre eux) de les spécifier **public**, pour les destructeurs la solution retenue n'a en principe aucune importance puisqu'ils sont appelés automatiquement. Toutefois, par habitude, on les rend généralement aussi **public**.

Si le développeur ne met aucun destructeur à disposition, comme pour les constructeurs, le compilateur en génère un par défaut qui ne fait rien!

Le destructeur ne s'occupe pas de libérer la place mémoire occupée par l'objet lui-même, mais il devra le faire pour les éléments référencés par des pointeurs depuis cet objet. En d'autres termes, dans les cas triviaux le destructeur généré par défaut convient parfaitement.

Dans le cas de la classe vecteur dans son état actuel, son destructeur peut prendre simplement la forme:

```
vecteur::~vecteur ( )
{
  delete identifiant;
}
```

[1] Le compilateur se révélerait incapable de distinguer 2 destructeurs puisqu'ils ne possèdent aucun paramètre.

Opérateur d'affectation

Si nous traitons de classes comportant des membres pointeurs, nous devons encore au minimum et impérativement mettre en évidence les problèmes liés à l'opérateur d'affectation. Sa problématique correspond à celle des constructeurs. Par défaut elle réalise une copie superficielle, ce qui convient fort bien si notre objet ne contient pas des membres pointeurs. Si tel n'est pas le cas, à nous de surcharger l'opérateur d'affectation "=". L'opération ressemble considérablement à celle d'un constructeur par recopie en profondeur, avec toutefois un certain nombre de nuances importantes.

Tout d'abord l'opérateur livre un résultat: la valeur affectée. Pour notre type vecteur, la méthode déclarée comme fonction membre de la classe prendra la forme suivante:

```
vecteur & operator = ( const vecteur & );
```

Ensuite, sa définition pourrait prendre une forme du genre de celle-ci-dessous:

```
vecteur & vecteur::operator = ( const vecteur & v )
{
  /* Si on n'affecte pas a lui-meme */
  if ( &v != this )
  { int idTemp = *v.identifiant * 100;
    delete identifiant;
    dx = v.dx;
    dy = v.dy;
    identifiant  = new int;
    *identifiant = idTemp;
  }
  return * this;
}
```

Relevons:

- Il faut ne rien faire d'autre que livrer le résultat si l'on tente d'affecter une variable à elle-même. Bien entendu écrire explicitement v1 = v2; ne paraît pas très raisonnable, mais une telle situation par l'intermédiaire de pointeurs devient tout à fait raisonnable.
- Il faut en principe libérer la place occupée par l'ancien objet pointé, à moins, comme cela pourrait être le cas dans notre exemple, qu'on la réutilise comme nouvel objet pointé!

Voici un exemple complet, mais dans lequel nous n'avons gardé que les fonctionnalités nécessaires à l'exemple, montrant l'utilisation de ces possibilités. Tout d'abord le fichier de déclarations dans lequel vous constaterez que nous avons pris, pour une fois et à titre d'exemple de ce qu'il faudrait faire, une précaution contre l'inclusion multiple, en définissant le symbole *__vecteur__*:

```
#if !defined __vecteur__
#define __vecteur__
/*
  Outils de base pour la gestion de vecteurs
   VECTEURS.h (CLASS1)
*/
#include <cstdlib>
class vecteur
{
public:

  /* Constructeur par copie */
  vecteur ( const vecteur & v, int id = 0 );

  /* Constructeur simple */
  vecteur ( float x = 0.0, float y = 0.0, int id = 0 );

  /* Destructeur */
  ~vecteur ( );

  /* Addition de 2 vecteurs */
  vecteur operator + ( const vecteur & ) const;

  /* Affectation entre vecteurs */
  vecteur & operator = ( const vecteur & );

  /* Affichage d'un vecteur */
  void affiche ( ) const;

  /* Permet de modifier l'identifiant */
  void changeIdentifiant ( int val );
private:
  float dx;
  float dy;
  int *identifiant;
};
#endif
```

Ensuite le fichier des définitions de notre classe:

```
/*
  Outils de base pour la gestion de vecteurs
      VECTEURS.CPP (CLASS1)
*/
#include "vecteurs.h"
#include <iostream>
using namespace std;


/* Constructeur par copie */
vecteur::vecteur ( const vecteur & v, int id )
{
  dx = v.dx;
 dy = v.dy;
  identifiant  = new int;
 *identifiant = id;
  cout << "Constructeur par copie: ";
  this->affiche (); cout << endl;
}


/* Constructeur simple */
vecteur::vecteur ( float x, float y, int id )
{
  dx = x;
  dy = y;
  identifiant  = new int;
  *identifiant = id;
  cout << "Constructeur simple: ";
  this->affiche (); cout << endl;
}


/* Destructeur */
vecteur::~vecteur ( )
{
 cout << "Destruction du vecteur: " ;
 this->affiche (); cout << endl;
  delete identifiant;
}
```

```
/* Affectation entre vecteurs */
vecteur & vecteur::operator = ( const vecteur & v )
{
  /* Si on n'affecte pas a lui-meme */
  if ( &v != this )
  { int idTemp = *v.identifiant * 100;
    delete identifiant;
    dx = v.dx;
    dy = v.dy;
    identifiant  = new int;
    *identifiant = idTemp;
  }
  cout << "Affectation: ";
  this->affiche (); cout << endl;
  return * this;
}


/* Addition de 2 vecteurs */
vecteur vecteur::operator + ( const vecteur & v ) const
{
  return vecteur ( v.dx + dx, v.dy + dy, -1 );
}


/* Affichage d'un vecteur */
void vecteur::affiche ( ) const
{
  cout << "Vecteur (" << *identifiant << ")=";
  cout << "{ " << dx << ", " << dy << " }";
}


/* Permet de modifier l'identifiant */
void vecteur::changeIdentifiant ( int val )
{
  *identifiant = val;
}
```

Bien entendu en pratique les constructeurs et le destructeur ne vont pas afficher des messages comme nous l'avons fait ici. Notre but dans cet exemple consiste à pouvoir suivre "à la trace" les opérations en regardant les résultats affichés et ainsi de mieux comprendre les mécanismes.

Finalement un programme de démonstration utilisant cette classe:

```
/*
  Exemple: Utilisation d'un classe
    TESTVECTEURS (CLASS1)
*/
#include "vecteurs.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main ( )
{
  /* Declaration des variables de travail  */
  static vecteur v1, v2 ( 2.0 ), v3 ;
  const vecteur v4 ( 8.0, 9.0, 1 );
  vecteur v5 (  v4, 5 );
  v4.affiche ( ); cout << endl;
  v5.affiche ( ); cout << endl;
  v5.changeIdentifiant ( 2 );
  v4.affiche ( ); cout << endl;
  v5.affiche ( ); cout << endl;
 vecteur *pt0 = new vecteur;
  vecteur *pt1 = new vecteur ( 3.0,5.0);
  vecteur *pt2 = new vecteur [3];
  vecteur v6 = vecteur ( 4.0, 5.0 );
  vecteur v7 [2]= { v1, v2 };
//  vecteur v4 [2]= { vecteur (2.0), vecteur (4.0) };

  cout << "Somme des 2 vecteurs = ";
  v3 = v1 + v2;
  v3.affiche ( );
  cout << endl;
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Et les résultats de l'exécution:

```
Constructeur simple: Vecteur (0)={ 0, 0 }
Constructeur simple: Vecteur (0)={ 2, 0 }
Constructeur simple: Vecteur (0)={ 0, 0 }
Constructeur simple: Vecteur (1)={ 8, 9 }
Constructeur par copie: Vecteur (5)={ 8, 9 }
Vecteur (1)={ 8, 9 }
```

```
Vecteur (5)={ 8, 9 }
Vecteur (1)={ 8, 9 }
Vecteur (2)={ 8, 9 }
Constructeur simple: Vecteur (0)={ 0, 0 }
Constructeur simple: Vecteur (0)={ 3, 5 }
Constructeur simple: Vecteur (0)={ 0, 0 }
Constructeur simple: Vecteur (0)={ 0, 0 }
Constructeur simple: Vecteur (0)={ 0, 0 }
Constructeur simple: Vecteur (0)={ 4, 5 }
Constructeur par copie: Vecteur (0)={ 0, 0 }
Constructeur par copie: Vecteur (0)={ 2, 0 }
Somme des 2 vecteurs = Constructeur simple: Vecteur (-1)={ 2, 0 }
Affectation: Vecteur (-100)={ 2, 0 }
Destruction du vecteur: Vecteur (-1)={ 2, 0 }
Vecteur (-100)={ 2, 0 }

Fin du programme...Appuyez sur une touche pour continuer...
Destruction du vecteur: Vecteur (0)={ 2, 0 }
Destruction du vecteur: Vecteur (0)={ 0, 0 }
Destruction du vecteur: Vecteur (0)={ 4, 5 }
Destruction du vecteur: Vecteur (2)={ 8, 9 }
Destruction du vecteur: Vecteur (1)={ 8, 9 }
Destruction du vecteur: Vecteur (-100)={ 2, 0 }
Destruction du vecteur: Vecteur (0)={ 2, 0 }
Destruction du vecteur: Vecteur (0)={ 0, 0 }
```

❑ *Les membres statiques*

Nous savons maintenant que chaque objet déclaré d'un type classe comporte sa propre copie des différents membres "données". Toutefois, dans certaines circonstances, il devient utile de pouvoir partager des informations entre tous les objets d'une même classe. Les membres "données" spécifiés explicitement **static** répondent à ce critère.

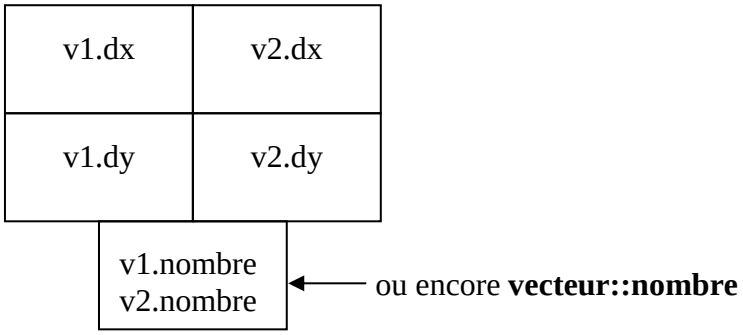
Ainsi avec une classe vecteur qui comprendrait entre autres dans sa définition:

```
class vecteur
{
  float dx;
  float dy;
  static int nombre;
  ...
};
```

Si nous déclarons:

```
vecteur v1,v2;
```

Nous nous trouvons dans la situation suivante:



Le membre *nombre* joue le rôle de variable globale dans le contexte de la classe! Il existe et peut s'utiliser même sans déclaration de variables de la classe, ceci sous la forme:

```
nom_de_classe::nom_de_membre
```

Toutefois, il faut évidemment que le membre soit **public** pour pouvoir y accéder en dehors de la classe. Si tel est le cas et que nous avons déclaré des objets de la classe nous pouvons aussi accéder ce membre sous la forme usuelle:

```
nom_de_variable.nom_de_membre
```

Ne déduisez pas du fait de l'utilisation du mot réservé **static** que ces membres sont initialisés avec des octets nuls. C'est au développeur de la classe d'initialiser explicitement de tels membres[1], opération qu'il ne peut pas réaliser dans la partie déclaration de la classe et doit donc différer à la partie définition sous la forme générale:

```
type nom_de_classe::nom_de_membre = valeur;
```

Une fonction membre d'une classe, qu'elle possède ou non des paramètres, peut aussi se définir comme **static**. Cela signifie qu'elle ne s'applique pas à un objet spécifique de la classe. Elle devient très utile lorsqu'il ne s'agit pour elle que d'accéder[2] à des membres "données" statiques.

Sa déclaration se fait normalement dans la partie déclaration de la classe:

```
static void laFonction ( ...);
```

Dans la partie définition de la classe nous donnons de manière usuelle le corps de la fonction, sans le mot **static**:

```
void laClasse::laFonction ( ... )
{ ... }
```

Cette fonction, pour autant qu'elle soit **public**, s'utilise comme les autres membres **static**:

```
laClasse::laFonction ( ... );
```

Voici une nouvelle version de la classe *vecteur* mettant partiellement en évidence les derniers aspects théoriques introduits. Comme dans l'exemple précédent nous avons ajouté de nombreux affichages dans le but d'aider à la compréhension du déroulement des opérations:

[1] Qu'ils soient publics ou privés cela revient au même!

[2] Aussi bien pour obtenir une valeur que pour la modifier.

```
#if !defined __vecteur__
#define __vecteur__
/*
  Outils de base pour la gestion de vecteurs
   Vecteurs.h  (CLASSE2STATIC)
*/
#include <cstdlib>
class vecteur
{
public:
  /* Constructeur */
  vecteur ( float x = 0.0, float y = 0.0 );
  /* Destructeur */
  ~vecteur ( );
  /* Addition de 2 vecteurs */
  vecteur operator + ( const vecteur & );
  /* Affichage d'un vecteur */
  void affiche ( ) const;
  /* Lecture d'un vecteur */
  void lire ( );
  /* Exemple de fonction static */
  static void afficheNombre ( );
private:
  float dx;
  float dy;
  static int nombre;
};
#endif
```

```
/*
  Outils de base pour la gestion de vecteurs
   VECTEURS.CPP  (CLASSE2STATIC)
*/
#include "vecteurs.h"
#include <iostream>
using namespace std;

/* Initialisation du compteur d'objets */
int vecteur::nombre = 0;
```

```
/* Constructeur */
vecteur::vecteur ( float x, float y )
{
  dx = x;
  dy = y;
  nombre++;
  cout << "Creation du vecteur: ";
  this->affiche ();
  cout << ". Il y en a maintenant: " << nombre << endl;
}

/* Destructeur */
vecteur::~vecteur ( )
{
  nombre--;
  cout << "Destruction du vecteur: ";
  this->affiche ();
  cout << ". Il y en a maintenant: " << nombre << endl;
}

/* Addition de 2 vecteurs */
vecteur vecteur::operator + ( const vecteur & v )
{
  vecteur temp;  // Pour l'exemple uniquement
  temp.dx = v.dx + dx;
  temp.dy = v.dy + dy;
  return temp;
}

/* Affichage d'un vecteur */
void vecteur::affiche ( ) const
{
  cout << "{ " << dx << ", " << dy << " }";
}

/* Lecture d'un vecteur  */
void vecteur::lire ( )
{
  cout << "\nDonnez la valeur de dx: ";
  cin >> dx;
  cout << "Donnez la valeur de dy: ";
  cin >> dy;
}
```

```
/* Exemple de fonction static */
void vecteur::afficheNombre ( )
{
  cout << "** Nous avons actuellement: " << vecteur::nombre
       << " vecteur(s) **\n";
}



/*
  Exemple: Utilisation d'une classe avec membre static
    TESTVECTEURS.CPP  (CLASSE2STATIC)
*/
#include "vecteurs.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main ( )
{
  cout << "Pour l'exemple, avant toute operation\n";
  vecteur::afficheNombre ( );
  /* Declaration des variables de travail  */
  vecteur v1 ( 2.0, 2.0 );
  vecteur v2 ( 5.3 ), v3 ;
  vecteur * ptv1 = &v1, *ptv2 = new vecteur;

  cout << "\n\nVecteur v1 avant traitement: ";
  v1.affiche ( ); cout << endl;
  cout << " Donnez un premier vecteur: ";
  v1.lire ();
  cout << " Donnez un deuxieme vecteur: ";
  v2.lire ();

  v3 = v1 + v2;
  cout << "Somme des 2 vecteurs = ";
  v3.affiche ( ); cout << endl;

  cout << "V1 accede par pointeur ";
  ptv1 -> affiche(); cout << endl;

  *ptv2 = v2;  // On dispose de l'affectation
  cout << "Copie de V2 accede par pointeur ";
  ptv2 -> affiche(); cout << endl;
  delete ptv2;
```

```
  /* Bloc artificiel pour demontrer le comportement */
  {
    static vecteur v11 ( 10.1, 11.2 );
    vecteur v12;
    v12 = v11;
  }


  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Affichage d'une exécution de ce code:

```
Pour l'exemple, avant toute operation
** Nous avons actuellement: 0 vecteur(s) **
Creation du vecteur: { 2, 2 }. Il y en a maintenant: 1
Creation du vecteur: { 5.3, 0 }. Il y en a maintenant: 2
Creation du vecteur: { 0, 0 }. Il y en a maintenant: 3
Creation du vecteur: { 0, 0 }. Il y en a maintenant: 4


Vecteur v1 avant traitement: { 2, 2 }
 Donnez un premier vecteur:
Donnez la valeur de dx: 1
Donnez la valeur de dy: 2
 Donnez un deuxieme vecteur:
Donnez la valeur de dx: 3
Donnez la valeur de dy: 4
Creation du vecteur: { 0, 0 }. Il y en a maintenant: 5
Destruction du vecteur: { 4, 6 }. Il y en a maintenant: 4
Somme des 2 vecteurs = { 4, 6 }
V1 accede par pointeur { 1, 2 }
Copie de V2 accede par pointeur { 3, 4 }
Destruction du vecteur: { 3, 4 }. Il y en a maintenant: 3
Creation du vecteur: { 10.1, 11.2 }. Il y en a maintenant: 4
Creation du vecteur: { 0, 0 }. Il y en a maintenant: 5
Destruction du vecteur: { 10.1, 11.2 }. Il y en a maintenant: 4

Fin du programme...Appuyez sur une touche pour continuer...
Destruction du vecteur: { 4, 6 }. Il y en a maintenant: 3
Destruction du vecteur: { 3, 4 }. Il y en a maintenant: 2
Destruction du vecteur: { 1, 2 }. Il y en a maintenant: 1
Destruction du vecteur: { 10.1, 11.2 }. Il y en a maintenant: 0
```

❑ *Les membres constant*

Une donnée, membre d'une classe, peut être constante. Toutefois nous ne pouvons pas l'initialiser lors de sa déclaration car si chaque objet déclaré de cette classe possède ce champ constant, ne pouvant pas changer pour l'objet en question, chacun peut bénéficier d'une valeur différente. A titre d'illustration, imaginons par exemple que cette valeur représente un numéro d'identification unique. Nous pourrions l'utiliser pour nos vecteurs dont la déclaration prendrait une forme du genre:

```
class vecteur
{
  float dx;
  float dy;
  const int identifiant;
  ...
};
```

L'initialisation du membre constant doit impérativement se faire lors de la création de l'objet, ceci par une forme particulière des définitions du (ou des) constructeur(s), La déclaration du constructeur elle ne changeant pas. Ceci donnerait par exemple pour notre classe *vecteur*, la déclaration:

```
vecteur ( float x = 0.0, float y = 0.0, int id = 0 );
```

Notez que nous avons conservé les valeurs par défaut des paramètres, y compris pour *id* qui représentera la valeur du membre constant, ceci afin d'éviter des problèmes avec les objets temporaires créés automatiquement.

La définition proprement dite du constructeur prendra la forme:

```
vecteur::vecteur ( float x, float y, int id ): identifiant ( id )
{
  dx = x;
  dy = y;
}
```

Nous devons compléter la ligne d'en-tête par ":" suivi du nom du membre constant avec entre parenthèses le nom du paramètre formel à lui associer. De cette manière le champ *identifiant* de notre exemple prendra automatiquement la valeur transmise pour *id* ou la valeur par défaut qui lui est associée.

Si les objets de la classe comportent plusieurs champs constants, on poursuit le processus sous forme de liste dont les éléments sont séparés par des virgules avec à chaque fois le nom du membre constant concerné et entre parenthèses le nom du paramètre formel à lui associer.

Rappelons qu'un membre **static** n'existe qu'à un seul exemplaire, ceci reste vrai pour un membre constant. Dans ce cas particulier l'initialisation de la variable se fait lors de la déclaration et non pas selon le mécanisme que nous venons de décrire.

Par contre nous pouvons utiliser cette syntaxe pour tous les paramètres, ce qui veut dire que le constructeur ci-dessus peut aussi s'écrire:

```
vecteur::vecteur (float x, float y, int id): identifiant (id),
                                             dx (x), dy (y)
{  }
```

Notez bien ici la présence d'un corps de fonction vide puisque toutes les opérations se réalisent implicitement par la ligne d'en-tête, ce qui ne sera pas forcément toujours le cas.

Une telle forme s'avère aussi obligatoire lorsqu'un membre consiste en une référence. Rappelons nous qu'une référence s'initialise lors de sa déclaration et ne change plus de valeur par la suite.

Nous verrons dans le paragraphe suivant une autre situation nécessitant également ce formalisme.

❑ *Les membres classes*

Jusqu'à maintenant nous avons utilisé des membres "données" d'un type simple. Toutefois un membre peut être d'un type classe quelconque! Cette situation ne pose pas de problème particulier si ce n'est que les constructeurs doivent respecter la syntaxe présentée ci-dessus. Ainsi, si nous définissons une classe *couple* dont les 2 membres "données" seraient de notre classe *vecteur* maintenant bien connue, son constructeur pourrait prendre la forme:

```
couple::couple ( float x1, float y1, float x2, float y2 ):
                          v1  (  x1,  y1  ),  v2  (  x2,  y2  )
{ }
```

Donnons un exemple complet basé sur cette idée, non pas qu'il apporte une grande information sur cette notion en elle-même, mais il nous permettra en simplifiant la classe *vecteur* au maximum de présenter une structure hiérarchique un peu plus complexe puisqu'elle comportera cette fois 5 fichiers. A savoir:

- Le fichier de déclaration de la classe *vecteur*:

```
#if !defined __vecteur__
#define __vecteur__
/*
  Outils de base pour la gestion de vecteurs
   VECTEURS.H (CLASSE3)
*/
class vecteur
{
public:

  vecteur ( float x = 0.0, float y = 0.0 );

  /* Affichage d'un vecteur */
  void affiche ( ) const;
private:
  float dx;
  float dy;
};
#endif
```

- Le fichier de définition de la classe *vecteur*:

```
/*
  Outils de base pour la gestion de vecteurs
      VECTEURS.CPP (CLASSE3)
*/
#include "vecteurs.h"
#include <iostream>
using namespace std;

vecteur::vecteur ( float x, float y )
{
  dx = x;
  dy = y;
  this->affiche (); cout << endl;
}
```

```
/* Affichage d'un vecteur */
void vecteur::affiche ( ) const
{
  cout << "Vecteur ";
  cout << "{ " << dx << ", " << dy << " }";
}
```

- Le fichier de déclaration de la classe *couple*:

```
#if !defined __couple__
#define __couple__
/*
  Outils de base pour la gestion de couples de vecteurs
   COUPLE.h (CLASSE3)
*/
#include "vecteurs.h"
class couple
{
public:

  couple ( float x1 = 0.0, float y1 = 0.0,
           float x2 = 0.0, float y2 = 0.0 );

  /* Affichage d'un couple de vecteurs */
  void affiche ( ) const;
private:
  vecteur v1;
  vecteur v2;
};
#endif
```

- Le fichier de définition de la classe *couple*:

```
/*
  COUPLE.CPP (CLASSE3)
*/
#include "vecteurs.h"
#include "couple.h"
#include <iostream>
using namespace std;

couple::couple ( float x1, float y1, float x2, float y2 ): v1
                               ( x1, y1 ), v2 ( x2, y2 )
{ }
```

```
/* Affichage d'un couple de vecteurs */
void couple::affiche ( ) const
{
  cout << "Couple { ";
  v1.affiche(); cout << ", ";
  v2.affiche();
  cout << "}\n";
}
```

- Finalement un pseudo programme de test

```
/*
  Exemple: Utilisation de classes
    TESTVECTEURS.CPP (CLASSE3)
*/
#include "couple.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main ( )
{
  /* Declaration des variables de travail  */
  couple c1, c2 ( 2.0 ), c3 ( 1.0, 2.0, 3.0, 4.0 );

  c1.affiche ( );
  cout << endl;
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Qui affiche comme résultats:

```
Vecteur { 0, 0 }
Vecteur { 0, 0 }
Vecteur { 2, 0 }
Vecteur { 0, 0 }
Vecteur { 1, 2 }
Vecteur { 3, 4 }
Couple { Vecteur { 0, 0 }, Vecteur { 0, 0 }}


Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *Classes et conversions*

Depuis longtemps nous savons convertir une valeur numérique d'un certain type vers une valeur correspondante d'un autre type numérique, ceci par l'intermédiaire des opérateurs *cast*.

Exemple:

```
int i;
float f;
…
i = int (f);  // ou i = (int) f;
```

Comme nous avons à faire ici à une conversion dégradante, nous sommes obligés de procéder de manière explicite. Mais nous savons aussi que certaines conversions sont réalisées automatiquement de manière implicite:

```
f = i;
```

D'autres conversions se font de manière implicite:

- Si nécessaire, lors du passage de paramètres.
- Si nécessaire, lors de l'évaluation d'expressions[1].

Dans une certaine mesure nous pouvons aussi considérer certains constructeurs de classes comme des fonctions de conversion d'un type de base prédéfini vers un objet de type classe. Rappelez-vous de la classe *Vecteur*, l'un de ses constructeurs pourrait être:

```
Vecteur ( float x = 0.0 )
{
      dx = x;
      dy = 0.0;
}
```

[1] Rappelons aussi que dans des expressions les type **char** et **short** sont automatiquement convertis en **int** avant toute utilisation.

Ce constructeur nous pouvons l'utiliser sous la forme usuelle:

```
Vecteur v1 ( 2.0f );
```

La valeur **float** 2.0 sert de base à la construction du vecteur. Il y a bien implicitement conversion d'une valeur de type **float** en une valeur de type *Vecteur*.

D'ailleurs cette même déclaration peut aussi s'écrire:

```
Vecteur v1 = 2.0f;
```

Ici, à nouveau, le constructeur est appelé implicitement pour réaliser la conversion. Le fait de pouvoir également l'utiliser, non pas dans une déclaration mais comme instruction, démontre certainement encore mieux le mécanisme:

```
v1 = 7.3f;
```

Dans ce contexte imaginons que la classe *Vecteur* met à disposition une méthode permettant de livrer, par exemple, la composante *x* d'un vecteur. Nous pouvons alors considérer cette méthode comme un outil de conversion permettant de passer d'un type classe (ici *Vecteur*!) à un type de base (ici **float**!).

Toutefois il sera plus propre et plus avantageux de définir dans notre classe un opérateur *cast*. La définition de cet opérateur prend une forme un peu particulière, voici son prototype:

```
operator float ( ) const;
```

Le type du résultat de l'opérateur ne se donne pas explicitement puisqu'il est compris dans la forme de l'opérateur lui-même (**float** ()). Il doit toujours se définir comme une fonction membre. Le **const** ajouté en fin de déclaration, comme nous l'avions déjà rencontré dans un autre contexte, n'est pas indispensable syntaxiquement, mais il nous permet d'appliquer l'opérateur à des objets constants. Le corps de l'opérateur ne présente aucune particularité, il retourne simplement la valeur désirée, par exemple, dans notre cas, le membre *dx*.

La présence de cet opérateur implique des conséquences importantes. Evidemment nous pourrons toujours l'appeler de marinière explicite et ceci sous les 2 formes usuelles permises pour les opérateurs *cast*:

```
cout << float ( v1 ) << endl;
cout << (float) v1 << endl;
```

Mais rappelez-vous que le compilateur génère des conversions implicites lorsque c'est possible et nécessaire. Ainsi dans un programme nous pouvons écrire:

```
f = v1;
```

si nous avons déclaré *f* de type **flaot**. Il y a alors appel implicite de notre opérateur *cast*

pour convertir le *Vecteur* en **flaot** avant d'affecter cette valeur à la variable *f*. Il en irait de même avec une fonction possédant un paramètre de type **flaot** et que, lors de l'appel, nous transmettions un *Vecteur*. L'utilisation implicite va encore plus loin! Si dans notre programme nous écrivons:

```
cout << v1 + 3.0f << endl;
```

l'opération se réalise correctement, le vecteur *v1* est converti en **flaot** et c'est l'addition sur les **float** qui est utilisée, livrant ainsi un résultat de type **float**.

Attention toutefois, si dans nos outils nous avions aussi défini un opérateur "+" sur le type *Vecteur*, l'instruction ci-dessus, sous cette forme, ne serait plus possible. En effet le compilateur disposerait de 2 chemins possibles:

- Comme avant, convertir le *Vecteur* en **float** et utiliser l'addition sur les **float**.
- Ou, convertir le **float** en *Vecteur* (par l'intermédiaire du constructeur) et utiliser l'addition que nous avons définie pour le type *Vecteur*.

Sans entrer dans les détails, signalons qu'il existe d'autres possibilités de rendre des situations ambigües, par exemple de définir un deuxième opérateur *cast* vers un type **int** ou **double** à partir de *Vecteur*.

Par contre, si nous ne disposons pas d'opérateur *cast* mais de l'opérateur "+" sur les vecteurs, nous pouvons alors à nouveau écrire une instruction du genre:

```
v1 = v2 + 3.0f;
```

Cette fois la valeur réelle 3.0 est convertie implicitement en un objet *Vecteur* par l'intermédiaire du constructeur de la classe.

Une autre possibilité permet de demander au compilateur qu'un constructeur ne soit jamais appelé implicitement. Pour cela il suffit de mettre le mot réservé **explicit** devant la déclaration du constructeur en question:

```
explicit Vecteur ( float x = 0.0, float y = 0.0 );
```

Nous pouvons alors à nouveau écrire des expressions du genre:

```
cout << v1 + 3.0f << endl;
```

Ceci, même si nous disposons d'un opérateur "+" sur les vecteurs, car la conversion de *Vecteur* en **float** demeure la seule possible. Par contre avec cette formulation nous perdons la possibilité d'écrire des instructions du genre:

v1 = 5.0f;

le constructeur ne permettant plus l'appel implicite!

Avec une généralisation de ce que nous avons indiqué ci-dessus, il n'y a aucun problème particulier pour:

- Réaliser un constructeur qui prend comme paramètre un objet d'une autre classe.
- Réaliser un opérateur *cast*, d'une classe vers une autre classe.

Voici un petit exemple de programme illustrant certaines de ces possibilités. Nous y avons ajouté des messages dans le constructeur et l'opérateur *cast* afin de suivre la trace de l'exécution:

```
#if !defined __Vecteur__
#define __Vecteur__
/*
  Outils de base pour la gestion de Vecteurs
   CAST/Vecteurs.h
*/
class Vecteur
{
public:

  /* Constructeur simple */
  Vecteur ( float x = 0.0, float y = 0.0 );

  /* Affichage d'un Vecteur */
  void affiche ( ) const;

//  /* Livre la partie dx (une des propositions faites)*/
//  float x ( ) const;

  /* Operateur cast */
  operator float ( ) const;

/* Addition de 2 vecteurs (incompatible avec d'autres possibilites) */
//  Vecteur operator + ( const Vecteur & ) const;
private:
  float dx;
  float dy;
};
#endif
```

```
/*
  Outils de base pour la gestion de Vecteurs
      CAST/Vecteurs.CPP (CLASS1)
*/
#include "Vecteurs.h"
#include <iostream>
using namespace std;

/* Constructeur simple */
Vecteur::Vecteur ( float x, float y )
{
  cout << "*Constructeur*\n";
  dx = x;
  dy = y;
}


/* Affichage d'un Vecteur */
void Vecteur::affiche ( ) const
{
  cout << "{ " << dx << ", " << dy << " }\n";
}

/* Operateur cast */
Vecteur::operator float ( ) const
{
  cout << "***Operateur cast***\n";
  return dx;
}



/*
  Exemple: Utilisation de conversions
    CAST/Testvecteur
*/
#include "Vecteurs.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main ( )
{
  /* Declaration des variables de travail  */
  Vecteur v1 ( 2.0 );
  Vecteur v2 = 3.0f;   // ou meme: v2 = 3;
  float f;

  v1.affiche ( );
  v2.affiche ( );
```

```
  v2 = 5.0f;   // ou meme: v2 = 5;
  v2.affiche ( );
  /* cast explicites */
  cout <<  float ( v2 ) << endl;
  cout <<  (float) v1 << endl;
  /* cast implicites */
  f = v2;
  cout << f << endl;
  cout << v2 + 3.0f << endl;
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution de ce programme livre comme résultats:

```
*Constructeur*
*Constructeur*
{ 2, 0 }
{ 3, 0 }
*Constructeur*
{ 5, 0 }
***Operateur cast***
5
***Operateur cast***
2
***Operateur cast***
5
***Operateur cast***
8

Fin du programme...Appuyez sur une touche pour continuer...
```

Classes: bitset, valarray et complex

Introduction

Dans ce chapitre nous introduisons quelques éléments de la bibliothèque C++ qui ne font pas partie de la bibliothèque C. Ils dérivent tous de la notion de classe. Dans le contexte C++, on parle de la bibliothèque *STL* (pour *Standard Template Library)*.

Nous ne donnerons pas tous les détails de chaque élément présenté ci-dessous, l'objectif consistant simplement à se faire une idée de leur utilisation dans les situations les plus courantes.

Nous avons déjà présenté la classe *vector* et bien que nous ne l'ayons pas fait dans tous les détails, nous n'y reviendrons pas ici. Toutefois rappelez-vous de son existence et de ses possibilités, cela peut toujours servir. Elle appartient à ce que l'on appelle les conteneurs[1] qui feront l'objet d'une description plus détaillée par la suite. Dans ce premier chapitre globalement consacré à la bibliothèque nous ne parlerons que des éléments qui ne sont pas des conteneurs et que nous n'avons pas encore abordés!

limits

Nous avions commencé la présentation de la bibliothèque héritée de C par introduire les éléments nous permettant de connaître les caractéristiques "numériques" valables dans l'environnement de travail (*limits.h / climit*). C++ offre en plus une autre possibilité allant dans ce sens, mais plus orientée vers une formulation propre à C++. Il s'agit de la classe générique *numeric_limits* et de ses spécialisations pour les différents types numériques[2] à savoir: **char**, **signed char**, **unsigned char**, **wchar**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **float**, **double** et **long double**.

[1] Plus précisément les conteneurs séquentiels.

[2] Ou considérés comme tels!

Ces classes comportent des définitions de constantes ou de fonctions fournissant les valeurs caractéristiques des différents types. Notons toutefois que certaines de ces valeurs n'ont pas de sens pour certains types. Plutôt que de donner de relativement longues et fastidieuses explications, nous vous fournissons ci-dessous un programme présentant les principales informations disponibles et la manière d'y accéder ainsi que les résultats obtenus dans notre environnement de travail. Pour les éléments correspondants, les valeurs obtenues sont évidemment les mêmes que celles que nous avions déjà vues dans le cadre de la bibliothèque C! Pour obtenir ces informations il vous faut inclure le fichier <*limits*>[1].

```
/*
   Programme exemple: Demontrant certaines limites propres
    a l'environnement de travail
   LIMITESCPP.CPP
*/
#include <limits>  // Valeurs dépendantes de l'implementation
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  cout << "\nnumeric_limits\n\n";

  cout << "Caracteres:\n";
  cout << "Le plus petit: " << numeric_limits<char>::min()
       << " de code: " << int (numeric_limits<char>::min()) << endl;
  cout << "Le plus grand: " << numeric_limits<char>::max()
       << " de code: " << int (numeric_limits<char>::max()) << endl;
  cout << "Ils utilisent " << numeric_limits<char>::digits
      << " bits et sont" ;
  if ( numeric_limits<char>::is_signed )
    cout << " signes\n\n";
  else
    cout << " non signes\n\n";

  cout << "Entiers non signes (pour changer):\n";
  cout << "Le plus petit: " << numeric_limits<unsigned int>::min()
       << ", le plus grand: " << numeric_limits<unsigned int>::max()
       << "\nIls utilisent " << numeric_limits<unsigned int>::digits
      << " bits et sont" ;
  if ( numeric_limits<unsigned int>::is_signed )
    cout << " signes\n\n";
  else
    cout << " non signes\n\n";
```

[1] Et non pas <climits>!

```
  cout << "Les reels \"double\":\n";
  cout << "Le plus petit: " << numeric_limits<double>::min()
       << ", le plus grand: " << numeric_limits<double>::max()
       << "\nIls utilisent " << numeric_limits<double>::digits
       << " bits ce qui represente "
       << numeric_limits<double>::digits10
       << " chiffres significatifs.\nIls sont evidemment";
  if ( numeric_limits<double>::is_signed )
    cout << " signe\n\n";
  else
    cout << " non signe\n";
  cout << "Le plus petit exposant (base 2): "
       << numeric_limits<double>::min_exponent << " soit en base 10: "
       << numeric_limits<double>::min_exponent10 << endl;
  cout << "Le plus grand exposant (base 2): "
       << numeric_limits<double>::max_exponent << " soit en base 10: "
       << numeric_limits<double>::max_exponent10 << endl;
  cout << "Base de la representation: "
       << numeric_limits<double>::radix;
  if ( numeric_limits<double>::is_exact )
    cout << ", ils ont une representation exacte\n";
  else
    cout << ", ils n'ont pas une representation exacte\n";
  cout << "Ecart entre 1.0 et le reel suivant: "
       << numeric_limits<double>::epsilon () << endl;
  cout << "La plus grande erreur d'arrondi: "
       << numeric_limits<double>::round_error () << endl;
  if ( numeric_limits<double>::has_infinity )
    cout << "Ces grandeurs ont une representation de l'infini\n";
  else
    cout << "Ces grandeurs n'ont pas de representation de l'infini\n";

  if ( numeric_limits<double>::has_signaling_NaN )
    cout << "Ces grandeurs ont une representation du NAN\n";
  else
    cout << "Ces grandeurs n'ont pas une representation du NAN\n";
  cout << "Les entiers eux... ";
  if ( numeric_limits<int>::has_signaling_NaN )
    cout << "ont une representation du NAN\n";
  else
    cout << "n'ont pas une representation du NAN\n";


  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution de ce programme dans notre environnement de travail fournit les résultats suivants:

```
numeric_limits

Caracteres:
Le plus petit: Ç de code: -128
Le plus grand: ⌂ de code: 127
Ils utilisent 7 bits et sont signes

Entiers non signes (pour changer):
Le plus petit: 0, le plus grand: 4294967295
Ils utilisent 32 bits et sont non signes

Les reels "double":
Le plus petit: 2.22507e-308, le plus grand: 1.79769e+308
Ils utilisent 53 bits ce qui represente 15 chiffres significatifs.
Ils sont evidemment signes

Le plus petit exposant (base 2): -1021 soit en base 10: -307
Le plus grand exposant (base 2): 1024 soit en base 10: 308
Base de la representation: 2, ils n'ont pas une representation exacte
Ecart entre 1.0 et le reel suivant: 2.22045e-016
La plus grande erreur d'arrondi: 0.5
Ces grandeurs ont une representation de l'infini
Ces grandeurs ont une representation du NAN
Les entiers eux... n'ont pas une representation du NAN
Appuyez sur une touche pour continuer...
```

❑ *La classe bitset*

La classe *bitset* est générique, elle nous permet de gérer facilement des bits, opération relativement courante en programmation système. La norme nous promet une réalisation efficace si le nombre de bits demandé ne dépasse pas la taille d'un entier long non signé. Nous n'avons pas appris à développer une classe générique. Un certain nombre de règles introduites pour les fonctions génériques s'appliquent également aux classes, toutefois il existe plusieurs différences que nous ne développerons pas ici! Signalons simplement que dans le cas qui nous intéresse, au moment de la déclaration d'un objet de cette classe nous devons le paramétrer avec une valeur entière mise entre < >. Cette valeur représente le nombre de bits à gérer:

```
bitset<8> flag;
```

Ici *flag* représente un ensemble de 8 bits!

Contrairement à d'autres classes (les vraies classes conteneurs!) la taille d'un objet *bitset* se fixe à la compilation et ne change pas par la suite.

Les constructeurs

Ils peuvent prendre plusieurs formes:

- Le plus simple, celui que nous avons utilisé ci-dessus pour déclarer la variable *flag* ne possède pas de paramètre. L'objet ainsi créé voit tous ses bits mis automatiquement à 0.
- Le deuxième constructeur prend un paramètre de type **unsigned long int** dont la valeur (les bits) sert à initialiser la variable *bitset*:

```
unsigned long int uint = 42;
bitset<8> flag ( uint );
```

Si vous fournissez une valeur trop grande par rapport au nombre de bits réservés, une exception sera levée.

- Le troisième constructeur prend comme paramètre un objet de la classe *string* initialisé avec des 0 et des 1:

```
string s ( "101010" );
bitset<8> flag ( s );
```

Attention: les bits les plus à droite de la chaîne correspondent à ceux de poids faible pour le *bitset*. Si la chaîne est plus courte que le nombre de bits à remplir les bits de poids fort du *bitset* seront mis à 0. Si la chaîne est plus longue que le nombre de bits à remplir, les caractères de droite seront ignorés. Si la chaîne comporte d'autres caractères que des 0 et des 1 une exception sera levée.

Les entrées/sorties

La classe *biset* surcharge de manière conventionnelle les opérateurs << et >>.

L'opérateur de sortie << affiche les bits du poids fort au poids faible de gauche à droite.

L'opérateur d'entrée >> lit les bits du poids fort au poids faible de gauche à droite. Pour la lecture le comportement correspond à ce qui se passe lors de l'affectation.

Les manipulations de bits

Les opérations logiques bit à bit ainsi que les affectations et les comparaisons ne peuvent se réaliser qu'entre *bitset* de même taille.

La classe met à disposition les opérateurs logiques bit à bit: ~ , | , &, ^

de même que leur combinaison avec une affectation: ~= , |= , &= , ^= .

Elle surcharge aussi les opérateurs de décalage à gauches (<<) et à droite (>>), ainsi que leur combinaison avec une affectation (<<= et >>=). Dans tous les cas des 0 entre à droite ou à gauche. Exemple:

```
flag <<= 2;
```

On dispose également de l'affectation (=) au sens usuel du terme.

L'accès individuel aux différents bits se réalise par l'intermédiaire de l'opérateur [ ] tout comme pour les tableaux, avec la même signification et les mêmes principes, à une exception importante près: la tentative d'accès à un bit inexistant provoque le levée d'une exception. Tant à l'utilisation qu'à l'affectation, ce bit correspond à un booléen, que l'on peut toujours traiter comme un entier.

La classe *bitset* met également à disposition des fonctions membres permettant de gérer et de tester les bits d'un ensemble sans utiliser l'opérateur [ ]. Il s'agit de:

- La méthode *test* qui possède comme paramètre le numéro du bit à tester et qui livre *true* si ce bit est à 1 et *false* sinon.

Exemple:
```
cout <<  flag.test(7) << endl;
```

- La méthode surchargée *flip* qui, si elle n'a pas de paramètre, inverse tous les bits de l'ensemble auquel on l'applique, alors que si elle possède un paramètre entier[1], seul le bit désigné par cette valeur sera inversé.

Exemples:
```
cout <<  flag.flip ()  << endl;
cout <<  flag.flip (5) << endl;
```

- La méthode surchargée *reset* qui, si elle n'a pas de paramètre, met à *false* (0) tous les bits de l'ensemble auquel on l'applique, alors que si elle possède un paramètre entier, seul le bit désigné par cette valeur sera mis à *false* (0).

Exemples:
```
cout <<  flag.reset ()  << endl;
cout <<  flag.reset (3) << endl;
```

[1] En réalité de type *size_t*!

- La méthode surchargée *set* qui, si elle n'a pas de paramètre, met à *true* (1) tous les bits de l'ensemble auquel on l'applique, alors que si elle possède un premier paramètre entier, seul le bit désigné par cette valeur sera mis à *true* (1), à moins de spécifier un deuxième paramètre de type booléen avec la valeur *false*, auquel cas le bit en question prend cette valeur.

Exemples:
```
cout <<  flag.set ()          << endl;
cout <<  flag.set (3)         << endl;
cout <<  flag.set (3, false ) << endl;
```

- La méthode booléenne sans paramètre *any* livre *true* si l'ensemble auquel on l'applique possède au moins un bit à 1.

Exemples:
```
cout <<  flag.any () << endl;
```

- La méthode booléenne sans paramètre *none* livre *true* si l'ensemble auquel on l'applique ne possède aucun bit à 1.

Exemples:
```
cout <<  flag.none () << endl;
```

- La méthode sans paramètre *size* livre une valeur entière[1] correspondant à la taille (le nombre de bits) du *bitset* auquel on l'applique.

Exemples:
```
cout <<  flag.size () << endl;
```

- La méthode sans paramètre *count* livre une valeur entière[2] correspondant au nombre de bits à 1 dans l'objet auquel on l'applique.

Exemples:
```
cout <<  flag.count () << endl;
```

❑ Les conversions

Si les constructeurs à disposition permettent d'initialiser un objet à partir d'un *string* ou d'un **unsignend long int**, la classe fournit les méthodes permettant de réaliser les opérations inverses. Il s'agit de: *to_ulong()* pour convertir un *bitset* vers un **unsignend long int** et pour la conversion en *string* la fonction générique dont la syntaxe d'utilisation vous paraîtra bien obscure (comme souvent en C++) puisque nous n'avons pas abordé ces éléments en théorie. Nous vous donnons donc brutalement ci-dessous un exemple d'utilisation basé sur les déclarations déjà

[1] En réalité de type size_t!

[2] En réalité de type size_t!

introduites dans ce chapitre. Prenez cet exemple comme une recette de cuisine:

```
s = flag.to_string<string::value_type, string::traits_type,
                        string::allocator_type>();
```

Nous voilà au terme de cette présentation de la classe *bitset*. Donnons pour conclure un exemple de programme compilable utilisant quelques-unes des possibilités présentées ci-dessus:

```
/*
  Demonstration de quelques possibilites de la classe bitset
  BITSET.CPP
*/
#include <iostream>
#include <bitset>
#include <string>
using namespace std;

int  main ( )
{
  bitset<14> ensemble1;
  string s ( "101011" );
  bitset<8> ensemble2 ( s );
//  bitset<8> ensemble2 ( s ,3 ,3 );  // Aussi possible!!
  unsigned long int uint = 43;
  bitset<8> ensemble3 ( uint );
  bitset<8> ensemble4 ( 15 );

  cout << "Utilisation de la classe bitset\n\n";
  cout << "Ensemble 1: " << ensemble1 << endl;
  cout << "Ensemble 2: " << ensemble2 << endl;
  cout << "Ensemble 2 en unsigned long: "
       << ensemble2.to_ulong () << endl;
  cout << "Ensemble 3: " << ensemble3 << endl;
  cout << "Ensemble 4: " << ensemble4 << endl;
  ensemble4 [7] = 1;
  cout << "Ensemble 4 modifie: " << ensemble4 << endl;
  cout << "Ensemble 4 & Ensemble 3: "
       << (ensemble4 & ensemble3) << endl;
  cout << "Donnez au maximum " << ensemble1.size() << " bits: ";
  cin  >> ensemble1;
  cout << "Valeur lue: " << ensemble1 << endl;
  uint = ensemble1.to_ulong();
  cout << "Sa conversion en unsigned long int " << uint << endl;
  s = ensemble1.to_string<string::value_type, string::traits_type,
                          string::allocator_type>();
  cout << "Sa conversion en string " << s << endl;
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution nous donne l'affichage suivant:

```
Utilisation de la classe bitset

Ensemble 1: 00000000000000
Ensemble 2: 00101011
Ensemble 2 en unsigned long: 43
Ensemble 3: 00101011
Ensemble 4: 00001111
Ensemble 4 modifie: 10001111
Ensemble 4 & Ensemble 3: 00001011
Donnez au maximum 14 bits: 100111001
Valeur lue: 00000100111001
Sa conversion en unsigned long int 313
Sa conversion en string 00000100111001
Appuyez sur une touche pour continuer...
```

❑ *La classe valarray*

La classe *valarray* est générique. Elle nous permet de gérer relativement facilement des vecteurs[1] et même des tableaux à plusieurs dimensions. La généricité porte sur le type des éléments des tableaux ainsi gérés. Ce type peut être n'importe quel type de base: **bool**, **char**, **int**, **long**, **unsigned**, **float**, **double**, **long double**, **complex**. En fait il peut s'agir d'un type de n'importe quelle classe supportant (définissant) les opérateurs usuels applicables à ces types de base.

L'intérêt principal d'une telle classe réside dans le fait qu'une implémentation spécifique peut tirer le meilleur parti du matériel utilisé. Ainsi avec une architecture SIMD (Single Instruction Multiple Data) dite aussi vectorielle, les opérations sur les éléments de tableaux pourront se faire en parallèle. D'ailleurs, la classe et ses dérivées sont prévues pour privilégier l'efficacité souvent au détriment de la souplesse et de la sécurité. Ainsi, par exemple, les opérations sont généralement prévues entre tableaux de mêmes dimensions, mais aucun contrôle effectif ne s'opérant, les résultats obtenus demeurent imprévisibles si l'on ne respecte pas cette contrainte!

❑ Quelques constructeurs

La classe met à disposition plusieurs constructeurs.

- Le plus simple n'a pas de paramètre; il réserve un tableau initialement vide, mais dont la taille et le contenu pourront varier au cours du temps comme pour tous les autres *valarray*, ceci en fonction des opérations réalisées. La classe étant générique, nous devons spécifier, comme pour tous les autres *valarray*, le type des futurs éléments du tableau.

  Exemple: `valarray<int> tab1;`

- Le deuxième constructeur possède un paramètre de type *size_t*[2] précisant la taille initiale du tableau. Dans ce cas tous les éléments du tableau réservé sont initialisés par défaut avec des octets à 0.

  Exemple: `valarray<int> tab2 (3);`

[1] Essentiellement au sens "calcul numérique" du terme!

[2] Rappel: *size_t* correspond à un type entier!

- Le troisième constructeur demande comme premier paramètre la valeur avec laquelle chaque élément du tableau sera initialisé; le type de cette valeur doit correspondre à celui annoncé en paramètre de généricité. Il possède un deuxième paramètre de type *size_t* précisant la taille initiale du tableau.

  Exemple: `valarray<int> tab3 ( -1, 3);`

- Un quatrième constructeur possède comme premier paramètre un tableau "classique" d'éléments du type des éléments du *valarray* et qui servira à l'initialiser. Il possède un deuxième paramètre de type *size_t* précisant la taille initiale du *valarray*. Si cette taille est plus petite que celle du tableau d'initialisation, seules les premières valeurs de celui-ci sont utilisées. Si cette taille est plus grande, seuls les premiers éléments du *valarray* sont initialisés, les autres contenant des valeurs indéterminées.

  Exemple:
  ```
  double t [] = { -0.1, 0.0, 0.1, 0.2 };
  valarray<double> tab4 ( t, 4 );
  ```

  Note: dans l'exemple ci-dessus nous avons fourni explicitement comme deuxième paramètre du constructeur la valeur 4, notre tableau servant de base à l'initialisation comportant effectivement 4 éléments. Toutefois, si nous voulons éviter des erreurs ou tout simplement rendre plus facile une éventuelle modification ultérieure[1], nous aurions pu écrire:

  ```
  valarray<double> tab4 (t, sizeof(t)/sizeof(double));
  ```

Affectation: =

L'affectation d'une expression *valarray* à une variable *valarray* de même dimension[2], pour autant que les éléments soient du même type, ne pose aucun problème. Bien évidemment les valeurs de la variable avant l'opération sont perdues.

Exemple: `tab3 = tab2;`

[1] Ce qui est toujours raisonnable comme principe de programmation!

[2] Si ce n'est pas le cas nous pouvons au préalable ajuster la taille de l'objet destination en lui appliquant la méthode *resize* à laquelle on transmet en paramètre la méthode *size* appliquée à l'objet source (c.f. "Connaissance et modification de taille")!

L'affectation d'une valeur du type des éléments du *valarray* à une variable *valarray* a pour effet d'affecter cette valeur à chacun des éléments.

Exemple: `tab3 = 2;`

De manière usuelle, l'opérateur d'affectation peut se combiner avec l'un des opérateurs:

`+ - * / % << >> & ^ |` (c'est-à-dire les opérateurs binaires arithmétiques et les opérateurs de manipulation de bits).

Exemple: `tab3 += tab2;  // =>  tab3 = tab3 + tab2;`

La valeur à droite du signe d'affectation peut non seulement être un *valarray* de même type et de même dimension que celui à gauche, mais il peut aussi s'agir d'une valeur du type des éléments du *valarray*. Dans ce cas l'opération s'effectue avec cette valeur sur chacun des éléments du tableau.

Exemple: `tab3 *= 2;  // =>  tab3 = tab3 * 2;`

❑ Accès aux éléments: [ ]

Les objets de cette classe étant considérés comme des tableaux au sens usuel du terme, on accède de manière classique aux éléments du tableau par l'opérateur [ ] qui est surchargé. Il offre les avantages et les inconvénients usuels qui lui sont associés:

- Aucun contrôle de la validité de l'indice n'est effectué[1]!
- La borne inférieure correspond toujours à 0!

Exemple: `++tab4[2];`

[1] Rappelons que l'objectif premier de la classe vise la performance et non la sécurité!

Connaissance et modification de la taille

La méthode sans paramètre *size()* appliquée à un objet de classe *valarray* livre en retour une valeur de type *size_t* représentant la taille actuelle de l'objet (son nombre d'éléments).

Exemple: `cout << tab4.size() << endl;`

La méthode surchargée *resize* a dans sa première forme un seul paramètre de type *size_t* qui permet de donner une nouvelle taille à un *valarray*. Toutefois faites attention, il ne s'agit pas là d'un réel redimensionnement comme ce sera le cas pour les vrais conteneurs que nous traiterons par la suite. En réalité, le tableau initial est détruit et remplacé entièrement par un nouveau de la taille demandée. Nous avons donc à faire à une opération relativement "coûteuse"!

Exemple: `tab4.resize(12);`

- Sous cette forme les éléments du tableau sont tous initialisés avec les valeurs fournies par le constructeur par défaut de la classe, à savoir des octets à 0.

Dans sa deuxième forme, la méthode possède un second paramètre, du type des éléments du tableau, représentant la valeur avec laquelle il faut tous les initialiser.

Exemple: `tab4.resize( 12, 3.5 );  //` [1]

Exemple 1

Voilà. Donnons un premier exemple de programme complet mettant en pratique l'essentiel des points présentés ci-dessus:

```
/*
  Demonstration de quelques possibilites de la classe valarray
  VALARRAY1.CPP
*/
#include <iostream>
#include <valarray>
using namespace std;
```

[1] Attention à l'ordre inversé des paramètres par rapport au constructeur correspondant!!!

```
/* Fonction d'affichage d'un valarray */
template <class T> void affiche ( const valarray<T> &tab )
{
  const int nbParLigne = 5;
  for ( size_t i = 0; i < tab.size();  )
  {
    cout << tab [i] << "  ";
    /* Passe a la ligne si necessaire! */
    if ( !( ++i % nbParLigne ) )
      cout << endl;
  }
  /* Si Passe a la ligne si necessaire! */
  if ( tab.size() % nbParLigne )
    cout << endl;
}

int  main ( )
{
  valarray<int> tab1;
  valarray<int> tab2 (7);
  valarray<int> tab3 ( -1, 3);
  double t [] = { -0.1, 0.0, 0.1, 0.2 };
  valarray<double> tab4 ( t, sizeof (t) / sizeof (double) );
  int val = 9;

  cout << "Utilisation de la classe valarray\n\n";
  cout << "Taille de tab1: " << tab1.size() << endl;
  cout << "Taille de tab2: " << tab2.size() << endl;
  tab1.resize ( 7, 3 );
  cout << "Les valeurs actuelles de tab1:" << endl;
  affiche ( tab1 );
  tab1 = tab2;
  cout << "Nouvelle taille de tab1: " << tab1.size() << endl;
  cout << "Les valeurs actuelles de tab1:" << endl;
  affiche ( tab1 );
  tab1 = 22;
  cout << "Les valeurs actuelles de tab1:" << endl;
  affiche ( tab1 );
  cout << "Les valeurs actuelles de tab2:" << endl;
  affiche ( tab2 );
  cout << "Les valeurs actuelles de tab3:" << endl;
  affiche ( tab3 );
  cout << "Les valeurs actuelles de tab4:" << endl;
  affiche ( tab4 );
  cout << "tab4[2] apres modification: " << ++tab4[2] << endl;

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Les résultats de l'exécution:

```
Utilisation de la classe valarray

Taille de tab1: 0
Taille de tab2: 7
Les valeurs actuelles de tab1:
3  3  3  3  3
3  3
Nouvelle taille de tab1: 7
Les valeurs actuelles de tab1:
0  0  0  0  0
0  0
Les valeurs actuelles de tab1:
22  22  22  22  22
22  22
Les valeurs actuelles de tab2:
0  0  0  0  0
0  0
Les valeurs actuelles de tab3:
-1  -1  -1
Les valeurs actuelles de tab4:
-0.1  0  0.1  0.2
tab4[2] apres modification: 1.1
Appuyez sur une touche pour continuer...
```

Bien que cela ne relève pas directement de la classe *valarray*, notons particulièrement la construction de notre fonction d'affichage. Nous voulons disposer d'une fonction permettant d'afficher une variable *valarray* dont les éléments peuvent être de n'importe quel type. Il nous faut donc réaliser une fonction générique dépendant du type des éléments du tableau, dont le paramètre est lui-même un *valarray* dépendant de ce paramètre de généricité, d'où le prototype de la fonction:

```
template <class T> void affiche ( const valarray<T> &tab );
```

Si notre fonction devait simplement pouvoir afficher des tableaux dont les éléments sont des entiers, alors son prototype se simplifie pour prendre la forme:

```
void affiche ( const valarray<int> &tab );
```

Opérations mathématiques

Nous pouvons appliquer entre *valarray* de même type l'ensemble des opérations arithmétiques applicables aux valeurs du type de ses éléments.

Un opérateur unaire tel "-" s'applique à chacun des éléments du *valarray*.

Exemple: `tab1 = -tab1;`

Les opérateurs binaires s'appliquent entre chaque élément de même position (indice) de ses opérandes pour donner la valeur du résultat à cette position.

Exemple: `tab1 = tab2 + tab3;`

Nous pouvons aussi (pour les tableaux d'éléments numériques) utiliser des fonctions mathématiques telles que sinus (*sin*) ou cosinus (*cos*).

Exemple: `tab1 = sin (tab1);`

Les opérateurs de comparaison

La classe met également à disposition tous les opérateurs de comparaison utilisables entre deux *valarray* de même dimension et d'éléments de même type. Le résultat obtenu consiste en un *valarray* de booléens de même taille que les 2 opérandes, dont chaque élément représente le résultat de la comparaison appliquée aux valeurs de même rang des opérandes.

Exemple:
```
valarray<bool> t ( 4 );
...
t = tab1 == tab2;
```

Fonctions spécifiques à la classe valarray

La classe met aussi à disposition quelques fonctionnalités plus spécifiques à la gestion des *valarray*.

- *min ()* : livre la plus petite valeur du tableau auquel on l'applique. Le résultat est du même type que celui des éléments du *valarray*.

Exemple: `cout << tab1.min();`

- *max ()* : livre la plus grande valeur du tableau auquel on l'applique. Le résultat est du même type que celui des éléments du *valarray*.

Exemple: `cout << tab1.max();`

- *sum ()* : livre la somme des valeurs du tableau auquel on l'applique. Le résultat est du même type que celui des éléments du *valarray*.

Exemple: `cout << tab1.sum();`

- *shift (**int**)* : livre un *valarray* correspondant à celui auquel on l'applique dont les éléments sont décalés d'un nombre de positions égal à son paramètre de type **int**. Si la valeur du paramètre est positive nous obtenons un décalage vers la gauche alors que pour un paramètre négatif il s'agit d'un décalage vers la droite. Des 0 entrent respectivement à gauche ou à droite suivant le sens du décalage.

Exemple: `cout << tab1.shift( -2 );`

- *cshift (**int**)* : livre un *valarray* correspondant à celui auquel on l'applique dont les éléments sont décalés circulairement d'un nombre de positions égal à son paramètre de type **int**. Si la valeur du paramètre est positive nous obtenons un décalage vers la gauche alors que pour un paramètre négatif il s'agit d'un décalage vers la droite. Les éléments du tableau qui sortent à gauche rentrent à sa droite et inversement suivant le sens du décalage.

Exemple: `cout << tab1.cshift( 1 );`

- Finalement la méthode surchargée *apply* permet d'appliquer à chaque élément d'un *valarray* une fonction de notre propre choix que nous lui transmettons en paramètre. Notre fonction elle-même doit avoir un paramètre du type des éléments du *valarray*. La surcharge vient du fait que ce paramètre se transmet soit par valeur soit par référence.

Exemple:
```
int maFonction ( int );
...
tab1 = tab1.apply ( maFonction );
```

Il sera certainement agréable et pratique de définir notre propre fonction sous forme générique (adaptable au type des éléments du *valarray*), ce qui ne changera pas son utilisation, mais nous évite éventuellement de multiples définitions.

Exemple:
```
template <class T> T maFonction ( T v );
...
tab1 = tab1.apply ( maFonction );
```

Nous sommes prêts pour donner un deuxième exemple d'utilisation de *valarray*:

```
/*
  Demonstration de quelques possibilites de la classe valarray
  VALARRAY2.cpp
*/
#include <iostream>
#include <valarray>
#include <cstdlib>
using namespace std;

/* Fonction d'affichage d'un valarray */
template <class T> void affiche ( const valarray<T> &tab )
{
  const int nbParLigne = 5;
  for ( size_t i = 0; i < tab.size();  )
  {
    cout << tab [i] << "  ";
    /* Vide la ligne si complete! */
    if ( !( ++i % nbParLigne ) )
      cout << endl;
  }
  /* Si la derniere ligne n'a pas encore ete videe */
  if ( tab.size() % nbParLigne )
    cout << endl;
}
```

```
/* Exemple de fonction triviale a appliquer a tous les
    elements d'un valarray
*/
template <class T> T maFonction ( T v )
{
  return 2 * v + 1;
}

int  main ( )
{
  double tab [] = { 0.0, 0.78, 1.571, 3.14 };
  valarray<double> t1 ( tab, sizeof (tab) / sizeof (double) );
  valarray<double> t2 (4);
  int tabint1 [] = { 7, 2, 1, 9, 3 };
  valarray<int> t3 ( tabint1, sizeof (tabint1) / sizeof (int) );

  /* Les operations "arithmetiques" */
  cout << "Utilisation de la classe valarray\n\n";
  cout << "Les valeurs actuelles de t1:" << endl;
  affiche ( t1 );
  t2 = t1 + t1;
  cout << "Les valeurs actuelles de t2=t1+t1:" << endl;
  affiche ( t2 );
  t1 = cos ( t2 );
  cout << "Les valeurs de t1=cos(t2):" << endl;
  affiche ( t1 );
  cout << "Les valeurs de -t1:" << endl;
  /* Affiche etant generique => conversion explicite obligatoire */
  affiche ( valarray<double>(-t1) );

  /* Operations specifiques aux valarray */
  cout << "\n\nLe tableau d'entiers t3:" << endl;
  affiche ( t3 );
  cout << "Le plus petit de ses elements: " << t3.min () << endl;
  cout << "Le plus grand de ses elements: " << t3.max () << endl;
  cout << "La somme de ses elements: " << t3.sum () << endl;

  cout << "Decalage de 2 positions a droite:" << endl;
  affiche ( t3.shift ( -2 ) );
  cout << "Decalage de 1 position a gauche:" << endl;
  affiche ( t3.shift ( 1 ) );
  cout << "Decalage circulaire de 2 positions a droite:" << endl;
  affiche ( t3.cshift ( -2 ) );
  cout << "Decalage circulaire de 1 position a gauche:" << endl;
  affiche ( t3.cshift ( 1 ) );
  cout << "Application de maFonction(2*X+1) a t3:" << endl;
  t3 = t3.apply ( maFonction );
  affiche ( t3 );

  int tabint2 [] = { 5, 15, 3, 9, 7 };
  valarray<int> t4 ( tabint2, sizeof (tabint2) / sizeof (int) );
  cout << "Resultat de t3 == t4:" << endl;
```

```
  affiche ( valarray<bool> ( t3 == t4 ) );
  cout << "Resultat de t3 > t4:" << endl;
  affiche ( valarray<bool> ( t3 > t4 ) );

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Son exécution nous livre le résultat suivant:

```
Utilisation de la classe valarray

Les valeurs actuelles de t1:
0  0.78  1.571  3.14
Les valeurs actuelles de t2=t1+t1:
0  1.56  3.142  6.28
Les valeurs de t1=cos(t2):
1  0.0107961  -1  0.999995
Les valeurs de -t1:
-1  -0.0107961  1  -0.999995


Le tableau d'entiers t3:
7  2  1  9  3
Le plus petit de ses elements: 1
Le plus grand de ses elements: 9
La somme de ses elements: 22
Decalage de 2 positions a droite:
0  0  7  2  1
Decalage de 1 position a gauche:
2  1  9  3  0
Decalage circulaire de 2 positions a droite:
9  3  7  2  1
Decalage circulaire de 1 position a gauche:
2  1  9  3  7
Application de maFonction(2*X+1) a t3:
15  5  3  19  7
Resultat de t3 == t4:
0  0  1  0  1
Resultat de t3 > t4:
1  0  0  1  0
Appuyez sur une touche pour continuer...
```

Partie de valarray

Nous pouvons de différentes manières sélectionner une partie des éléments d'un *valarray* pour construire par exemple un nouveau tableau d'éléments ne comportant que certaines valeurs du tableau initial ou pour aller modifier certaines valeurs du tableau initial car les sélections que nous allons apprendre à réaliser représentent des *lvalues* donc des éléments modifiables!

Sélection par vecteur d'indice

La première approche possible consiste à fournir un *valarray* complémentaire fixant les indices des éléments à prendre en considération dans le tableau source. Notez que ces indices ne doivent pas nécessairement être donnés dans l'ordre. Cette possibilité n'offre que peu d'intérêt dans le cas de modifications du tableau source par contre elle devient très intéressante dans le contexte de la création d'un nouveau tableau. Le comportement devient indéterminé si la même valeur d'indice est utilisée plus d'une fois.

Avec les déclarations:

```
int tab   [] = { 0, 78, 15, 314, 1, 2 };
valarray<int> t1 ( tab, 6 );
size_t tabIndice [] = { 5, 2, 4, 0 };
valarray<size_t> indice ( tabIndice, 4 ) );
```

`t1 [ indice ]` correspond à un tableau constitué des valeurs: `{ 2, 15, 1, 0 }`

Sélection par masque

La deuxième possibilité consiste à déterminer les éléments sélectionnés à l'aide d'un *valarray* de booléens dans lequel chaque position à *true* correspond à celle d'une valeur à prendre en considération dans le tableau source.

Ainsi avec les déclarations:

```
int tab   [] = { 0, 78, 15, 314, 1, 2 };
valarray<int> t1 ( tab, 6 );
bool mask [] = { false, true, true, false, true, false };
valarray<bool> tbool ( mask, 6 );
```

Après l'instruction:

```
t1  [ tbool ] = -1;
```

t1 contient les valeurs: `{ 0, -1, -1, 314, -1, 2 }`

Sélection par tranches (slice)

La troisième possibilité nous permet par l'intermédiaire de la fonction *slice* de fixer un indice de départ dans le *valarray*, le nombre de valeur à prendre en considération et le pas entre chaque indice à considérer. Il s'agit là, dans l'ordre, des 3 paramètres à transmettre à *slice*.

Ainsi avec les déclarations:

```
int tab   [] = { 0, 78, 15, 314, 1, 2 };
valarray<int> t1 ( tab, 6 );
```

Après l'instruction: `t1 [ slice ( 1, 3, 2 ) ] = -2;`

t1 contient les valeurs: `{ 0, -2, 15, -2, 1, -2 }`

Cette dernière forme peut devenir utile pour gérer des tableaux à plusieurs dimensions sous la forme d'un *valarray* donc finalement d'un vecteur. Ainsi pour une *matrice* rectangulaire de *nbLignes* et sur *nbColonnes*, l'expression:

```
matrice [ slice ( i * nbColonnes, nbColonnes, 1 ) ]
```

définit les éléments de la ligne numéro *i* de la matrice, et l'expression:

```
matrice [ slice ( j, nbLignes, nbColonne ) ]
```

définit les éléments de la colonne numéro *j* de la matrice[1].

Donnons maintenant un exemple illustrant quelques unes de ces possibilités:

```
/*
  Demonstration de quelques possibilites de la classe valarray
  VALARRAY3.CPP
*/
#include <iostream>
#include <valarray>
using namespace std;

/* Fonction d'affichage d'un valarray */
template <class T> void affiche ( const valarray<T> &tab )
{
  const int nbParLigne = 6;
  for ( size_t i = 0; i < tab.size();  )
  {
    cout << tab [i] << "  ";
    /* Vide la ligne si complete! */
    if ( !( ++i % nbParLigne ) )
      cout << endl;
  }
  /* Si la derniere ligne n'a pas encore ete videe */
  if ( tab.size() % nbParLigne )
    cout << endl;
}
```

[1] Il existe une autre possibilité allant dans ce sens (*gslice*) que nous ne décrirons pas ici!

```
int  main ( )
{
  int tab   [] = { 0, 78, 15, 314, 1, 2 };
  valarray<int> t1 ( tab, sizeof (tab) / sizeof (int) );
  bool mask [] = { false, true, true, false, true, false };
  valarray<bool> tbool ( mask, sizeof (mask) / sizeof (bool) );
  size_t tabIndice [] = { 5, 2, 4, 0 };
  valarray<size_t> indice ( tabIndice,
                            sizeof (tabIndice) / sizeof (size_t) );

  cout << "Utilisation de la classe valarray\n\n";
  /* Selection par masque */
  cout << "Les valeurs actuelles de t1:\n";
  affiche ( t1 );
  cout << "Les valeurs du masque:\n";
  affiche ( tbool );
  valarray<int> t2 = t1 [ tbool ];
  cout << "Les valeurs apres masquage de t1:\n";
  affiche ( t2 );

  /* Selection par tableau d'indice */
  cout << "\nLes valeurs de t1 sont inchangees, Indice vaut:\n";
  affiche ( indice );
  t2.resize( indice.size() );
  t2 = t1 [ indice ];
  cout << "Les valeurs apres selection par table d'indice:\n";
  affiche ( t2 );

  /* Selection par tranches */
  cout << "\nLes valeurs de t1 sont inchangees, slice(1,3,2)=-2\n";
  t2.resize( 3 );
  t1 [ slice ( 1, 3, 2 ) ] = -2;
  affiche ( t1 );

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution donne comme résultat:

```
Utilisation de la classe valarray

Les valeurs actuelles de t1:
0  78  15  314  1  2
Les valeurs du masque:
0  1  1  0  1  0
Les valeurs apres masquage de t1:
78  15  1

Les valeurs de t1 sont inchangees, Indice vaut:
5  2  4  0
Les valeurs apres selection par table d'indice:
2  15  1  0

Les valeurs de t1 sont inchangees, slice(1,3,2)=-2
0  -2  15  -2  1  -2
Appuyez sur une touche pour continuer...
```

❑ *La classe complex*

Nous nous contenterons de donner ici les grandes lignes du contenu de la classe *complex*, qui, comme son nom l'indique, met à disposition les outils de base relatifs à la gestion des nombres complexes.

La classe *complex* est générique et peut être instanciée avec l'un des types: **float**, **double** ou **long double**. Ses constructeurs peuvent donc prendre l'une des formes suivantes:

```
complex<float> c1;
complex<double> c2 ( 3.2, 5.3 );
complex<double> c3 ( c2 );
```

Le constructeur par défaut initialise les champs des parties réelles et imaginaires à 0.0. On fournit au deuxième constructeur les valeurs respectives de la partie réelle et de la partie imaginaire. Le troisième constructeur initialise la variable avec le contenu d'un autre objet du type complexe.

La classe met à disposition les entrées/sorties sur les complexes (<< / >>). L'affichage d'une valeur prend la forme:

```
(3.2,5.3)
```

Lors d'une saisie, l'utilisateur donne les valeurs sous la même forme[1]. Toutefois, il peut ne donner qu'une valeur réelle (donc aussi une valeur entière) qui représentera la partie réelle du complexe, sa partie imaginaire prenant alors la valeur 0.0.

On dispose des opérateurs unaires (+, -) ainsi que des 4 opérateurs arithmétiques de base (+, -, /, *) y compris sous leur forme de combinaison avec l'opérateur d'affectation (=). Ces opérations se font également sans problème entre un complexe et une valeur réelle du même type que celui des membres du complexe.

Les fonctions usuelles de la bibliothèque mathématique (*sin, cos, tan, sinh, cosh, tanh, sqrt, log, log10*) s'appliquent aussi aux complexes. Il en va de même pour la fonction *pow*, mais pour elle signalons que nous pouvons élever à une puissance entière (**int**), réelle du même type que les membres du complexe ou finalement à une puissance complexe.

Il existe aussi la possibilité (*polar*) d'obtenir une valeur complexe à partir de deux valeurs réelles représentant le complexe sous sa forme polaire.

[1] Ceci rend impossible l'utilisation d'une localisation utilisant la virgule au lieu du point décimal!

En bref, si nous disposons d'une valeur *c* d'un type complexe, les fonctions:

- `real ( c )`: livre la partie réelle du complexe.
- `imag ( c )`: livre la partie imaginaire du complexe.
- `norm ( c )`: livre la norme du complexe.
- `conj ( c )`: livre la valeur conjuguée du complexe.

Voilà, nous n'en dirons pas plus sur le sujet et donnons pour terminer cette partie un exemple de programme utilisant des complexes:

```
/*
  Demonstration de l'utilisation des nombres complexes
  COMPLEXES.CPP
*/
#include <iostream>
#include <complex>
#include <cstdlib>
using namespace std;

int  main ( )
{
  complex<double> c1, c2, resultat;
  char operateur;
  bool ok = true;

  cout << "Operations arithmetiques sur les complexes\n";
  cout << "\n\nDonnez le premier operande: ";
  cin >> c1;
  cout << "\n\nDonnez le deuxieme operande: ";
  cin >> c2;
  /* Recommencer si operateur incorrecte */
  do
  {
    cout << "\n\nDonnez l'operateur(+,-,*,/): ";
    cin >> operateur;
```

```
    /* En fonction du desire de l'utilisateur */
    switch ( operateur )
    {
      case '+':
        resultat = c1 + c2; break;
      case '-':
        resultat = c1 - c2; break;
      case '*':
        resultat = c1 * c2; break;
      case '/':
        resultat = c1 / c2; break;
      default: // Erreur
        ok = false;
    }
  } while ( !ok );

  // Afficher le resultat
  cout << c1 << "  " << operateur << "  " << c2 << " = " << resultat
       << endl;
  printf ( "Fin du programme " );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Un exemple d'exécution:

```
Operations arithmetiques sur les complexes


Donnez le premier operande: (0.0, 1.0)


Donnez le deuxieme operande: (0.0, -1.0)


Donnez l'operateur(+,-,*,/): *
(0,1)  *  (0,-1) = (1,0)
Fin du programme Appuyez sur une touche pour continuer...
```

Les amies

Introduction

Dans ce chapitre nous allons ajouter quelques notions complémentaires relatives à la programmation objet et plus particulièrement aux classes en C++. Rappelons tout d'abord que tout ce qui se fait sur une classe peut également se faire sur une structure (**struct**), la seule différence résidant dans la visibilité par défaut.[1]

Les fonctions amies simples

Encore un rappel: toutes les fonctions membres d'une classe ont un accès total à tous ses membres qu'ils soient **public** ou **private**. En dehors de la classes, par contre, seuls les membres **public** sont visibles et donc utilisables!

Une telle situation pose parfois des problèmes (entre autre d'efficacité!). La notion d'amies (fonction ou classe) peut contribuer à résoudre ce problème.

Une fonction amie s'annonce comme telle dans la classe en faisant précéder sa déclaration (son prototype) du mot réservé: **friend**. N'importe quelle unité, compilée séparément (se sera généralement le cas!) ou pas, peut alors implémenter cette fonction. Elle le fait comme pour une simple fonction "normale", sans aucune autre indication complémentaire. Généralement une telle fonction possède un ou plusieurs paramètres du type de la classe en question, et/ou livre un résultat de ce type, mais ce n'est pas une obligation absolue.[2]

Une fonction amie simple n'est pas une fonction membre d'une classe: elle ne possède pas un premier paramètre implicite du type de la classe.

[1] **public** par défaut pour une structure, **private** pour une classe. Toutefois rien ne s'oppose dans les 2 cas à ce que nous forcions les choses selon nos besoins!

[2] Toutefois, c'est ce qui justifie en principe d'avoir accès aux membres privés de la classe.

Voici un exemple compilable sous forme d'unités séparées, avec fichier d'inclusion, qui démontre cette possibilité:

```
#if !defined __vecteur__
#define __vecteur__
/*
  Exemple minimal d'une classe declarant une fonction amie
   VECTEURS.h (AMI1)
*/
/* Definition du type vecteur */
class vecteur
{
public:

  vecteur ( float x = 0.0, float y = 0.0 );

  /* Affichage d'un vecteur */
  friend void affiche ( const vecteur );

private:
  float dx;
  float dy;
};
#endif
```

```
/*
  Exemple minimal d'une classe declarant une fonction amie
   VECTEURS.cpp (AMI1)
*/
#include "vecteurs.h"

vecteur::vecteur ( float x, float y )
{
  dx = x;
  dy = y;
}
```

```
/*
  Exemple: Utilisation de fonction amie
    TESTVECTEURS.cpp (AMI1)
*/
#include "vecteurs.h"
#include <cstdlib>
#include <iostream>
using namespace std;

/*
    Affichage d'un vecteur: la fonction amie;
    Toutefois, n'importe qui peut se pretendre ami!!!
    ... et avoir acces aux membres prives!!!
*/
void affiche ( vecteur v )
{
  cout << "Vecteur ";
  cout << "{ " << v.dx << ", " << v.dy << " }";
}


int main ( )
{ /* Declaration des variables de travail  */
  vecteur v1, v2 ( 2.2 ), v3 ( 1.0, 2.0 );

  affiche ( v1 ); cout << endl;
  affiche ( v2 ); cout << endl;
  affiche ( v3 ); cout << endl;

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Dont l'exécution fournit les résultats suivants:

```
Vecteur { 0, 0 }
Vecteur { 2.2, 0 }
Vecteur { 1, 2 }

Fin du programme...Appuyez sur une touche pour continuer...
```

Relevez qu'ici la fonction amie *affiche* a été définie dans l'unité de compilation comportant le programme principal. Elle aurait pu l'être ailleurs!

Cette possibilité ouvre toutefois des brèches potentielles au niveau de la sécurité car il ne sera pas possible d'empêcher une unité de se faire passer pour une amie et ainsi de réaliser des opérations dangereuses.

❑ *Les fonctions amies opérateurs*

Nous savons que nous pouvons surcharger les opérateurs. Cette opération peut également se réaliser sans problème particulier sous la forme d'une fonction amie. Toutefois cette possibilité nous amène à reprendre une situation délicate que nous n'avions pas résolue par le passé: "pourquoi n'avions-nous pas pu surcharger les opérateurs d'entrées/sorties << et >> en tant que membre de nos classes"?

La raison est simple, une fonction membre possède toujours un premier paramètre implicite du type classe en question (l'objet auquel on applique la méthode!). Par définition les opérateurs << et >> eux s'appliquent à un flux. Ils ne peuvent donc pas disposer d'un paramètre implicite du type de la classe. Ceci implique que l'on ne peut pas les définir comme fonctions membres à part entière. La solution à ce problème: les fonctions amies!

Elles ne posent pas de problème particulier par rapport à ce qui a été dit ci-dessus. Voici donc un exemple illustrant cette possibilité:

```
#if !defined __vecteur__
#define __vecteur__
#include <iostream>
using namespace std;
/*
  Utilisation d'une fonction amie pour surcharger un operateur
  VECTEURS.h (AMI2)
*/
/* Definition du type vecteur */
class vecteur
{
public:

  vecteur ( float x = 0.0, float y = 0.0 );

  /* Surcharge de l'operateur == pour les objets vecteur */
  friend bool operator == ( const vecteur&, const vecteur& );
  /* Surcharge de l'operateur << pour les objets vecteur */
  friend ostream &operator << ( ostream &sortie, const vecteur& );
  /* Surcharge de l'operateur >> pour les objets vecteur */
  friend istream &operator >> ( istream &entree, vecteur& );

private:
  float dx;
  float dy;
};
#endif
```

```
/*
  Utilisation d'une fonction amie pour surcharger un operateur
  VECTEURS.cpp (AMI2)
*/
#include "vecteurs.h"
#include <iostream>
using namespace std;

vecteur::vecteur ( float x, float y )
{
  dx = x; dy = y;
}

/* Surcharge de l'operateur >> pour les objets vecteur */
istream &operator >> ( istream &entree, vecteur &v )
{
  char caractereCourant;
  const char debut = '{', fin= '}', milieu = ',';
  vecteur temp;
  bool enOrdre = true;

  entree >> caractereCourant;
  /* Lectures avec controle de validite! */
  if ( caractereCourant == debut )
  {
    entree >> temp.dx >> caractereCourant;
    if ( caractereCourant == milieu )
    {
      entree >> temp.dy >> caractereCourant;
      enOrdre = caractereCourant == fin;
    }
     else
    enOrdre = false;
  }
  else
    enOrdre = false;
  /* Si tout c'est bien passe: */
  if ( enOrdre )
   v = temp;
  else
    /* Activer l'indicateur d'erreur */
    entree.clear ( entree.rdstate () | ios::badbit );
  return entree;
}

/* Surcharge de l'operateur == pour les objets vecteur */
bool operator == ( const vecteur &v1, const vecteur &v2 )
{
  return v1.dx == v2.dx && v1.dy ==  v2.dy;
}
```

```
/* Surcharge de l'operateur << pour les objets vecteur */
ostream &operator << ( ostream &sortie, const vecteur &v )
{
  sortie << "Vecteur ";
  sortie << "{ " << v.dx << ", " << v.dy << " }";
  return sortie;
}



/*
  Exemple: Utilisation d'une fonction amie
    pour un operateur
    TESTVECTEURS.cpp (AMI2)
*/
#include "vecteurs.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main ( )
{
  vecteur const vecteurNul ( 0.0, 0.0 );
  vecteur v;

  cout << "Lecture/ecriture de vecteurs:\n\n";

  /* Tant que l'utilisateur ne donne pas un vecteur nul: */
  do
  {
    cout << "Donnez un vecteur { dx, dy }: ";
    /* Lecture avec contole: */
    if ( cin >> v )
      cout << "Le vecteur lu est: " << v << endl;
    else
    /* Erreur de lecture */
    {
      cout << "!!!!ERREUR!!!!\n\n";
      cin.clear ();
      cin.sync ();
    }
  } while ( !(v == vecteurNul) );
  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Résultat d'une exécuion:

```
Lecture/ecriture de vecteurs:

Donnez un vecteur { dx, dy }: {2.3,1.1}
Le vecteur lu est: Vecteur { 2.3, 1.1 }
Donnez un vecteur { dx, dy }: toto
!!!!ERREUR!!!!

Donnez un vecteur { dx, dy }: XXL
!!!!ERREUR!!!!

Donnez un vecteur { dx, dy }:     {   2    ,     3  }
Le vecteur lu est: Vecteur { 2, 3 }
Donnez un vecteur { dx, dy }: {0,0}
Le vecteur lu est: Vecteur { 0, 0 }
Fin du programme...Appuyez sur une touche pour continuer...
```

Notes:

- Relevez dans la surcharge de l'opérateur >> l'utilisation d'une variable temporaire, ceci afin de simuler au plus près le comportement sur les types de base, à savoir ne pas modifier le paramètre si l'opération n'a pas été menée correctement.
- Toujours pour être le plus conforme possible à ce qui se passe par défaut sur les types de base, dès la détection d'un problème nous arrêtons l'opération et activons l'indicateur: *badbit*.
- Dans cet exemple il n'était pas nécessaire de définir l'opérateur de comparaison == comme fonction amie. Cette forme n'a été introduite ici qu'à titre d'exemple! Nous aurions pu nous contenter de définir cet opérateur par une fonction membre usuelle de la forme:
-

```
bool operator == ( const vecteur& ) const;
```

Où encore sous la forme:

```
bool operator == ( vecteur ) const;
```

❑ *Les fonctions amies de plusieurs classes*

Une fonction amie peut l'être de plusieurs classes. Ceci se révèle nécessaire lorsqu'elle doit posséder des paramètres de deux classes différentes et accéder aux membres privés de chacune d'elles. Dans chaque classe elle est déclarée comme **friend**.

Dans un contexte de compilation séparée, ce que nous recommandons toujours, chacun des fichiers d'inclusion des classes devra comporter lui-même une annonce de l'autre (ou des autres si plus de 2 classes sont à considérer. Une telle annonce prend la forme simple suivante:

class nom_de_classe;

Si les 2 classes sont définies dans le même fichier source (en principe peu recommandé!), seule l'une d'entre-elles nécessite impérativement une telle annonce.

Voici un exemple, totalement artificiel, car si l'on veut un code réaliste, celui-ci devient vite très volumineux. Pour bien marquer la compilation séparée, nous avons tiré une ligne horizontale entre le contenu de chacun des fichiers:

```
#ifndef __c1__
#define __c1__
/* Exemple de classe artificielle */
/* C1.h (AMI3) */
class c2;
class c1
{
public:
  c1 ( int v1 = 1, int v2 = 2 );
  void friend affiche ( const c1, const c2 );
private:
  int m1, m2;
};
#endif
```

```
/* Exemple de classe artificielle */
/* C1.cpp (AMI3) */
#include "c1.h"
c1::c1 ( int v1, int v2 )
{
   m1 = v1; m2 = v2;
}
```

```
#ifndef __c2__
#define __c2__
/* Exemple de classe artificielle */
/* C2.h (AMI3) */
class c1;
class c2
{
public:  c2 ( float v1 = 1.1, float v2 = 2.2 );
  void friend affiche ( const c1, const c2 );
private:
  float m1, m2;
};
#endif
```

```
/* Exemple de classe artificielle */
/* C2.cpp (AMI3) */
#include "c2.h"
c2::c2 ( float v1, float v2 )
{
   m1 = v1; m2 = v2;
}
```

```
/*
  Test d'une fonction amie de plusieurs classes
  TestMultiAmies.cpp (AMI3)
*/
#include "c1.h"
#include "c2.h"
#include <iostream>
#include <cstdlib>
using namespace std;

/* Fonction amie */
void affiche ( c1 p1, c2 p2 )
{
  cout << "De c1: " << p1.m1 << " - " << p1.m2 << endl;
  cout << "De c2: " << p2.m1 << " - " << p2.m2 << endl;
}
```

```
int main ( )
{
  c1 v1, v2 ( 3 ), v3 ( 4, 5 );
  c2 v11, v12 ( 3.3 ), v13 ( 4.4, 5.5 );

  affiche ( v1, v11 );
  affiche ( v2, v12 );
  affiche ( v3, v13 );

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution de ce programme fournit les résultats suivants:

```
De c1: 1 - 2
De c2: 1.1 - 2.2
De c1: 3 - 2
De c2: 3.3 - 2.2
De c1: 4 - 5
De c2: 4.4 - 5.5
Appuyez sur une touche pour continuer...
```

Note: Dans cet exemple nous définissons à nouveau la fonction amie dans le fichier du programme principal, mais nous pouvons très bien le faire dans n'importe quel autre fichier compilé séparément.

❑ *Les fonctions membres amies*

La situation présentée ci-dessus peut se résoudre d'une autre manière, ce qui nous permettra d'introduire un nouveau point de théorie: les fonctions membres amies d'une autre classe.

Lors de la déclaration d'une fonction amie il est possible de spécifier quelles est la classe ayant le droit d'être amie donc de définir la fonction en question. Pour cela il suffit de mettre devant le nom de la fonction celui de la classe ayant ce droit, en séparant les 2 par l'opérateur de résolution de portée ::. Une telle situation réduit les risques de tromperies/erreurs (seule la classe en question ayant le droit de définir la fonction) mais ne les éliminent pas totalement.

Le prototype prendra la forme générale suivante:

```
type friend classe::fonction ( ... );
```

Dans le fichier d'inclusion déclarant la relation d'amitié il faudra inclure le fichier d'inclusion de la classe annoncée comme amie.

Dans la classe définissant la fonction amie, le prototype de celle-ci s'écrit de manière usuelle:

```
type fonction ( ... );
```

Par contre le fichier d'inclusion commencera par une annonce de l'autre classe.

Pour les fichiers implémentant les fonctions des classes en question il n'y a rien de particulier à signaler!

Puisque maintenant la fonction devient un membre de la classe *classe*, elle possède un premier paramètre implicite de cette classe, plus éventuellement d'autres paramètres de n'importe quel type, mais vraisemblablement au moins un du type de la classe ayant annoncée la relation d'amitié.

Voici un exemple, toujours artificiel, illustrant ces possibilités:

```
#ifndef __c1__
#define __c1__
/* Exemple de classe artificielle */
/* C1.h (AMI4) */
#include "c2.h"
class c1
{
public:
  c1 ( int v1 = 1, int v2 = 2 );
  void friend c2::affiche ( const c1 );
private:
  int m1, m2;
};
#endif
```

```
/* Exemple de classe artificielle */
/* C1.cpp (AMI4) */
#include "c1.h"
c1::c1 ( int v1, int v2 )
{
   m1 = v1; m2 = v2;
}
```

```
#ifndef __c2__
#define __c2__
/* Exemple de classe artificielle */
/* C2.h (AMI4) */
class c1;
class c2
{
public:  c2 ( float v1 = 1.1, float v2 = 2.2 );
  void affiche ( const c1 );
private:
  float m1, m2;
};
#endif
```

```
/* Exemple de classe artificielle */
/* C2.cpp (AMI4) */
#include "c2.h"
#include "c1.h"
#include <iostream>
using namespace std;
c2::c2 ( float v1, float v2 )
{
   m1 = v1; m2 = v2;
}

/* Fonction amie */
void c2::affiche ( c1 p1 )
{
  cout << "De c1: " << p1.m1 << " - " << p1.m2 << endl;
  cout << "De c2: " << this->m1 << " - " << this->m2 << endl;
}
```

```
/*
  Test d'une fonction amie
  TestMultiAmies.cpp (AMI4)
*/
#include "c1.h"
#include "c2.h"
#include <iostream>
#include <cstdlib>
using namespace std;

int main ( )
{
  c1 v1, v2 ( 3 ), v3 ( 4, 5 );
  c2 v11, v12 ( 3.3 ), v13 ( 4.4, 5.5 );

  v11.affiche ( v1 );
  v12.affiche ( v2 );
  v13.affiche ( v3 );

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution de ce programme fournit les mêmes résultats que la version qui précède!

❑ *Les classes amies*

Introduisons une dernière notion liée aux relations d'amitié. Si nous désirons définir une relation d'amitié pour toutes les fonctions d'une classe, nous pouvons évidemment pratiquer pour chacune d'elles de la même manière que présenté ci-dessus. Toutefois il sera alors plus simple de préciser que toute une classe est amie de l'autre!

Pour ce faire, il suffit de déclarer dans la classe celle devant devenir amie, ceci en le précisant de manière explicite sous la forme :

```
class c1
{
      friend class c2;
      ...
```

On ne précise plus devant les déclarations de fonctions qu'elles sont amies puisqu'elles le deviennent toutes !

Dans le fichier d'inclusion de la classe amie nous procèderons à une annonce de la classe ayant annoncée la relation d'amitié, ou, éventuellement selon les besoins, à l'inclusion de son fichier de prototypes!

Il n'y a rien d'autre à signaler pour cette situation. Tant du point de vue de leurs déclarations que de le leurs définitions, les fonctions s'écrivent de manière usuelle simple dans la classe amie.

Les classes génériques

Introduction

Nous avons déjà introduit la notion de généricité au niveau des fonctions. Même si les règles de base restent valables, la situation change passablement et la syntaxe se complexifie considérablement.

Nous avons toujours jusqu'à présent privilégié le principe de la compilation séparée. Toutefois ici, dans une première étape en tous cas, nous allons regrouper dans la définition de la structure les déclarations des fonctions membres. Ceci pour la simplification des notations et par un aspect pratique que nous justifierons par la suite.

Les paramètres

Comme pour les fonctions génériques, devant la déclaration normale d'une classe vous ajoutez le mot réservé **template** suivi entre < > d'un ou de plusieurs paramètres génériques formels, soit un identificateur (qui représentera un type) précédé du mot réservé **class**[1] et/ou tout simplement le nom d'un type (sans le mot **class** devant !) suivi d'un identificateur qui représente une constante du type en question, généralement appelé paramètre expression.

Exemple: **template** <**class** T, **int** i> **class** Exemple1…

Comme nous le rappel l'exemple ci-dessus, un générique (fonction ou classe) peut comporter plusieurs paramètres. Qu'ils soient d'une catégorie ou d'une autre, nous pouvons leurs assigner des valeurs par défaut. Il suffit de compléter la définition par:

- = *Nom_Du_Type_Par_Défaut* pour les paramètres de type

et par:

[1] Rappelons que le mot **class** peut être remplacé par **typename**!

- = *Constante_Du_Type* pour les paramètres expressions.

Exemple: **template** <**class** T = **int**, **int** i = 10> **class** Exemple1…

Relevons toutefois qu'à partir du moment où un paramètre possède une valeur par défaut, tous les suivants doivent aussi en posséder une.

Les paramètres de type peuvent correspondre à n'importe quel type simple ou structuré (classe, …) et pourquoi pas à une classe générique. Dans ce dernier cas, le paramètre doit être annoncé comme tel:

Exemple:

template <**class** T> **class** Exemple1… // cas non generique

template <**tempelate** <**class** U> **class** Exemple1, …> **class** Exemple2 … // cas generique

Pour les paramètres "expression", dans le contexte d'une classe, ils doivent impérativement correspondre à un type entier ou assimilable (simple, long ou court; signé ou non; caractère simple ou étendu), ou à un type pointeurs. Ce dernier cas nous permet, par exemple, de transmettre lors de l'instanciation l'adresse d'une fonction.

❑ *Corps de la classe*

Comme nous l'avons dit en guise d'introduction, puisque pour l'instant nous définissons les corps comme fonctions membres de la classe, il n'y a rien de bien particulier à signaler. Ils s'écrivent comme des fonctions membres usuelles.

Relevons tout de même que:

- Les paramètres génériques de types peuvent s'utiliser partout où l'on a le droit d'utiliser normalement un identificateur de type.
- Les paramètres génériques *expressions*, quant à eux se comportent au sein de la classe comme des constantes locales.

Instanciation

Pour les classes génériques, contrairement aux fonctions, il n'y a pas d'instanciation implicite. Nous serons toujours obligé de la faire explicitement, en transmettant les paramètres effectifs désirés, à moins que ceux-ci (ou certains d'entre-eux) possèdent des valeurs par défaut. Cette opération se réalise au moment de la déclaration d'un objet de la classe en question en complétant ce nom de classe entre < > par la liste des paramètres effectifs, séparés les uns des autres par des virgules.

Ainsi, avec la déclaration de classe que nous avons déjà utilisée:

template <**class** T = **int**, **int** i =10> **class** Exemple1…

nous pouvons déclarer, en admettant ici que la classe possède un constructeur par défaut:

1. Exemple1<**double**, 20> objet1;
2. Exemple1<**unsigned short int**> objet2;
3. Exemple1< > objet3;
4. Exemple1<vector<***int***> > objet4;

Ce dernier exemple nous montre que le paramètre effectif peut correspondre lui-même à l'instance d'une classe elle-même générique!

Dans l'exemple 2 ci-dessus, certains paramètres formels possèdent des valeurs par défaut, il n'est pas indispensable de donner les derniers paramètres effectifs si les valeurs par défaut nous conviennent.

Si comme dans l'exemple 3, si tous les paramètres formels possèdent des valeurs par défaut, nous pouvons très bien n'en fournir aucun au moment de l'instanciation, mais alors la présence des < > sans rien à l'intérieur est obligatoire, ceci pour bien montrer au compilateur qu'il s'agit d'une instance de classe.

Relevons encore que, pour les paramètres de la catégorie *expression*, seule une expression constante peut servir lors de l'instanciation.

Voilà, nous sommes prêts pour donner un exemple complet. Bien que nous sachions déjà que les outils existent dans la bibliothèque standard (*STL*), nous allons illustrer ceci en créant une nouvelle fois les bases d'une gestion de piles. Nous en donnerons par la suite plusieurs variantes permettant de montrer différents aspects théoriques.

```
#ifndef __PILE__
#define __PILE__
/*
  Gestion de base d'une pile statique
  PileStatique2a/Pile2.h
*/
template <class INFO> class Pile
{
   unsigned int sommet;
   const unsigned int taille;
   INFO * lesElements; // L'information

public:
  Pile ( unsigned int grandeur ) : taille ( grandeur )// Constructeur[1]
  {
    sommet = 0;
    lesElements = new INFO [taille];
  }

  ~Pile ( ) // Destructeur
  {
    delete [] lesElements;
  }

  /* Fonction d'insertion d'un element sur la pile  */
  void empiler ( const INFO & valeur )
  {
     /* Empiler si c'est possible! */
     if ( sommet < taille )
       *(lesElements + sommet++) = valeur;
    else // Erreur => exception
       throw "\n\nPile Pleine\n\n";
  } // empiler

  /* Fonction d'extraction d'un element de la pile  */
  INFO desempiler ( )
  {
     /* Desempiler si c'est possible! */
     if ( sommet )
       return *(lesElements + --sommet );
     else
       throw "\n\nPile vide\n\n";
  } // desempiler
```

[1] Rappel: cette forme d'initialisation est obligatoire pour un membre constant, et facultative pour les autres membres! Par contre cette forme présente l'avantage de pouvoir être utilisée quel que soit le type utilisé!

```
  /* Fonction pour determiner si la pile est vide   */
  bool pileVide ( )
  {
     return sommet == 0;
  } // pileVide
};
#endif
```

```
/*
  Programme exemple: Test des outils de gestion de pile statique.
  Lit une serie de valeurs et les affiche dans l'ordre inverse!
  Dans le cas present: par l'utilisation d'une pile
 PileStatique2a/TestPile
*/
#include "Pile2.h"
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  typedef int INFO;
  INFO valeur; // La valeur courante du traitement
  Pile<INFO> laPile (3);

  try
  {
    /* Traitez toutes les valeurs de l'utilisateur, jusqu'a un 0  */
    do
    {
      cout << "Donnez la valeur a inserer, 0 pour terminer : ";
      cin >> valeur;
      /* Empiler la valeur donnee */
      if ( valeur != 0 )
        laPile.empiler ( valeur );
    } while ( valeur != 0 );
    /* Afficher les valeurs dans l'ordre inverse */
    cout << "\n= Les valeurs dans l'ordre inverse =\n";
    /* Tant qu'il y a des elements sur la pile   */
    while ( !laPile.pileVide ( ) )
    {
      valeur = laPile.desempiler ( );
      cout << valeur << endl;
    }
  }
```

```
  catch ( const char * message )
  {
    cout << message;
  }

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Le résultat d'une exécution:

```
Donnez la valeur a inserer, 0 pour terminer : 1
Donnez la valeur a inserer, 0 pour terminer : 5
Donnez la valeur a inserer, 0 pour terminer : 2
Donnez la valeur a inserer, 0 pour terminer : 0

= Les valeurs dans l'ordre inverse =
2
5
1

Fin du programme...Appuyez sur une touche pour continuer...
```

❑ *Compilation séparée*

Pourquoi avons-nous mis tout (les corps compris) dans le fichier d'inclusion?

Tout d'abord, comme annoncé au début du chapitre, ceci n'est pas obligatoire, mais simplifie les écritures. Ensuite le compilateur a besoin des définitions (des corps) pour savoir quel code générer au moment d'une instanciation! Ceci veut malheureusement dire que nous devons fournir à nos futurs clients les sources de ces codes[1]!

Rien nous interdit de définir ces corps dans un autre fichier! Toutefois, faudra-t-il lui donner l'extension ".cpp", puisqu'il s'agit de définitions, ou ".h" puisque l'utilisateur devra inclure ce fichier dans son propre code? A vous de choisir (en général ".h")!

Dans le fichier de définitions, les notations vont se compliquer quelque peu! Comme pour les classes non génériques, il faut rappeler pour chaque fonction son appartenance à la classe, mais cette fois en complétant son nom par ses paramètres de généricité mis entre < >, plus, devant cela, le mot réservé template suivi entre < > des paramètres de généricité annoncés dans la classe. Ceci nous donnes, pour la classe dont la déclaration commencerait par:

```
template <class INFO> class Pile …
```

pour la définition de son constructeur par exemple:

```
template <class INFO> Pile<INFO>::Pile ( … )
```

En pratique cela revient au même que de dire que chaque fonction est déclarée comme génériques!

Oui, tout cela semble bien compliqué au niveau des notations, d'autant plus que nous pouvons légitimement nous demander si cela vaut la peine de procéder ainsi!

Voici la version "en compilation séparée", avec définition des fonctions membres hors de la classe:

[1] La directive **export** déjà brièvement introduite dans le chapitre des fonctions génériques devrai2t permettre de résoudre ce problème. Toutefois rappelons que, bien qu'il s'agisse d'une possibilité définie dans la norme, elle n'est pas encore supportée dans tous les environnements. Par contre, certains d'entre-eux offrent des possibilités spécifiques pour résoudre ce problème. Nous ne traiterons pas de ces aspects dans le cadre de ce cours!

```
#ifndef __PILE__
#define __PILE__
/*
  Gestion de base d'une pile statique
  PileStatique2/Pile2.h
*/
template <class INFO> class Pile
{
   unsigned int sommet;
   const unsigned int taille;
   INFO * lesElements; // L'information

public:
  Pile ( unsigned int grandeur = 20 );  // Constructeur
  ~Pile ( ); // Destructeur

  /* Fonction d'insertion d'un element sur la pile             */
  void empiler ( const INFO &valeur );

  /* Fonction d'extraction de l'element au sommet de la pile   */
  INFO desempiler ( );

  /* Fonction pour determiner si la pile est vide              */
  bool pileVide ( );
  /* ... dans la realite, il y en aurait certainement d'autres */
};
#endif
```

```
/*
  Gestion de base d'une pile statique
  PileStatique/PILE.cpp
*/
#include "Pile2.h"
using namespace std;

template <class INFO> Pile<INFO>::Pile ( unsigned int grandeur ):
         taille ( grandeur ) // Constructeur
{
  sommet = 0;
  lesElements = new INFO [taille];
}

template <class INFO> Pile<INFO>::~Pile ( ) // Destructeur
{
  delete [] lesElements;
}

/* Fonction d'insertion d'un element sur la pile  */
```

```
template <class INFO> void Pile<INFO>::empiler ( const INFO & valeur )
{
   /* Empiler si c'est possible! */
   if ( sommet < taille )
     *(lesElements + sommet++) = valeur;
   else // Erreur => exception
     throw "\n\nPile Pleine\n\n";
} // empiler


/* Fonction d'extraction d'un element de la pile  */
template <class INFO> INFO Pile<INFO>::desempiler ( )
{
   /* Desempiler si c'est possible! */
   if ( sommet )
     return *(lesElements + --sommet );
   else
     throw "\n\nPile vide\n\n";
} // desempiler


/* Fonction pour determiner si la pile est vide   */
template <class INFO> bool Pile<INFO>::pileVide ( )
{
   return sommet == 0;
} // pile_vide

/* ... dans la realite, il y en aurait certainement d'autres */
```

Pour le programme de test, mis à part le fait qu'il faudra lui inclure non seulement le fichier des déclarations (le ".h" habituel) mais aussi celui des définitions des fonctions, il ne change pas. Nous ne redonnons donc pas son contenu ici!

Les résultats restent bien évidemment identiques à ceux de la première version.

❑ *Variante*

Dans les 2 versions qui précèdent nous avons laissé à la charge du constructeur de fixer la taille des piles. Pour illustrer les diverses possibilités, nous vous proposons ci-dessous une variante où l'on impose la taille par l'intermédiaire d'un paramètre générique *expression*. Cette méthode présente aussi l'avantage de disposer, dans la classe, directement de la valeur sans avoir à la conserver dans une constante membre. Nous profiterons aussi de cette version pour montrer l'utilisation des valeurs par défaut pour les paramètres génériques.

Voici cette version adaptée:

```
#ifndef __PILE__
#define __PILE__
/*
  Gestion de base d'une pile statique
  PileStatique2c/Pile2.h
*/
template <class INFO = int, unsigned int grandeur = 20> class Pile
{
   unsigned int sommet;
   INFO * lesElements; // L'information

public:
  Pile ( ) // Constructeur
  {
    sommet = 0;
    lesElements = new INFO [grandeur];
  }

  ~Pile ( ) // Destructeur
  {
    delete [] lesElements;
  }
```

```
  /* Fonction d'insertion d'un element sur la pile  */
  void empiler ( const INFO & valeur )
  {
     /* Empiler si c'est possible! */
     if ( sommet < grandeur )
       *(lesElements + sommet++) = valeur;
     else // Erreur => exception
       throw "\n\nPile Pleine\n\n";
  } // empiler



  /* Fonction d'extraction d'un element de la pile  */
  INFO desempiler ( )
  {
     /* Desempiler si c'est possible! */
     if ( sommet )
       return *(lesElements + --sommet );
     else
       throw "\n\nPile vide\n\n";
  } // desempiler



  /* Fonction pour determiner si la pile est vide   */
   bool pileVide ( )
  {
     return sommet == 0;
  } // pileVide
};
#endif



/*
  Programme exemple: Test des outils de gestion de pile statique.
  Lit une serie de valeurs et les affiche dans l'ordre inverse!
  Dans le cas present: par l'utilisation d'une pile
 PileStatique2c/TestPile
*/
#include "Pile2.h"
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  typedef int INFO;
  INFO valeur; // La valeur courante du traitement
  Pile<> laPile;
```

```
  try
  {
    /* Traitez toutes les valeurs de l'utilisateur, jusqu'a un 0  */
    do
    {
      cout << "Donnez la valeur a inserer, 0 pour terminer : ";
      cin >> valeur;
      /* Empiler la valeur donnee */
      if ( valeur != 0 )
        laPile.empiler ( valeur );
    } while ( valeur != 0 );
    /* Afficher les valeurs dans l'ordre inverse */
    cout << "\n= Les valeurs dans l'ordre inverse =\n";
    /* Tant qu'il y a des elements sur la pile   */
    while ( !laPile.pileVide ( ) )
    {
      valeur = laPile.desempiler ( );
      cout << valeur << endl;
    }
  }
  catch ( const char * message )
  {
    cout << message;
  }

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Relevez particulièrement, dans le programme de test, la déclaration de la variable *laPile* qui utilise toutes les valeurs par défaut des paramètres formels:

```
Pile<> laPile;
```

Spécialisation

Tout comme nous l'avions vu pour les fonctions, nous pouvons aussi spécialiser les classes, ou certaines fonctions d'une classe générique. Sans entrer dans le détail, nous allons donner maintenant quelques indications de base sur la manière de procéder pour réaliser cette opération.

Spécialisation d'une fonction

Mettons nous dans la situation relativement simple où nous définissons la spécialisation d'une fonction particulière dans le même fichier source que la déclaration de la classe. Il faudra préciser pour la ligne d'en-tête de la spécialisation de la fonction, entre le type de son résultat et son nom, le nom de la classe à laquelle elle se rapporte, suivi entre < > du (des) paramètre(s) et finalement l'opérateur de résolution de portée "::".

Voici les grandes lignes de ce que cela donnerait, si dans la première version de notre pile dans ce chapitre, nous décidions par exemple de spécialiser la fonction *empiler* pour le type **double**:

```
#ifndef __PILE__
#define __PILE__
template <class INFO> class Pile
{
 /* Contenu strictement identique à ce que nous avions defini */
 /*  dans la premiere version de ce chapitre pour notre pile  */
};



/* A titre d'exemple le squelette de la spécialisation de  */
/*  la fonction empiler pour pour le type double           */
void Pile<double>::empiler ( const double & valeur )
{
   /* Son nouveau corps ... */
} // empiler

#endif
```

Spécialisation de toute la classe

Dans un tel cas, nous donnerons une deuxième définition complète de la classe, dont l'entête prendra la forme générale suivante:

1. Le mot **template**.
2. Des < > vides de contenu.
3. Le mot **class** suivi du nom de la classe à spécialiser.
4. Entre < > le ou les paramètre(s) de la spécialisation.

Restons, à titre d'illustration, dans la situation de notre pile en version initiale, mais en admettant cette fois que nous voulons spécialiser toute la classe (donc toutes ces fonctions) pour le type **double**. Le squelette de cette nouvelle version deviendrait:

```
#ifndef __PILE__
#define __PILE__
template <class INFO> class Pile
{
 /* Contenu strictement identique à ce que nous avions defini */
 /*  dans la premiere version de ce chapitre pour notre pile  */
};


template <> class Pile<double>
{
 /* Le contenu complet de la classe adapte au type double */
};
#endif
```

Notes complémentaires:

- Nous ne le développerons pas plus ici, mais signalons qu'une classe générique peut posséder des fonctions amies!
- Nous ne pouvons pas surcharger des classes génériques, comme c'était le cas pour les fonctions!

Notion d'itérateurs et d'algorithmes

Introduction

Dans le chapitre qui précède nous avons introduit des éléments de la bibliothèque qui ne font pas partie du groupe des conteneurs ou, tout au moins pas complètement pour certains d'entre-eux, car ils n'en possèdent qu'une partie des caractéristiques.

Ce chapitre va servir de charnière, permettant d'introduire des notions qui nous seront utiles pour présenter les conteneurs.

Les conteneurs permettent pour l'essentiel de travailler de manière homogène quelque soit la structure utilisée. Ceci signifie qu'en règle générale les mêmes opérations peuvent se réaliser sur des conteneurs de nature différente. Toutefois suivant le genre de conteneur utilisé certaines opérations se révèleront plus ou moins efficaces. Le problème ne sera pas de savoir comment s'utilise tel conteneur par rapport à tel autre, mais lequel utiliser pour obtenir les meilleurs résultats dans mon application!

La norme n'impose aucune contrainte sur l'implémentation des structures sous jacentes mais uniquement sur les performances de telle ou telle opération en fonction du type de conteneur.

Les conteneurs se divisent en 2 grandes catégories:

- Les séquentiels, dont le contenu est organisé physiquement selon les désires du développeur.
- Les associatifs dont le contenu physique est organisé selon une clé logique.

Les séquentiels comprennent les classes:

- vector[1]
- list
- deque

[1] Que nous avons déjà quelque peu abordée à titre d'illustration dans le cadre du chapitre de la présentation des tableaux puisqu'ils en sont en quelque sorte une généralisation.

Il existe aussi des spécialisations dont nous reparlerons dont nous reparlerons plus tard:

- stack, queue, priority_queue9.

Les associatifs quant à eux comportent essentiellement les classes:

- map
- multimap
- set
- multiset

Cependant, avant d'aborder ces éléments, il existe des concepts généraux qu'il nous faut introduire au préalable de manière globale et même pour certains d'entre-eux en se basant sur des notions déjà connues.

Il s'agit essentiellement des notions:

- d'algorithmes.
- d'itérateurs.

❑ *La notion d'algorithmes*

Cette partie de cours vise simplement à vous donner quelques idées élémentaires sur le principe des algorithmes, ainsi qu'à fournir des exemples basés sur la structure classique de tableaux que nous connaissons bien. Ces exemples nous les généraliserons aux conteneurs immédiatement après.

La notion d'algorithme représente une des forces de la bibliothèque *STL*[1] car elle permet de traiter de manière homogène des structures de données de natures différentes.

[1] **S**tandard **T**emplate **L**ibrary

Pour utiliser ces algorithmes, il faut inclure le fichier *algorithme* et pour certains d'entre eux le fichier *numeric*. Les algorithmes y sont définis sous forme de fonctions génériques qui s'adapteront donc aux types des effectifs paramètres transmis lors de l'appel.

Donnons immédiatement un exemple et nous décrirons les algorithmes utilisés ensuite.

```
/*
  Introduction à la notion d'algorithme
    ALGORITHMES1.cpp
*/
#include <cstdlib>
#include <iostream>
#include <algorithm>
#include <numeric>
using namespace std;

/* Affichage d'un petit tableau */
void affiche ( int tab [], int nb )
{
  for ( int i = 0; i < nb; i++ )
    cout << tab [ i ] << '\t';
  cout << endl;
}

int  main ( )
{
  int tab [] = { 3, 2, -5, 2, 4 };
  static int tab2 [5];
  cout << "Nos premiers algorithmes!\n\n";
  cout << "Le tableau \"tab\":\n";
  affiche ( tab, 5 );
  cout << "Somme des elements: "
       << accumulate ( tab, tab + 5, 0 ) << endl;
  sort ( tab, tab + 5 );
  cout << "Le tableau \"tab\" apres tri:\n";
  affiche ( tab, 5 );
  cout << "La plus petite valeur: "
       << *min_element ( tab, tab + 5 ) << endl;
  cout << "Le tableau \"tab2\":\n";
  affiche ( tab2, 5 );
  copy ( tab, tab + 5, tab2 );
  cout << "Le tableau \"tab2 apres copie de tab1\":\n";
  affiche ( tab2, 5 );
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution nous fournit les résultats suivants:

```
Nos premiers algorithmes!

Le tableau "tab":
3       2       -5      2       4
Somme des elements: 6
Le tableau "tab" apres tri:
-5      2       2       3       4
La plus petite valeur: -5
Le tableau "tab2":
0       0       0       0       0
Le tableau "tab2 apres copie de tab1":
-5      2       2       3       4
Appuyez sur une touche pour continuer...
```

Comme nous l'avions annoncé, notre premier exemple se base sur une structure connue: un tableau d'entiers. Toutefois les éléments du tableau pourraient être de n'importe quel type[1] et l'on remplacera le tableau par d'autres structures dans la suite!

D'une manière générale un algorithme s'applique à une suite de données d'un certain type. Si, dans l'exemple ci-dessus, nous les avons appliqués à tout un tableau, nous aurions aussi pu ne traiter qu'une partie (contiguë) du tableau.

Le premier algorithme utilisé dans l'exemple permet de faire la somme tous les éléments à traiter; elle livre un résultat du type de ces éléments:

```
accumulate ( adresseDebut, adresseFin, valeurInitiale )
```

Il nous vient de *numeric* et donc en principe ne s'applique qu'à des éléments de types numériques[2]. Ces paramètres:

adresseDebut: l'adresse du début de la zone à traiter[3].

adresseFin: l'adresse de la fin de la zone à traiter. **Attention** cette adresse représente bien la fin de la zone, c'est-à-dire l'adresse qui vient **après** le dernier élément à traiter.

[1] A la condition toutefois pour la majorité des algorithmes que les opérateurs de comparaison (== et <) soient définis pour le type en question.

[2] Toutefois, pour autant que l'opérateur "+" soit défini sur le type des éléments, nous pouvons envisager son utilisation!

[3] Ici un pointeur que l'on généralisera prochainement à la notion d'itérateur.

valeurInitiale: une valeur du type des éléments à traiter qui sert à initialiser la somme, généralement 0 pour des valeurs numériques.

Le deuxième algorithme utilisé permet lui de trier les éléments d'une structure de données.

La fonction *sort* vient de *algorithme*, mais pour être utilisée il faut respecter quelques contraintes supplémentaires, à savoir:

- Il faut disposer d'une structure d'ordre partiel sur les éléments, c'est-à-dire que l'opérateur "<" soit défini sur le type des éléments.
- La structure de données doit permettre un accès direct aux éléments, ce qui est le cas pour les tableaux.

La fonction *sort* ne livre pas de résultat, son rôle consistant à trier les éléments de la structure.

La forme générale d'un appel à cet algorithme:

```
sort ( adresseDebut, adresseFin );
```

avec les mêmes considérations pour ses 2 paramètres que pour les 2 premiers de la fonction accumulate!

Note: avouez tout de même que cette formulation se révèle plus simple et plus agréable d'utilisation que celle que nous avions présentée dans le cadre de la bibliothèque C. Elle est rendue possible par la généricité qui permet à l'algorithme de s'adapter à la taille des objets désignés par les pointeurs.

Le troisième algorithme utilisé permet de retrouver la plus petite valeur de la structure traitée. Ses 2 paramètres ont encore une fois la même signification et il livre comme résultat une référence sur la valeur trouvée[1]. Sa forme générale:

```
min_element ( adresseDebut, adresseFin )
```

Evidemment il existe une fonction équivalente pour trouver la plus grande valeur de la structure:

```
max_element ( adresseDebut, adresseFin )
```

Le dernier algorithme utilisé dans notre exemple nous permet de copier les éléments d'une structure de données dans une autre. Ici nous avons copié les éléments d'un tableau dans ceux d'un autre tableau de même type. Toutefois dans la suite nous pourrons copier d'un conteneur vers un autre de nature différente[2], pour autant que le type des éléments soit identique.

[1] Ici sous la forme d'un pointeur, plus généralement dans la suite sous celle d'un itérateur.

[2] Ou d'un tableau vers un conteneur et inversement.

La forme générale de cette fonction:

```
copy ( debutSource, finSource, debutDestination );
```

Comme au préalable *debutSource* et *finSource* permettent de délimiter la zone à copier, alors que *debutDestination* fixe évidemment le début de la zone où se fait la copie. La fonction livre en retour l'adresse de la fin de la zone de copie, ce qui permet éventuellement d'enchaîner/imbriquer les opérations.

Voilà, cela termine notre introduction aux algorithmes, il en existe encore bien d'autres dont une partie vous sera présentée par la suite.

Nous allons maintenant généraliser leur utilisation en introduisant une autre idée fondamentale de la bibliothèque standard: le principe des itérateurs.

❑ *La notion d'itérateurs*

Pour la suite et puisque nous connaissons déjà quelque peu la classe *vector*, admettons que nous avons déclaré par exemple:

```
vector<int> vecteur (5);
```

Tous les algorithmes se basent en fait sur le principe d'itérateur. Un itérateur consiste simplement en une généralisation de la notion de pointeur. C'est pour cela que nous avons déjà pu utiliser des algorithmes sur les tableaux. Rappelons qu'en C le nom d'un tableau ne représente rien d'autre qu'une adresse, donc un pointeur, donc un itérateur.

Chaque classe *conteneur* met à disposition un type *itérator*. Comme nous basons nos exemples sur la classe *vector* (d'entiers), nous pouvons nous déclarer une variable de ce type de la manière suivante:

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

Si l'itérateur doit désigner un objet ne devant pas changer de valeur, nous utiliserons la notation:

```
vector<int>::const_iterator indice;
```

Un itérateur désignera donc un élément de la structure. Etant assimilable à un pointeur, nous accédons à l'élément désigné par l'opérateur usuel: *:

```
*indice = 12;
```

Toutes les classes concernées mettent également à disposition 2 fonctions:

- `begin ( )` qui livre un itérateur désignant le premier élément de la structure:

```
indice = vecteur.begin ( );
```

- `end ( )` qui livre un itérateur pointant sur l'adresse de l'élément qui **suit** le dernier de la stucture[1].

Les classes en question fournissant aussi l'opérateur ++ sur les itérateurs, nous pouvons facilement construire une boucle nous permettant de parcourir tous les éléments d'un objet de la classe:

```
for ( vector<int>::iterator i = vecteur.begin();
          i != vecteur.end(); i++ ) ...
```

Relevez l'utilisation de la condition: `i != vecteur.end()`! Nous nous arrêtons dès que nous nous trouvons après le dernier élément de la structure.

Les opérations que nous venons de décrire sont valables pour tous les conteneurs sans exception. Toutefois certains d'entre-eux, tel *vector* qui nous intéresse actuellement, possèdent des propriétés spécifiques[2] et offrent d'autres possibilité de traitement. A savoir:

- Si le conteneur offre un accès bidirectionnel:
-

C'est-à-dire si la structure permet non seulement d'être parcourue du début vers la fin, mais aussi de la fin vers le début. En d'autres termes, si elle permet à partir de la position d'un élément de revenir à l'élément qui le précède. Dans ce cas elle dispose également de l'opérateur -- sur ses itérateurs.

[1] Il n'existe donc pas, il faut le considérer comme un élément "virtuel"!

[2] Nous préciserons pour chaque conteneur par la suite!

Toutefois cette possibilité pose un nouveau problème puisque *end ()* ne désigne pas le dernier élément, mais l'élément virtuel qui suit! Ces conteneurs nous offrent un deuxième type d'itérateur: *reverse_iterator* qui nous permet de parcourir la structure en sens inverse:

```
vector<int>::reverse_iterator indice;
```

ou, si l'objet ne doit pas être modifié:

```
vector<int>::const_reverse_iterator indice;
```

Les fonctions suivantes livrent un résultat de ce type:

- `rend ( )` livre un itérateur pointant sur le dernier élément de la structure.
- `rbegin ( )` qui livre un itérateur désignant l'élément "virtuel" qui précède le premier de la structure.

La boucle permettant de parcourir en sens inverse tous les éléments de la structure devient:

```
for ( vector<int>::reverse_iterator i = vect.rbegin();
     i != vect.rend (); i++) ...
```

Bien que cela soit peu joli, mais pouvant aider à la compréhension, la boucles ci-dessus pourrait s'écrire:

```
vector<int>::iterator fin = vect.begin () - 1;
for ( vector<int>::iterator i = vect.end () - 1;
     i != fin; i-- )
```

Ceci nous montre que pour un itérateur bidirectionnel, nous pouvons lui appliquer la même arithmétique que celle des pointeurs.

- Si le conteneur offre un accès direct, c'est le cas des *vector*:

Il se comporte alors comme un tableau et nous pouvons accéder à ses éléments par l'intermédiaire de l'opérateur [ ]:

```
vecteur [3] = 12;
```

La boucle pour parcourir l'ensemble des éléments de la structure deviendrait simplement:

```
for ( int i = 0; i < vect.size (); i++ )
```

Quelques remarques complémentaires:

- Pour bien faire le lien avec le paragraphe qui précède, nous pouvons dire que les algorithmes ne s'appliquent pas à un conteneur, mais à une suite contiguë d'éléments d'un conteneur. Avec un itérateur désignant un élément, nous pouvons parcourir (traiter) une partie quelconque de conteneur. D'ailleurs, certaines méthodes et certains algorithmes livrent en retour un résultat de type itérateur.
- La taille des conteneurs que nous allons présenter dans la suite, et c'était déjà valable pour *vector*, peut varier dynamiquement. Cela implique qu'il ne faut pas tenter d'utiliser un itérateur dont la valeur a été fixée avant une modification des caractéristiques du conteneur (adjonction ou suppression d'éléments), avec des variantes suivant la nature du conteneur.
- Notons finalement que 2 itérateurs sur une même structure de données peuvent être comparés pour l'égalité (==) et le différent de (!=). Les autres opérateurs de comparaisons n'étant à priori pas disponibles, mais avec des variantes suivant le conteneur!

Comme d'habitude donnons maintenant un programme compilable utilisant certaines de ces possibilités. Il ressemble beaucoup au programme qui précède, mais adapté à la classe *vector* et aux itérateurs:

```
/*
  Introduction a la notion d'iterateur
    ITERATEURS.CPP
*/
#include <cstdlib>
#include <iostream>
#include <algorithm>
#include <numeric>
#include <vector>
using namespace std;

/* Affichage d'un petit vecteur */
void affiche ( vector<int> vect )
{
  for ( vector<int>::iterator i = vect.begin(); i != vect.end(); i++ )
    cout << *i << '\t';
  cout << endl;
}
```

```
int  main (  )
{
  int tab [] = { 3, 2, -5, 2, 4 };
  vector<int> vect (5);
  vector<int>::iterator indice;
  cout << "Iterateurs sur conteneurs!\n\n";
  cout << "Le vecteur \"vect\":\n";
  affiche ( vect );
  cout << "Le vecteur apres copie du tableau:\n";
  copy ( tab, tab + 5, vect.begin ());
  affiche ( vect );
  cout << "Somme des elements: "
       << accumulate ( vect.begin (), vect.end (), 0 ) << endl;
  sort ( vect.begin (), vect.end () );
  cout << "Le vecteur \"vect\" apres tri:\n";
  affiche ( vect );
  cout << "La plus petite valeur: "
       << *min_element ( vect.begin (), vect.end () ) << endl;

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

L'exécution nous fournit les résultats suivants:

```
Iterateur sur conteneurs!

Le vecteur "vect":
0       0       0       0       0
Le vecteur apres copie du tableau:
3       2       -5      2       4
Somme des elements: 6
Le vecteur "vect" apres tri:
-5      2       2       3       4
La plus petite valeur: -5
Appuyez sur une touche pour continuer...
```

Il y existe une hiérarchie parmi ces itérateurs. Un itérateur à accès direct possède toutes les propriétés d'un itérateur bidirectionnel qui lui-même possède toutes les propriétés d'un itérateur unidirectionnel.

En réalité il existe encore deux autres groupes d'itérateurs plus bas dans la hiérarchie, les itérateurs d'entrée et ceux de sortie. Comme leur nom le laisse supposer ces itérateurs sont normalement étroitement liés aux flux d'entées/sorties.

Les itérateurs de sortie

Il existe une classe générique: *ostream_iterator* qui met à disposition un constructeur prenant comme paramètre un flux de sortie. La généricité portant sur le type des éléments à envoyer sur ce flux[1]:

```
ostream_iterator<int> sortie ( cout );
```

Vous pouvez affecter une valeur à l'objet désigné par un itérateur de sortie, c'est même la seule opération raisonnable pour un tel itérateur:

```
*sortie = 12;
```

au lieu d'écrire:

```
cout << 12;
```

Par contre vous ne pouvez pas lire sa valeur, ce qui parait logique.

Bien que cela soit totalement inutile (sans effet) il est possible d'incrémenter un tel opérateur:

```
sortie++;
```

Si nous affectons une deuxième valeur entière à notre itérateur *sortie*, celle-ci sera accolée à la première ce qui ne sera certainement ni esthétique ni pratique. Pour résoudre ce problème il existe un deuxième constructeur de la classe *ostream_iterator* possédant 2 paramètres. Le premier correspond à nouveau à un flux de sortie et le deuxième à une chaîne de caractères à envoyer automatiquement sur le flux de sortie après chaque affectation faite par l'intermédiaire de l'itérateur.

Dans le programme exemple ci-dessus, le corps de la fonction pour afficher un objet de type *vector* d'entiers, pourrait alors prendre la forme:

[1] En réalité la classe possède 2 autres paramètres de généricité que nous ne détaillons pas plus ici car ils possèdent des valeurs par défaut qui nous conviennent. Signalons simplement que le deuxième permet d'utiliser un autre type de caractères que **char**.

```
void affiche ( vector<int> vect )
{
  ostream_iterator<int> sortie ( cout, "\t");
  for ( vector<int>::iterator i = vect.begin();
                              i != vect.end(); i++ )
    *sortie = *i;
  cout << endl;
}
```

Toutefois cette solution peut se simplifier en se rappelant que nous disposant d'un algorithme permettant de copier un conteneur (ou une partie de celui-ci) dans un autre conteneur. Alors pourquoi ne pas copier simplement notre *vector* sur la sortie:

```
void affiche ( vector<int> vect )
{
  ostream_iterator<int, char> sortie ( cout, "\t");
  copy ( vect.begin (), vect.end (), sortie );
  cout << endl;
}
```

Note: dans cette version le deuxième paramètre de généricité pour la déclaration de l'objet n'est pas indispensable puisque nous lui donnons la valeur qu'il a de toute façon par défaut. Nous ne l'avons mis qu'à titre d'illustration.

Itérateur d'entrée

Pour les entrées, de même que pour les sorties, il existe une classe générique: *istream_iterator* qui met à disposition un constructeur prenant comme paramètre un flux de d'entrée. La généricité portant sur le type des éléments à envoyer sur ce flux[1]:

```
istream_iterator<int> entree ( cin );
```

[1] En réalité la classe possède d'autres surcharges du constructeur.

Vous pouvez lire, obtenir (consulter) la valeur désignée par un itérateur d'entrée, c'est même la seule opération raisonnable pour un tel itérateur:

```
valeur = entree;
```

au lieu d'écrire:

```
cin >> valeur;
```

Par contre vous ne pouvez pas modifier la valeur désignée par l'itérateur, ce qui parait logique.

Bien que cela soit totalement inutile (sans effet) il est possible d'incémenter un tel opérateur:

```
entree++;
```

Cet itérateur livre une valeur spéciale s'il désigne une fin de fichier de telle sorte que cette valeur corresponde logiquement à la fin de structure.

Voici un exemple de code utilisant ces possibilités:

```
/*
  Introduction à la notion d'iterateur
    ITERATEURS2.cpp
*/
#include <cstdlib>
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

/* Affiche un vectreur une copie et un iterateur de sortie */
void affiche ( vector<int> vecteur )
{
  ostream_iterator<int> sortie ( cout, "\t");
  copy ( vecteur.begin (), vecteur.end (), sortie );
  cout << endl;
}
```

```
int  main (  )
{
  vector<int> vecteur;
  cout << "Iterateur d'entree!\n\n";

  /* Lecture des element du vecteur */
  copy ( istream_iterator<int> (cin),istream_iterator<int> (),
         back_inserter ( vecteur ) );

  cout << "Le vecteur apres lecture:\n";
  affiche ( vecteur );

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Les conteneurs

Introduction

Nous avons vu au chapitre qui précède que les conteneurs se divisent en 2 grandes catégories:

- Les séquentiels, dont le contenu est organisé physiquement selon les désires du développeur.
- Les associatifs dont le contenu physique est organisé selon une clé logique.

Les séquentiels comprennent les classes:

- vector Ils correspondent assez bien à la notion de tableaux à une dimension.
- list Ils correspondent assez bien à la notion de listes doublement chaînées.
- deque Un compromis entre *vector* et *list.*

Plus des spécialisations dites adaptateurs:

- stack (pile)
- queue (file/queue)
- priority_queue (file/queue de priorité)

Les associatifs quant à eux comportent essentiellement les classes:

- map
- multimap
- set
- multiset

Tous ces conteneurs représentent des classes génériques qu'il faudra instancier au minimum avec le type des éléments qu'ils devront contenir.

Dans le but d'uniformiser la syntaxe et donc de pouvoir facilement passer d'une forme de conteneur à l'autre, chacune des classes met à disposition des fonctions identiques, pour autant que la fonctionnalité soit supportée par la classe. Elles fournissent également différents types et constantes. Signalons pour l'instant le type *value_type*, qui en général correspond strictement au type des éléments du conteneur, donc au type pour lequel nous avons instancié le générique. Toutefois l'utilisation de *value_type* permettra d'obtenir des codes plus cohérents, lisibles et portables. Ils mettent également à disposition le type *reference* et *const_reference* qui permettent d'accéder aux éléments de la structure. Par la suite nous aurons l'occasion d'introduire d'autres de ces définitions "standards".

Tous les conteneurs[1] mettent à disposition un certain nombre de méthodes communes dont une grande partie a été introduite dans le chapitre consacré à la classe *vector*. En voici brièvement résumé la liste:

- *size_type size()* ***const****;* Livre la taille actuelle du conteneur.
- *size_type max_size()* ***const****;* Livre la taille maximale du conteneur.
- ***bool*** *empty ()* ***const****;* Permet de savoir si un conteneur est vide ou non.
- La méthode *insert* comporte plusieurs surcharges:

  iterator insert (iterator position, ***const*** *T& x);* Insère, à la *position*, un nouvel élément de valeur *x*. Livre en retour la position de cet élément.

  void *insert (iterator position, size_type n,* ***const*** *T& x);* Insère, à la *position*, un nombre égal à *n* d'éléments, tous de valeur *x*.

  template *<****class*** *InputIterator>*
  void *insert (iterator position, InputIterator first, InputIterator last);* Insère dans la structure, à la *position* les éléments compris entre *first* (inclus) et *last* (non inclus).

- **void** swap (conteneur<T,Allocator>& lst); Permet d'échanger le contenu de 2 conteneurs de même type. Méthode très efficace puisqu'elle ne fait qu'échanger les références aux structures. Vu ainsi la syntaxe semble bien compliquée, pourtant il suffit de lui passer comme paramètre un conteneur du même type que celui auquel on applique la méthode.
- *iterator erase (iterator position);* et *iterator erase (iterator first, iterator last);* Permettent d'effacer respectivement l'élément à la *position* ou la série d'éléments compris entre *first* et *last*. Livre en retour une référence sur l'élément qui suit le dernier effacé.

[1] A l'exception de *bitset* déjà traité et que nous n'avons d'ailleurs pas mentionné dans cette liste car il joue un rôle et possède des particularités très spécifiques!

- ***void*** *clear ();* Permet de vider complètement la structure de toutes ses données.
- *iterator begin (); et const_iterator begin () const;* Itérateur livrant la référence au premier élément de la structure.
- *iterator end (); et const_iterator end () const;* Itérateur livrant la référence à la position qui suit le dernier élément de la structure.
- *iterator rbegin (); et const_iterator rbegin () const;* Itérateur inverse livrant la référence du dernier élément de la structure.
- *iterator rend (); et const_iterator rend () const;* Itérateur inverse livrant la référence au pseudo élément qui précède le premier de la structure.
- = et != Les opérateurs de comparaison.

Les conteneurs séquentiels

Les fonctionnalités communes

En plus des fonctionnalités déjà citées et communes à tous les conteneurs, la famille des conteneurs séquentiels possèdes aussi un groupe de méthodes communes.

- Les constructeurs dont le plus simple n'a pas de paramètre[1] et permet de construire un objet "vide"! Un deuxième constructeur possède 2 paramètres (donc 3 en réalité!), le premier de type *size_type*, permet de fixer la taille initiale du conteneur et le deuxième la valeur des éléments (identique pour chacun). Le type du deuxième de ses paramètres dépend du type des éléments du conteneur. Il possède une valeur par défaut qui correspond à la valeur livrée par le constructeur par défaut de l'objet, il n'est donc pas indispensable de fournir ce paramètre.[2]

- ***void*** *resize (size_type sz, T c = T());* Permet, par son premier paramètre, de fixer une nouvelle taille au conteneur auquel on l'applique. Si cette taille se révèle plus petite que l'ancienne, les éléments en plus sont perdus. Si elle est plus grande les nouveaux éléments sont initialisés avec la valeur du deuxième paramètre si présent et, sinon, avec le constructeur par défaut du type des éléments.

- ***void*** *assign (size_type n,* ***void*** *T& u);* Détruit l'ancien contenu du conteneur et le remplace par *n* fois la valeur du deuxième paramètre. Il existe une surcharge de cette méthode prenant 2 paramètres de type *iterator* et permettant d'initialiser le nouveau contenu avec une séquence d'un autre conteneur, pas nécessairement du même type, mais dont les éléments eux sont du même type.
- *reference back ()*; et *const_reference back ()* ***const****;* livrent une référence sur le dernier élément de la structure, sans en modifier le contenu. Ces références peuvent s'utiliser pour obtenir la valeur de l'élément dans les 2 cas et pour modifier cette valeur pour la première des méthodes.

- *reference front ();* et ***const****_reference front ()* ***const****;* livrent une référence sur le

[1] En réalité tous les constructeurs possèdent un paramètre de plus (un allocateur) possédant une valeur par défaut convenant fort bien dans la majorité des cas!

[2] D'autres constructeurs, non décrits ici, existent!

premier élément de la structure, sans en modifier le contenu. Ces références peuvent s'utiliser pour obtenir la valeur de l'élément dans les 2 cas et pour modifier cette valeur pour la première des méthodes.

- ***void*** *push_back (* ***const*** *T& x);* Ajoute à la fin du conteneur un élément de valeur *x*.

- ***void*** *pop_back ();* Enlève l'élément se trouvant à la fin du conteneur. Relevez que la méthode ne livre pas la valeur extraite. C'est au programmeur, s'il en a besoin d'aller la chercher au préalable par un appel à *back ()*.

Nous avons maintenant fait le tour des fonctionnalités communes à tous les conteneurs séquentiels.

Voici un petit exemple illustrant quelques-unes de ces possibilités générales. Nous le basons sur la classe *vector* puisque nous la connaissons déjà, mais il suffit de remplacer partout *vector* par *list* ou *deque* et cela continue à fonctionner de la même manière:

```
/*
  Operations communes aux conteneurs sequentiels
    Conteneur1.cpp
*/
#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;

/* Affichage d'un vecteur */
void affiche ( const vector<int> v )
{
  for ( vector<int>:: const_iterator pt = v.begin(); pt != v.end();
                                                          ++pt )
    cout << " " << *pt;
  cout << endl;
}
```

```
int main ()
{
  vector<int> v1;
  vector<int> v2;

  cout << "Programme exemple conteneur1\n\n";
  v1.assign ( 5, 11 );
  v1.push_back ( 0 );

  v2.assign ( v1.begin(), v1.end() );
  affiche ( v2 );

  int tab[]={ -2, 7, 3, 1 };
  v1.assign ( tab, tab + 4 );
  cout << "Element a extraire: " << v1.back () << endl;
  v1.pop_back ();
  affiche ( v1 );

  cout << "Taille de v1: " << v1.size() << endl;
  cout << "Taille de v2: " << v2.size() << endl;

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Il affiche lors de son exécution:

```
Programme exemple conteneur1

 11 11 11 11 11 0
Element a extraire: 1
 -2 7 3
Taille de v1: 3
Taille de v2: 6
Appuyez sur une touche pour continuer...
```

La classe vector

Nous commencerons par la classe *vector* puisque nous la connaissons déjà et que nous ne nous y attarderons pas! Rappelons que nous disposons là d'une structure proche de celle des tableaux, ce qui signifie entre autre que nous dia posons de l'accès direct à ces éléments par l'intermédiaire de l'opérateur [ ] ou de la méthode *at* qui elle réalise un contrôle d'accès et lève l'exception *out_of_range* en cas de violation. Ces accès ont une complexité constante (*0(1)*) quelque soit le nombre d'éléments de la structure. Pour cette classe, du point de vue performances, ces accès directs se révèlent préférables à l'utilisation d'itérareurs. Toutefois ces itérateurs offrent aussi un accès direct, ce qui signifie que l'on peut écrire de expressions telles que*: iterateur + deplacement*!

L'insertion et la suppression en fin de structure sont également des opérations de complexité constante, d'où l'existence de méthodes le permettant de manière simplifiée: *push_back* et *pop_back*. Par contre la complexité de l'insertion ou de la suppression à un autre endroit est de complexité en *0(n)*, c'est-à-dire qu'elle croît proportionnellement au nombre d'éléments dans le conteneur.

Notez que le changement de taille, ou l'ajout d'un élément, rend non valables les valeurs que possédaient des itérateurs avant une telle opération. N'oubliez pas que toute la structure peut nécessiter un déplacement lors d'une telle opération.

Relevons encore l'existence des méthodes:

- *size_type capacity ()* ***const****;* Livre la place potentielle actuelle pour l'objet sans le déplacer. A ne pas confondre avec *size ()* (size <= capacity) ni avec *max_size ()* (max_size >= capacity)!
- ***void*** *reserve (size_type n);* Réserve de la place pour *n* objet dans le vecteur. Cette opération modifie le résultat de *capacity* mais pas de *size*! L'objectif ici consiste à améliorer les performances en évitant des déplacements ultérieurs puisque l'espace nécessaire a déjà été réservé.

Pour conclure, utilisez la classe *vector* si dans votre application vous devez réaliser beaucoup de consultations et relativement peu d'insertions ou de suppressions.

Voilà, nous n'en dirons pas plus sur la classe *vector*, ni ne donnerons un nouvel exemple de programme puisque nous l'avons déjà fait par le passé.

La classe list

Un objet de la classe *list* correspond essentiellement à la notion usuelle de liste chaînée. En fait il s'agit de listes doublement chaînées permettant un accès séquentiel avant ou arrière, mais **pas** un accès direct aux éléments de la structure. La classe convient fort bien pour tous les traitements séquentiels de données.

L'insertion et la suppression d'un élément en n'importe quel endroit de la structure se fait avec une complexité constante (*0(1)*). L'insertion d'un élément laisse tous les itérateurs valables et la suppression d'un élément ne rend non valables que les itérateurs désignant des éléments situés après dans la structure.

En plus des fonctionnalités valables pour tous les conteneurs et tous les conteneurs associatifs, la classe *list* met à disposition les fonctionnalités suivantes:

- **void** push_front (**const** T & x); Insère x comme premier élément du conteneur.
- **void** pop_front (); Retire le premier élément de la structure.
- **void** splice (iterator position, list<T,Allocator>& x);
  void splice (iterator position, list<T,Allocator>& x, iterator i);
  void splice (iterator position, list<T,Allocator>& x, iterator first, iterator last);
  Insère les éléments d'une liste ou d'une partie de liste dans une autre. De plus, extrait ces éléements de la liste source. Dans les 3 formes, le premier paramètre représente la position d'insertion dans la liste destination. Le deuxième paramètre représente la liste source.
  - Pour la première surcharge, toute la liste source est copiée dans destination. Source est donc vide après l'opération.
  - Pour la deuxième forme, seul l'élément désigné par l'itérateur fourni comme troisième paramètre est recopié dans destination et donc extrait de la source.
  - La troisième forme permet, sous la forme de 2 itérateurs, de spécifier l'intervalle des valeurs à copier depuis la source. Attention: la valeur désigné par l'térateur de début est copier dans la destination, alors que celle désignée pour la fin (quatrième paramètre!) ne l'est pas.
- **void** remove (**const** T & value); Enlève de la liste tous les éléments qui correspondent à la *valeur* spécifiée en paramètre.
- **template** <**class** Predicate> **void** remove_if (Predicate pred); Enlève de la liste tous les éléments pour lesquels le prédicat transmis en paramètre est vrai. Dans une première approche, disons qu'un prédicat correspond à une fonction booléenne qui possède un paramètre du type des éléments de la liste.[1]

[1] Il existe une autre possibilité sous forme de classe!

- **void** unique ();
  template <**class** BinaryPredicate> **void** unique (BinaryPredicate binary_pred);
  Permet de ne garder dans la liste qu'une occurrence d'une valeur donnée. La deuxième forme nous permet, en transmettant l'adresse d'un prédicat binaire de préciser ce que pour nous signifie "2 valeurs sont égales". Un prédicat binaire est à priori une fonction booléenne possédant 2 paramètres du type des éléments de la liste.
- **void** sort ();
  template <class Compare> **void** sort (Compare comp); Permet des trier les éléments de la liste à laquelle on applique la méthode. Pour les 2 formes, les mêmes remarques que ci-dessus pour la méthode *unique* sont valables.
- **void** reverse (); Inverse l'ordre des éléments du conteneur: le premier devient le dernier, le dernier le premier et ainsi de suite.
- **void** merge (list<T,Allocator>& x);
  template <**class** Compare> **void** merge (list<T,Allocator>& x, Compare comp);
  Permet de réaliser la fusion de 2 listes en conservant l'ordre des éléments. Nous ne préciserons pas plus ici.

Voici un exemple, artificiel, montrant l'utilisation de quelques-unes de ces possibilités:

```
/*
  Operations sur les listes
    Liste.cpp
*/
#include <iostream>
#include <list>
#include <string>
#include <cstdlib>
using namespace std;

/* Affichage d'une liste */
void affiche ( list<string> v )
{
  cout << '\t';
  for ( list<string>::iterator pt = v.begin(); pt != v.end(); ++pt )
    cout << " " << *pt;
  cout << endl;
}
```

```
/* Predicat, ici les noms commençant par la lettre "p" */
bool si ( const string & s )
{
  return s [0] == 'p' || s [0] == 'P';
}

/* Predicat binaire ici les noms commençant par la meme lettre */
bool compare ( const string & s1, const string & s2 )
{
  return tolower ( s1 [0] ) == tolower ( s2 [0] );
}


int main ()
{
  string travail [] = { "Pierre", "Francois", "Paul", "Vincent",
                        "Jean", "paul" };
  list<string> v1;

  cout << "Programme exemple de travail sur les listes\n\n";
  v1.assign ( travail, travail + 6 );
  cout << "Liste initiale: \n";
  affiche ( v1 );
  v1.push_front ( "Jean" );
  cout << "Apres insertion en tete: \n";
  affiche ( v1 );
  v1.reverse ();
  cout << "Apres inversion: \n";
  affiche ( v1 );
  v1.sort ();
  cout << "Apres tri simple: \n";
  affiche ( v1 );
  v1.sort ( compare );
  cout << "Apres tri force: \n";
  affiche ( v1 );
  v1.unique ();
  cout << "Apres rendue unique simple: \n";
  affiche ( v1 );
  v1.remove_if ( si );
  cout << "Apres remove_if: \n";
 affiche ( v1 );

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Les résultats affichés par ce programme:

```
Exemple de travail sur les listes

Liste initiale:
        Pierre Francois Paul Vincent Jean paul
Apres insertion en tete:
        Jean Pierre Francois Paul Vincent Jean paul
Apres inversion:
        paul Jean Vincent Paul Francois Pierre Jean
Apres tri simple:
        Francois Jean Jean Paul Pierre Vincent paul
Apres tri force:
       Francois Jean Jean paul Pierre Paul Vincent
Apres rendue unique simple:
        Francois Jean paul Pierre Paul Vincent
Apres remove_if:
        Francois Jean Vincent
Appuyez sur une touche pour continuer...
```

❑ La classe deque

Cette classe représente un compromis entre vecteurs et listes. En plus des opérations communes à tous les conteneurs et à tous les conteneurs séquentiels, elle partage:

- Avec la classe *vector* la méthode *at* et l'opérateur [ ]
- Avec la classe *list* les méthodes *push_front* et *pop_front*.

La classe *deque* ne possède aucune autre fonctionnalité spécifique.

Même si les opérations d'accès aux éléments restent de complexité *0(1)* comme pour les vecteurs, autrement dit, elles ne dépendent pas de la taille du conteneur, elles demeureront globalement moins performantes que la même opération réalisée sur un vecteur.

Nous ne donnerons pas plus de précisions sur cette classe.

Comme exemple de programme, vous pouvez reprendre ce que nous avions proposé pour la classe *vector* au moment où nous l'avions introduite et vous y remplacez simplement partout *vector* pas *deque*.

Les spécialisations

Nous avons déjà signalé qu'il existe 3 spécialisations de conteneurs séquentiels, que l'on appel des adaptateurs, à savoir:

- stack (pile)
- queue (file/queue)
- priority_queue (file/queue de priorité)

stack

Par définition d'une pile, les éléments nouveaux se mettent toujours au sommet de la pile et on les extrait aussi à partir du sommet dans l'ordre inverse de leur insertion. D'où l'abréviation LIFO (Last In First Out) donnée à une telle structure.

Nous connaissons cette classe pour l'avoir déjà introduite et utilisée. Rappelons:

- Pour l'utiliser il faut: `#include <stack>`
- La classe est générique, il faut l'instancier avec le type de ses éléments. La déclaration d'une variable se fait sous la forme: `stack<int> laPile;`
- De part le principe de son utilisations, les seules méthodes à disposition sont: *empty(), size (), top (), push (valeur), pop ()*. Pour les 2 premières elles fonctionnent de manière usuelle puisqu'elles sont communes à tous les conteneurs séquentiels. Quant à *top*, *push* et *pop* elles fonctionnent respectivement comme *back*, *push_back* et *pop_back*; seul le nom change!

Remarque complémentaire:

Par défaut cet adaptateur est construit sur la base de la classe *deque*. Toutefois, comme tous les conteneurs séquentiels offrent les fonctionnalités nécessaires, rien n'empêche de forcer l'adaptateur à se baser sur l'une des 2 autres classes, voir même sur une classe que nous aurions développée et qui offrirait les fonctionnalités nécessaires. Par exemple, pour baser notre pile d'entiers sur la classe *list*, nous déclarons:

```
stack<int, list<int> > laPile;
```

Attention:

1. N'oubliez pas alors d'ajouter un : `#include <list>`.
2. Si le type des éléments ne correspond pas à un type de base, mais à un type que vous vous êtes déclaré (**struct** par exemple), cette déclaration doit impérativement être globale.

Voilà, vous savez tout (ou presque!) sur l'adaptateur *stack*. Nous ne donnerons pas d'exemple puisque nous l'avons déjà fait dans le cadre du chapitre sur les tableaux, dans la première partie de ce cours.

❑ queue

Par définition d'une queue (file), les éléments nouveaux se mettent toujours à une extrémité (l'arrière) et on les extrait à partir de l'autre extrémité (le début). Cela correspond bien à une file d'attente usuelle. D'où l'abréviation FIFO (First In First Out) donnée à une telle structure.

Les règles d'utilisation:

- Pour l'utiliser il faut: `#include <queue>`
- La classe est générique, il faut l'instancier avec le type de ses éléments. La déclaration d'une variable se fait sous la forme: `queue<int> laQueue`;
- De part le principe de son utilisation, les seules méthodes à disposition sont: *empty(), size (), front (), back (), push (valeur), pop ()*. Pour les 2 premières elles fonctionnent de manière usuelle puisqu'elles sont communes à tous les conteneurs séquentiels. Les méthodes *front* et *back* permettent d'accéder respectivement au premier et au dernier élément du conteneur. Notez que vous pouvez ainsi modifier la valeur de chacun de ces 2 éléments! Quant à *push* et *pop,* elles fonctionnent respectivement comme *push_back* et *pop_back*; seul le nom change!

Remarque complémentaire:

Par défaut cet adaptateur est consstruit sur la base de la classe *deque*. Toutefois, le conteneur *list* offre aussi les fonctionnalités nécessaires, rien n'empêche de forcer l'adaptateur à se baser sur cette classe, voir même sur une classe que nous aurions développée et qui offrirait les fonctionnalités nécessaires. Par exemple, pour baser notre queue d'entiers sur la classe *list*, nous déclarons:

```
queue<int, list<int> > laQueue;
```

Attention:

1. N'oubliez pas alors d'ajouter un: *#include <list>*.
2. Si le type des éléments ne correspond pas à un type de base, mais à un type que vous vous êtes déclaré (**struct** par exemple), cette déclaration doit impérativement être globale.

Voilà, vous savez tout (ou presque!) sur l'adaptateur *queue*.

Voici un exemple d'utilisation d'une telle structure. Pour l'exemple nous construisons notre structure sur un type que nous définissons. Cette même forme sera reprise au paragraphe suivant pour présenter les queues de priorité, ce qui permettra ainsi de comparer les comportements.

Vous noterez que lors de la déclaration de l'objet *laQueue*, nous avons volontairement précisé, comme deuxième paramètre de généricité *deque<vector>*. Ceci n'est pas indispensable, puisque c'est ce qui se passe par défaut. Nous l'avons fait uniquement pour l'exemple.

Voici ce code:

```
/*
   Adaptateur queue
    Queue.cpp
*/
#include <iostream>
#include <queue>
#include <cstdlib>
using namespace std;
struct vecteur
{  int x, y; };
```

```
int main ()
{
  queue<vecteur, deque<vecteur> > laQueue;
  vecteur v;  // Valeur courante
  char operation;

  cout << "Programme exemple getion d'une queue\n\n";
  /* Tant que l'utilisateur le veut ... */
  do
  {
    cout << "Choisissez:\n\ti: inserer\te: extraire\tq: quitter\n"
         << "Quelle operation: ";
    cin >> operation;
    /* En fonction du choix... */
    switch ( operation )
    {
      case 'i': case 'I':  // Inserer un nouvel element
        cout << "Coordonnee X: "; cin >> v.x;
        cout << "Coordonnee Y: "; cin >> v.y;
        laQueue.push ( v ); break;
      case 'e': case 'E':  // Extraire l'element en tete
        if ( !laQueue.empty() )
        { /* Queue non vide => operation possible! */
          v = laQueue.front ();
          cout << "Valeur extraite: X= " << v.x << ", Y= "
                                         << v.y << endl;
          laQueue.pop();
        }
        else
          cout << "La queue est vide!\n";
        break;
      case 'q': case 'Q': break;  // C'est fini
      default:
        cout << "***Erreur, recommencez!\n";
    }
  } while ( !( operation == 'q' || operation == 'Q' ) );

  return EXIT_SUCCESS;
}
```

A titre d'exemple, une exécution de se programme peut livrer comme résultat:

```
Programme exemple getion d'une queue

Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: i
Coordonnee X: 1
Coordonnee Y: 3
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: i
Coordonnee X: 2
Coordonnee Y: 2
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: i
Coordonnee X: 4
Coordonnee Y: 4
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: e
Valeur extraite: X= 1, Y= 3
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: e
Valeur extraite: X= 2, Y= 2
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: e
Valeur extraite: X= 4, Y= 4
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: e
La queue est vide!
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: q
```

priority_queue

Pour une queue de priorité (file de priorité), les éléments nouveaux sont introduits dans la structure au bon endroit en fonction de leur priorité. On les extrait à partir du début, de telle sorte que ce soit toujours l'élément le plus prioritaire qui soit pris. Les règles d'utilisation:

- Pour l'utiliser il faut aussi: `#include <queue>`
- La classe est générique, il faut l'instancier au minimum avec le type de ses éléments. Pour les types de base (pour les autres type nous fournirons des indications complémentaires après l'exemple de programme!), où nous pouvons garder toutes les valeurs par défaut, la déclaration d'une variable se fait sous la forme: *priority_queue*<***int***> *laQueue*;
- De part le principe de son utilisations, les seules méthodes à disposition sont: *empty(), size (), top (), push (valeur), pop ()*. Pour les 2 premières elles fonctionnent de manière usuelle puisqu'elles sont communes à tous les conteneurs séquentiels. La méthode *top* permet d'accèder à l'élément le plus prioritaire du conteneur. Quant à *push* et *pop*, elles permettent respectivement d'introduire, au bon endroit dans la structure, un nouvel élément et d'extraire de la structure l'élément le plus prioritaire! Notez que l'on ne sait pas ce qui se passe lorsque 2 éléments ont la même priorité!

Remarque complémentaire:

Par défaut cet adaptateur est consstruit sur la base de la classe *vector*. Toutefois, le conteneur *deque* offre aussi les fonctionalités nécessaires, rien n'empêche de forcer l'adaptateur à se baser sur cette classe, voir même sur une classe que nous aurions développée et qui offrirait les fonctionnalités nécessaires[1]. Par exemple, pour baser notre queue d'entiers sur la classe *deque*, nous déclarerons:

```
priority_queue<int, deque<int> > laQueue;
```

[1] Entre autre l'accès direct aux éléments!

Attention:

- N'oubliez pas alors d'ajouter un: #include <*deque*>.
- Si le type des éléments ne correspond pas à un type de base, mais à un type que vous vous êtes déclaré (**struct** par exemple), cette déclaration doit impérativement être globale.

Voilà, vous savez tout (ou presque!) sur l'adaptateur *queue*.

Voici un exemple d'utilisation d'une telle structure. Nous le basons sur la même structure que pour la queue "normale" et nous donnerons comme exemple d'exécution les même valeurs, afin de pouvoir étudier les différences de comportement.

Voici ce code:

```
/*
  Adaptateur queue de priorite
    QueuePriorite.cpp
*/
#include <iostream>
#include <queue>
#include <deque>
#include <cstdlib>
using namespace std;

struct vecteur
{  int x, y;
  bool operator () ( const vecteur & v1, const vecteur & v2 )
  {
    return v1.x*v1.x + v1.y*v1.y < v2.x*v2.x + v2.y*v2.y;
  }
};
```

```
//class compare
//{ public:
//    bool operator () ( const vecteur & v1, const vecteur & v2 )
//    {  return v1.x*v1.x + v1.y*v1.y < v2.x*v2.x + v2.y*v2.y;
//    }
//};


int main ()
{
  priority_queue<vecteur, deque<vecteur>, vecteur > laQueue;
//  priority_queue<vecteur, deque<vecteur>, compare > laQueue;

  vecteur v;  // Valeur courante
  char operation;

  cout << "Programme exemple getion d'une queue de priorite\n\n";
  /* Tant que l'utilisateur le veut ... */
  do
  {
    cout << "Choisissez:\n\ti: inserer\te: extraire\tq: quitter\n"
         << "Quelle operation: ";
    cin >> operation;
    /* En fonction du choix... */
    switch ( operation )
    {
      case 'i': case 'I':  // Inserer un nouvel element
        cout << "Coordonne X: "; cin >> v.x;
        cout << "Coordonne Y: "; cin >> v.y;
        laQueue.push ( v ); break;
      case 'e': case 'E':  // Extraire l'element en tete
        if ( !laQueue.empty() )
        { /* Queue non vide => operation possible! */
          v = laQueue.top ();
          cout << "Valeur extraite: X= " << v.x << ", Y= "
                                         << v.y << endl;
          laQueue.pop();
        }
        else
          cout << "La queue est vide!\n";
        break;
      case 'q': case 'Q': break;  // C'est fini
      default:
        cout << "***Erreur, recommencez!\n";
    }
  } while ( !( operation == 'q' || operation == 'Q' ) );

  return EXIT_SUCCESS;
}
```

Voici le résultat d'une exécution. Comparez avec ceux obtenus par la gestion d'une queue "simple":

```
Programme exemple getion d'une queuede priorite

Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: i
Coordonne X: 1
Coordonne Y: 3
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: i
Coordonne X: 2
Coordonne Y: 2
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: i
Coordonne X: 4
Coordonne Y: 4
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: e
Valeur extraite: X= 4, Y= 4
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: e
Valeur extraite: X= 1, Y= 3
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: e
Valeur extraite: X= 2, Y= 2
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: e
La queue est vide!
Choisissez:
        i: inserer      e: extraire     q: quitter
Quelle operation: q
```

Remarques complémentaires:

Nous avons vu que pour les types de base, seul le premier paramètre du constructeur est indispensable. La raison est simple, pour eux, comme pour tous les autres types d'ailleurs, le deuxième paramètre (le conteneur de base) convient généralement parfaitement. Il n'en va pas de même du troisième paramètre et comme nous ne pouvons pas utiliser la valeur par défaut du deuxième si nous devons donner le troisième, nous sommes obligés de fournir tous les paramètres.

Sous une forme simplifié, nous pouvons dire que ce troisimème paramètres correspond à une classe implémentant lopérateur ( ) sous forme d'une fonction booléen rélisant une "comparaison" entre 2 valeurs du type de l'instanciation des éléments de la queue de priorité.

Ceci peut se faire directement dans la classe définissant le type des éléments, comme nous l'avons utilisé dans l'exemple de programme ci-dessus:

```
struct vecteur
{  int x, y;
  bool operator () ( const vecteur & v1, const vecteur & v2 )
  {
    return v1.x*v1.x + v1.y*v1.y < v2.x*v2.x + v2.y*v2.y;
  }
}
```

ou par l'intermédiaire d'une classe spécifique implémentant cet opérateur, comme nous avons essayé de le mettre en évidence dans les parties mises en commentaires:

```
class compare
{ public:
    bool operator () ( const vecteur & v1, const vecteur & v2 )
    {  return v1.x*v1.x + v1.y*v1.y < v2.x*v2.x + v2.y*v2.y;
    }
};
```

C'est cette classe qu'il faut passer comme troisième paramètre au constructeur.

❑ *Les conmteneurs assocciatifs*

Raappelons qu'ils sont au nobre de 4, soit les:

map, multimap, set, multiset

Contrairement à ce qui se passait avec les conteneurs séquentiels, pour les associatifs ce n'est pas l'utilisateur qui décide de l'endroit où s'insère un nouvel élément dans la structure. Cet endroit dépendra de la valeur d'une clé explicite pour les *map* et *multimap*, implicite pour les *set* et *multiset*.

La différence entre *map* et *multimap* réside dans le fait que pour les *map* la clé doit être unique, c'est-à-dire que dans le conteneur il ne peut y avoir qu'un seul objet possédant cette clé. Par contre pour les *multimap* cette clé peut apparaître plusieurs fois dans le conteneur associée à des valeurs différentes ou égales pour les autres membres du type.

La distinction entre *set* et *multiset* revient au même. Toutefois il faut nuancer cette affirmation. Dans un *set*, la valeur globale de l'objet fait office de clé, et non pas un membre de l'objet comme pour les *map*. Donc dans un set une seule occurrence d'une valeur donnée peut y être introduite, alors qu'un multiset peut comprendre plusieurs fois la même valeur.

La classe pair

Avant d'aller plus loin dans la description des conteneurs associatifs, il nous faut introduire une notion complémentaire qui nous sera utile, celle de la classe *pair*.

Son objectif: regrouper en un seul objet des couples d'objets de n'importe quel type. La définition de cette classe se trouve dans <utility> qu'il faut en principe inclure dans votre code[1].

La classe est générique, il faut l'instancier avec les 2 types constituant le couple. Puisqu'il existe un constructeur par défaut nous pouvons donc déclarer par exemple une variable formée d'un caractère et d'un entier sous la forme:

```
pair<char, int> couple;
```

Un autre constructeur vous permet d'initialiser l'objet que vous déclarez avec un couple de valeurs des types appropriés:

```
pair<char, int> autreCouple ( 'a', 27 );
```

La classe met à disposition l'opérateur d'affectation:

```
couple = pair<char, int> ( 'a', 27 );
```

Toutefois, dans une telle situation il sera péférable d'utiliser la méthode *make_pair*:

```
couple = make_pair ( 'a', 27 );
```

Le premier membre d'un tel couple s'appelle toujours *first* et le deuxième *second*, ainsi:

```
cout << couple.first << "   " << couple.second << endl;
```

affichera: `a   27`

La méthode *make_pair* peut éventuellement s'utiliser indépendemment du fait d'avoir déclaré une variable du type approprié. Ainsi, même si ce n'est pas très utile ici:

```
cout << make_pair ("Toto", "TUTU").first;
```

affichera: `Toto`

Nous comprendrons l'utilité de cette classe en traitant des conteneurs *map* et *multimap*.

[1] De nonmbreux autres fichiers d'inclusion incluent eux-mêmes se fichier; vous en héritez donc parfois de manière indirecte.

Fonctionnalité communes

En fait, tous les conteneurs associatifs possèdent les mêmes fonctionnalités, la subtilité résidera dans la manière de les utiliser. Tout d'abord celles valables pour tous les conteneurs (séquentiels ou associatifs). Rappelons-en brièvement la liste:

- L'opétrateur: =
- Les itérateurs: *begin, end, rbegin, rend*
- Les méthodes: *size, empty, max_size, insert, erase, swap, clear*

Ensuite viennent les méthodes communes aux conteneurs associatifs, soit:

- *find, count, lower_bound, upper_bound, key_comp, value_comp, equal_range.*

set

Commençons par introduire la notion de *set*, un peu plus simple à comprendre. On peut l'assimiler à un ensemble d'éléments de n'importe quel type (celui qui servira à instancier le ceteneur!). Une valeur donnée ne peut apparaître qu'une seule fois dans un *set* alors qu'elle pourra se trouver à plusieurs exemplaires dans un *multiset*.

Pour utiliser un *set* il faut inclure dans votre programme:

```
#include <set>
```

L'instanciation se fait en donnant au minimum le type des éléments du *set*. Les 2 autres paramètres ayant une valeur par défaut, nous devrons donner une valeur pour le deuxième, comme nous l'avons vu pour les queues de priorité pour leur troisième paramètre.

Relevez tout de suite que les éléments d'un *set* se comportent comme des constantes, c'est-à-dire qu'une fois introduit dans le *set*, on ne peut plus modifier leur contenu. Pour obtenir un effet identique il faudra extraire l'élément du *set* puis réintroduire la nouvelle valeur. Toutefois ceci ne pose aucun problème au vu des opérations généralement réalisées sur ce type de conteneur.

Le constructeur par défaut n'a pas de paramètre, il permet de déclarer un *set* vide. Les 2 autres constructeurs permettent, pour l'un, d'initialiser avec tout le contenu d'un autre *set* de même type, et pour l'autre, d'initialiser avec 2 itérateurs (début et fin) sur un conteneur dont les éléments sont du même type.

Les méthodes:

- `iterator find ( const key_type& x ) const;`

Livre un itérateur sur l'élément dont la valeur *x* est transmise en paramètre, ou l'itérateur *end()* si *x* ne se trouve pas dans le *set*. Surtout utile dans ce contexte pour enchainer avec d'autres opérations.

- `size_type count ( cont key_type& x ) const;`

Livre 1 si la valeur *x* est présente dans le *set* et 0 sinon[1].

- `iterator lower_bound ( const key_type& x ) const;`

Livre un itérateur sur le premier élément du *set* qui n'est pas plus petit que la valeur de *x*[2].

- `iterator upper_bound ( const key_type& x ) const;`

Livre un itérateur sur le premier élément du *set* qui est plus grand que la valeur de *x*[3].

- `pair<iterator,iterator> equal_range ( const key_type& x ) const;`

Ne présente aucun intérêt dans le contexte d'un set, nous en raparlerons pour les *multiset*!

Pour finir la liste, *key_comp()* et *value_comp()* livrent tous 2 l'obets (fonction ou classe) permettant les comparaisons d'éléments du *set*.

A titre d'exemple d'utilisation d'un *set*, reprenons comme programme la détermination des nombres premiers. Notez toutefois que la solution proposée ci-dessous n'est certainement pas idéale, ni efficace!

[1] Rappel: dans un set une valeur donnée n'est présente qu'une seule fois au maximum!

[2] a) De manière interne les éléments sont ordonnées dans le set. b) Si aucun élément correspond, on obtient *end()*.

[3] Si aucun élément correspond, on obtient *end()*.

```
/*
  Set, nombres premiers
    Set.cpp
*/
#include <iostream>
#include <set>
#include <cstdlib>
using namespace std;

int main ()
{
  set<int> table;
  unsigned int limite;  // Jusqu'ou?
  unsigned int nbValeurs = 1 ; // Combien deja dans la table?
  const unsigned int NB_PAR_LIGNE = 10;


  cout << "Nombres premiers:\n\nDonnez la limite (>3): ";
  cin >> limite;
  cout << "Les nombres premiers sont:\n\n" << 2;
  /* Chercher tous les nombres premiers ... */
  for ( unsigned int courant = 3; courant <= limite; courant += 2 )
  {
    set<int>::iterator it;
    /* Courant a-t-il un diviseur? */
    for ( it = table.begin(); it != table.end(); it++ )
      if ( courant % *it == 0 ) break;
    /* Si on n'a pas touvé de divisuers */
    if ( it == table.end() )
    {
      table.insert ( courant );
      if ( ++nbValeurs % NB_PAR_LIGNE == 0 )
        cout << endl;
      else
        cout << ", ";
      cout << courant;
    }
  }
  cout << endl;

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

multiset

Tout d'abord rappelons qu'une valeur donnée peut apparaître plusieurs fois dans un *multiset*.

Pour utiliser un *multiset* il faut aussi inclure dans votre programme:

```
#include <set>
```

L'instanciation se fait de la méme manière que pour un *set* simple et les constructeurs présentent les mêmes caractéritiques. Les méthodes sont identiques puisque communes à tous les conteneurs associatifs. Nous nous contenterons donc ci-dessous de mettre en évidence (gras) les variantes de comportement.

- `iterator find ( const key_type& x ) const;`

Livre un itérateur sur **l'un des** éléments dont la valeur *x* est transmise en paramètre, ou l'itérateur *end()* si *x* ne se trouve pas dans le *set*.

- `size_type count ( cont key_type& x ) const;`

Livre **le nombre de fois que** la valeur *x* est présente dans le *set*.

- `iterator lower_bound ( const key_type& x ) const;`

Livre un itérateur sur le premier élément du *set* qui n'est pas plus petit que la valeur de *x*[1].

- `iterator upper_bound ( const key_type& x ) const;`

Livre un itérateur sur le premier élément du *set* qui est plus grand que la valeur de *x*[2].

- `pair<iterator,iterator> equal_range ( const key_type& x ) const;`

Livre, sous forme d'un couple (*pair*) 2 itérateurs permettant de délimiter l'ensemble des valeurs égales à *x*! Si le *multiset* ne contient qu'une seule fois cette valeur les 2 itérateurs désignent cet élément et si le *multiset* ne contient pas la valeur *x*, les 2 itérateurs prennent la valeur *end()*.

[1] a) De manière interne les éléments sont ordonnées dans le set. b) Si aucun élément correspond, on obtient *end()*.

[2] Si aucun élément correspond, on obtient *end()*.

Voici un exemple de programme utilisant un *multiset* et un *set*, permettant de compter le nombre de chiffres apparaissant dans une ligne de texte donnée au clavier:

```
/*
  Multiset, compter les chiffres
    MultiSet.cpp
*/
#include <iostream>
#include <set>
#include <string>
#include <cstdlib>
using namespace std;

int main ()
{
  char lesChiffres [] = "0123456789";
  set<char> chiffres ( lesChiffres, lesChiffres + 9 );
  multiset<char> caracteres;
  char caractereCourant;

  cout << "Compteur de chiffres\n\nDonnez une phrase:\n";

  /* Traiter toutes la phrase */
  do
  {
    caractereCourant = cin.get ();
    caracteres.insert ( caractereCourant );
  } while ( caractereCourant != '\n' );

  for ( set<char>::iterator it = chiffres.begin(); it !=
chiffres.end(); it++ )
    cout << *it << " est apparu " << caracteres.count ( *it ) << "
fois\n";

  cout << endl;

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

map

Les informations contenues dans un *map* (ou un *multimap*) se composent toujours de 2 parties (*pair*). La première sert de clé pour accéder à l'information contenue dans la deuxième. La complexité de l'opération d'accès à un élément donné par sa clé est de l'ordre de *0(log n^1)*.

Pour l'utiliser, il faut inclure le fichier *map* dans votre programme:

```
#include < map>
```

Comme pour tous les autres conteneurs, la classe *map* est générique. Cela signifie que lors de la déclaration d'un objet il faudra au minimum donner comme premier paramètre le type de la clé et comme deuxième le type de l'information associé:

```
map<int, string> monMap;
```

Comme dans les cas déjà étudiés, si nécessaire (si la clé est une classe) nous devons fournir un troisième paramètre sous la forme d'une classe implémentant l'opérateur ( ).

Dans l'exemple ci-dessus, nous utilisons le constructeur par défaut. Comme pour les autres conteneurs, nous pouvons aussi initialiser la variable lors de sa déclaration, soit avec un autre conteneur du même type:

```
map<int, string> tonMap ( monMap );
```

soit avec 2 itérateurs définissant une tranche d'un autre conteneur dont les éléments sont du type: "couple d'éléments" respectivement du type de la clé et du type de l'information. Si nous disposons par exemple d'un *vector* de la forme:

```
vector< pair<int, string> > vecteur…;
```

Nous pouvons alors déclarer:

```
map<int, string> tonMap ( vecteur.begin(), vecteur.end() );
```

Dans le cadre de la classe *map* les clés doivent être uniques, donc ne peuvent pas apparaitre à plusieurs exemplaires dans un conteneur, même si l'information associée est différente. Par analogie avec les *set/multiset*, nous pouvons dire qu'un *multimap* pourra comporter des clés multiples, c'est-à-dire apparaissant à plusieurs exemplaires et qui peuvent être associées à des informations identiques ou différentes.

[1] Où *n* représente toujours le nombre d'éléments dans le conteneur.

Contrairement aux *set/multiset* que nous venons de voir, la partie information[1] d'un *map/multimap* peut changer de valeur en cours de traitement.

Un tel conteneur offre, du point de vue du principe tout au moins, un accès direct à ses éléments par l'intermédiaire de sa clé. Cela signifie que la classe surcharge l'opérateur [ ]. L'indice à donner correspond à une valeur potentielle de la clé et la partie de l'objet ainsi désigné correspond à l'information. Avec ce qui précède, nous pouvons écrire:

```
monMap [ 12 ] = "Toto";
```

qui enregistre dans le conteneur un élément dont la clé vaut *2* et l'information *Toto*.

Cette manière d'écrire permet aussi d'accéder à l'information liée à une clé donnée:

```
cout << monMap [ 12 ] << endl;
```

affichera *Toto*.

Attention: accéder pour consultation à un élément qui n'existe pas crée cet élément, qui à partir de là se trouvera présent dans le conteneur!

Bien entendu nous pouvons continuer à accéder au éléments par l'intermédiaire d'itéraateurs:

```
cout << ( * monMap.begin() ).second << endl;
```

Le member *first* d'un element représente sa clé et le member *second* son information.

Reprenons brièvement brièvement les différantes méthodes, qui, rappelons le, sont communes à tous les conteneurs associatif.

- iterator find (const key_type& x);
  const_iterator find (const key_type& x) const;

Livre un itérateur sur l'élément dont la valeur *x* est transmise en paramètre, ou l'itérateur *end()* si *x* ne se trouve pas dans le *set*. Contrairement aux *set* nous disposons de 2 versions puisque ici nous pouvons modifier l'information de l'élément de clé *x*.

- `size_type count ( cont key_type& x ) const;`

Livre 1 si la valeur *x* est présente dans le *set* et 0 sinon[2].

- `iterator lower_bound ( const key_type& x ) const;`

[1] Nous verrons que théoriquement on peut aussi changer la valeur de la clé, bien que cela soit plus que vivement déconseillé.

[2] Rappel: dans un map une valeur donnée n'est présente qu'une seule fois au maximum!

iterator lower_bound (**const** key_type& x);

Livre un itérateur sur le premier élément du *set* qui n'est pas plus petit que la valeur de *x*. Comme pour find et contrairement aux set, nous disposons de 2 versions, la deuxième permettant de modifier l'objet désigné par l'itérateur.

- `iterator upper_bound ( const key_type& x ) const;`
  iterator upper_bound (**const** key_type& x);

Livre un itérateur sur le premier élément du *set* qui est plus grand que la valeur de *x*[1]. A nouveau on retrouve les 2 formes.

- `pair<const_iterator,const_iterator> equal_range ( const key_type& x ) const;`
  `pair< iterator, iterator> equal_range ( const key_type& x );`

Livre sous la forme d'un couple d'itérateurs l'intervalle des éléments du conteneur dont la clé est égale à *x*. Le premier itérateur du résultat (*pair::first*) correspond toujours à la valeur de: *lower_bound (x)* et le deuxième (*pair::second*) à la valeur de: *lower_bound (x)*. Les map possédant une clé unique, cela ne représente pas un grand intérêt, l'intervalle désignant toujours l'unique élément s'il existe. Si la clé *x* n'est pas présente dans le *map*, les 2 itérateurs livrés sont égaux et désigne le premier élément dont la clé est plus grande que *x*, ou *end ()* s'il n'y en a aucun.

A nouveau, *key_comp()* et *value_comp()* livrent tous 2 l'obets (fonction ou classe) permettant les comparaisons d'éléments du *set*.

Comme toujours, voici un exemple de programme compilable, utilisant les possibilités des conteneurs map:

```
/*
  Map, gestion simplifiee de personnes
    Map.cpp
*/
#include <iostream>
#include <map>
#include <string>
#include <cstdlib>
using namespace std;

  typedef unsigned int CLE;
  typedef string INFO;
```

[1] Si aucun élément correspond, on obtient *end()*.

```
/* Affichage de tous les membres de la societe! */
void affiche ( const map<CLE, INFO> societe)
{
  cout << "No\tNOM\n\n";
  for ( map<CLE, INFO>::const_iterator it = societe.begin ();
          it != societe.end (); it++ )
    cout << it->first << '\t' << it->second << endl;
  cout << endl;
}

int main ()
{
  map<CLE, INFO> societe;
  char operation;
  INFO information;
  CLE laCle;

  cout << "Programme exemple gestion d'un map\n\n";
  /* Tant que l'utilisateur le veut ... */
  do
  {
    cout << "Choisissez:\ni: inserer\tc: consulter\ta: afficher"
         << "\tq: quitter\nQuelle operation: ";
    cin >> operation;
    /* En fonction du choix... */
    switch ( operation )
    {
      case 'i': case 'I':  // Inserer un nouvel element
        cout << "Numero de membre: "; cin >> laCle;
        cout << "Nom: "; cin >> information;
        societe [ laCle ] = information; break;
      case 'c': case 'C':  // Consulter un element
        cout << "Numero de membre: "; cin >> laCle;
        cout << "Le membre numero: " << laCle << " est: "
        << societe [laCle] << endl;
        break;
      case 'a': case 'A':
        affiche ( societe ); break;
      case 'q': case 'Q': break;  // C'est fini
      default:
        cout << "***Erreur, recommencez!\n";
    }
  } while ( !( operation == 'q' || operation == 'Q' ) );

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Avec un exemple d'exécution:

```
Programme exemple gestion d'un map

Choisissez:
i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: i
Numero de membre: 12
Nom: Toto
Choisissez:
i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: a
No      NOM

12      Toto

Choisissez:
i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: c
Numero de membre: 2
Le membre numero: 2 est:
Choisissez:
i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: a
No      NOM

2
12      Toto

Choisissez:
i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: i
Numero de membre: 2
Nom: Titi

Choisissez:
i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: a
No      NOM

2       Titi
12      Toto
```

```
Choisissez:
i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: i
Numero de membre: 3
Nom: Toto
Choisissez:
i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: a
No      NOM

2       Titi
3       Toto
12      Toto

Choisissez:
i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: w
***Erreur, recommencez!
Choisissez:
i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: q
Appuyez sur une touche pour continuer...
```

multimap

Nous pouvons faire le parallèle avec les *set*, les *multimap* correspondent à des *map* pouvant posséder des clés multiples. Toutefois, attention, l'opérateur [ ] n'est plus disponible, ce qui est tout à fait raisonnable puisque la même clé peut désigner plusieurs éléments différents.

Pour l'utiliser, il faut aussi inclure le fichier *map* dans votre programme:

```
#include < map>
```

Comme toujours, lors de la déclaration d'un objet il faut au minimum donner comme premier paramètre le type de la clé et comme deuxième le type de l'information associé:

```
multimap <string, int> monMap;
```

avec les mêmes remarques complémentaires que pour les autres conteneurs associatifs.

La méthode *find* livrera quant à elle un itérateur sur un des éléments possédant cette clé.

Relevons encore comme différences:

- `size_type count ( cont key_type& x ) const;` présente un intérêt par rapport aux map, puisque plusieurs elements peuvent psséder la même clé.
- *erase*, sous sa forme: `size_type erase ( const key_type& x )` peut provoquer l'effacement de plusieurs éléments du conteneur, tous ceux qui possèdent cet clé.

Finalement, et toujours en raison du fait de la présence possible de clés multiples, les méthodes: *lower_bound*, *upper_bound* et *equal_range* reprennent tous leur sens!

Terminons ce chapitre par un exemple:

```
/*
  Multimap, gestion simplifiee de personnes
    Multimap.cpp
*/
#include <iostream>
#include <map>
#include <string>
#include <cstdlib>
using namespace std;

  typedef string NOM;
  typedef double TAILLE;

/* Affichage de tous les membres de la societe! */
void affiche ( const multimap<NOM, TAILLE> societe)
{
  cout << "Taille\tNom\n\n";
  for ( multimap<NOM, TAILLE>::const_iterator it = societe.begin ();
          it != societe.end (); it++ )
    cout << it->second << '\t' << it->first << endl;
  cout << endl;
}
int main ()
{
  multimap<NOM, TAILLE> societe;
  char operation;
  TAILLE laTaille;
  NOM leNom;

  cout << "Programme exemple gestion d'un multimap\n\n";
  /* Tant que l'utilisateur le veut ... */
  do
  {
    cout << "Choisissez:\n\ti: inserer\tc: consulter\ta: afficher"
         << "\tq: quitter\nQuelle operation: ";
    cin >> operation;
    /* En fonction du choix... */
    switch ( operation )
    {
      case 'i': case 'I':  // Inserer un nouvel element
        cout << "Nom: "; cin >> leNom;
        cout << "Taille: "; cin >> laTaille;
        societe.insert ( make_pair ( leNom, laTaille ) ); break;
      case 'c': case 'C':  // Consulter un element
        cout << "Nom: "; cin >> leNom;
        cout << leNom << " a une taille de "
             << societe.find(leNom)->second << " m.\n";
        break;
```

```
     case 'a': case 'A':
        affiche ( societe ); break;
      case 'q': case 'Q': break;  // C'est fini
      default:
        cout << "***Erreur, recommencez!\n";
     }
  } while ( !( operation == 'q' || operation == 'Q' ) );

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Et le résultat d'une exécution de ce programme:

```
Programme exemple gestion d'un multimap

Choisissez:
        i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: i
Nom: Toto
Taille: 1.66
Choisissez:
        i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: c
Nom: tutu
tutu a une taille de 4.49864e-312 m.
Choisissez:
        i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: i
Nom: Tutu
Taille: 187
Choisissez:
        i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: a
Taille  Nom

1.66    Toto
187     Tutu

Choisissez:
        i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: i Tutu
Nom: Taille: 1.87
```

```
Choisissez:
        i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: a
Taille  Nom

1.66    Toto
187     Tutu
1.87    Tutu

Choisissez:
        i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: c
Nom: Tutu
Tutu a une taille de 187 m.
Choisissez:
        i: inserer      c: consulter    a: afficher     q: quitter
Quelle operation: q
Appuyez sur une touche pour continuer...
```

Héritage 1 / héritage simple

Introduction

Dans ce chapitre nous allons introduire les notions de base de l'héritage, qui représente, après l'encapsulation, le deuxième principe de base de la programmation objet. Rappelons que l'encapsulation permet, pour des raisons de sécurité, de cacher à l'utilisateur l'implémentation des données.

L'héritage consiste à construire une classe dérivée à partir d'une classe existante, dont elle héritera des données et des méthodes[1]. La classe dérivée pourra aussi, par rapport à sa classe de base:

- Compléter ses données en ajoutant de nouveaux membres.
- Compléter ses fonctionnalités en ajoutant de nouvelles méthodes.
- Modifier le comportement des objets en surchargeant/redéfinissant les méthodes offertes par la classe de base.

Nous allons donner ci-dessous, brutalement, un premier exemple assez mauvais. Toutefois il nous permettra d'introduire nos premières règles et petit à petit des possibilités meilleures. Nous définirons d'abord une classe *Embarcation*; un tel objet est caractérisé par sa longueur et son nombre de places (le nombre de passagers qu'elle peut embarquer)[2]. De cette classe nous dériverons une nouvelle classe *Voilier*, qui héritera des caractéristiques d'une *Embarcation*, mais nous y ajouterons la surface de voile ainsi qu'une indication s'il s'agit d'un voilier lesté ou non.

Voici le code, avec toujours ce que nous considérons comme une démarche propre, le découpage en différentes unités compilables séparément:

[1] Nous nuancerons ceci dans la suite de ce chapitre.

[2] Dans la réalité, il faudra certainement gérer bien plus d'informations. Toutefois, afin de réduire le code, nous simplifierons, comme nous l'avons toujours fait, les exemples qui vont suivre, même si ainsi ils ne reflètent que peu la réalité!

```
#ifndef __EMBARCATION__
#define __EMBARCATION__
/*
  Gestion de base d'un objet embarcation
  HERITAGE1/Embarcation.h
*/
class Embarcation
{
   unsigned int nbPlaces;
   float        longueur;

public:
  /* Affiche les caracteristiques d'une embarcation */
  void afficher ();
  /* Permet de modifier le nombre de places */
  void modifierPlaces   ( unsigned int );
  /* Permet de modifier la longueur */
  void modifierLongueur ( float );
};
#endif
```

```
/*
  Gestion de base d'un objet embarcation
  HERITAGE1/Embarcation.cpp
*/
#include "Embarcation.h"
#include <iostream>
using namespace std;

/* Affiche les caracteristiques d'une embarcation */
void Embarcation::afficher ()
{
  cout << "Longueur: " << longueur << "m; nombre de places: "
       << nbPlaces << endl;;
}

/* Permet de modifier le nombre de places */
void Embarcation::modifierPlaces ( unsigned int nombre )
{
  nbPlaces = nombre;
}

/* Permet de modifier la longueur */
void Embarcation::modifierLongueur ( float grandeur )
{
  longueur = grandeur;
}
```

```
#ifndef __VOILIER__
#define __VOILIER__
#include "Embarcation.h"
/*
  Classe Voilier, herite de Embarcation
  HERITAGE1/Voilier.h
*/
class Voilier : public Embarcation
{
   /* Adjonction aux caracteristiques d'une embarcation */
   float surfaceVoile;
   bool  leste;

public:
  /* Fixe la surface de voile et indique s'il est leste! */
  void fixerCaracteristiques ( float, bool );
};
#endif


/*
  Classe Voilier, herite de Embarcation
  HERITAGE1/Voilier.cpp
*/
#include "Voilier.h"
#include <iostream>
using namespace std;

/* Fixe la surface de voile et indique s'il est leste! */
void Voilier::fixerCaracteristiques ( float surface, bool estLeste )
{
  surfaceVoile = surface;
  leste        = estLeste;
}



/*
  Programme exemple: Test des notions d'heritage
  HERITAGE1/TestBateaux
*/
#include "Embarcation.h"
#include "Voilier.h"
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  Embarcation uneEmbarcation;
  Voilier     unVoilier;
```

```
  uneEmbarcation.modifierLongueur ( 5.3 );
  uneEmbarcation.modifierPlaces ( 5 );
  unVoilier.modifierLongueur ( 3 );
  unVoilier.modifierPlaces ( 3 );
  unVoilier.fixerCaracteristiques( 24.0, true );
  cout << "Programme \"Bateaux\"\n\n";
  cout << "Caracteristiques de l'embarcation:\n";
  uneEmbarcation.afficher ();
  cout << " Caracteristiques du voilier:\n";
  unVoilier.afficher ();

  uneEmbarcation = unVoilier;
  cout << " Caracteristiques de l'embarcation apres affectation:\n";
  uneEmbarcation.afficher ();  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Résultat de l'exécution:

```
Programme "Bateaux"


Caracteristiques de l'embarcation:
Longueur: 5.3m; nombre de places: 5
Caracteristiques du voilier:
Longueur: 3m; nombre de places: 3
Caracteristiques de l'embarcation apres affectation:
Longueur: 3m; nombre de places: 3

Fin du programme...Appuyez sur une touche pour continuer...
```

Encore une fois, avant d'aller plus loin, rappelons qu'il s'agit d'un exemple de mauvaise qualité!

Pour la classe *Embarcation* qui servira de base pour construire des classes dérivées, nous nous trouvons dans une situation classique que nous connaissons bien maintenant. Relevons simplement que pour l'instant nous n'y définissons, ni constructeur ni destructeurs, ce qui entre autre nous a obligé d'introduire les méthodes: *modifierPlaces* et *modifierLongueur*. La méthode *afficher* va permettre, comme son nom le laisse supposer, d'afficher les caractéristiques d'une embarcation.

Classe dérivée

Un *Voilier* est une *Embarcation*, par contre l'inverse n'est pas (toujours) vrai! La classe *Voilier* va hériter des propriétés de la classe *Embarcation*. Elle sera dérivée d'elle. Ceci se traduit par la forme que nous donnons à l'en-tête de la déclaration de cette classe, à savoir, dans notre code:

```
class Voilier : public Embarcation { …
```

Le ***public***[1] *Embacation* précise que la classe *Voilier* hérite d'*Embarcation*, mais en plus il précise que non seulement les éléments **public** d'*Embarcation* peuvent s'utiliser dans *Voilier*, mais que les utilisateurs de la classe *Voilier* auront aussi accès à ces éléments. Rien ne nous empêchera par la suite de dériver une classe à partir de *Voilier*. Elle héritera alors aussi des propriétés d'*Embarcation*. Le processus peut se poursuivre à autant de niveaux que désirés.

Si la classe dérivée a directement accès aux éléments **public** de la classe dont elle hérite, ce n'est pas le cas des éléments privés. Heureusement, car sinon il suffirait à l'utilisateur de créer une classe dérivée pour violer le principe d'encapsulation. Donc dans notre exemple *Voilier* n'a pas un accès direct à *nbPlaces* et *longueur*. Par contre il peut modifier (affecter) ces grandeurs par l'intermédiaire des méthodes *modifierPlaces* et *modifierLonguer*, qui elles sont **public** dans *Embarcation*.

Utilisation de la classe héritée

Dans la classe *Voilier* nous avons ajouté, à une Embarcation "de base", les membres données représentant la surface de voile du bateau et l'indication s'il est lesté ou non. Au niveau méthodes, nous mettons simplement à disposition *fixerCaracteristiques*, permettant de compléter les informations propre à un *Voilier*. Pour les informations de base (*longueur* et *nombrePlaces*), nous utilisons pour l'instant, bien que cela ne soit pas très pratique, les méthodes héritées *d'Embarcation*. Ceci nous démontre cette possibilité, nous y reviendrons d'ici peu en parlant de l'affichage. Puisque ce n'est pas une bonne solution, modifions tout de suite cette méthode *fixerCaracteristiques*: en lui ajoutant 2 paramètres pour la longueur et le nombre de places, de telle sorte que nous puissions fixer toutes les caractéristiques d'un *Voilier* et que par la suite cette opération puisse faire l'objet d'un constructeur. Dans le corps de la méthode nous appelons

[1] Comme nous le verrons par la suite, il existe d'autres possibilités: **private** et **protected**!

directement *modifierPlaces* et *modifierLonguer*. Ainsi, l'utilisateur final, s'il ne manipule que des voiliers, n'a pas à se préoccuper de ce qui se trouve dans *Embarcation*! Une nouvelle version de cette méthode pourrait se présenter comme suit:

```
/* Fixe les caracteristiques d'un voilier */
void Voilier::fixerCaracteristiques (
                              float longueur, unsigned int nbPlaces,
                              float surface, bool estLeste )
{
  Embarcation::modifierLongueur ( longueur );
  modifierPlaces   ( nbPlaces );
  surfaceVoile = surface;
  leste        = estLeste;
}
```

Note: pour l'exemple, nous avons laissé 2 variantes des appels, chose qui évidemment en pratique ne devrait pas se faire! Dans la première:

```
Embarcation::modifierLongueur ( longueur );
```

nous explicitons que *modifierLongueur* vient de *Embarcation*, ce qui n'est pas indispensable puisqu'il n'y a pas ambiguïté! D'où la deuxième forme d'appel utilisé:

```
modifierPlaces   ( nbPlaces );
```

Notez aussi que ces méthodes s'appliquent automatiquement à l'objet courant, celui qui a appelé *fixerCaracteristiques*. Pourtant cet objet est de type *Voilier* et non *Embarcation*, mais *Voilier* dérive d'*Embarcation*, il y a conversion implicite.

❑ *Redéfinition des méthodes héritées*

Et qu'advient-il de l'affichage? Si nous ne définissons pas de méthode explicite d'affichage pour un *Voilier*, un tel objet hérite de l'affichage d'une *Embarcation*. Toutefois cette méthode ne prend en considération que la longueur et le nombre de places, une *Embarcation* ne possédant que ces caractéristiques. Si nous avions accès aux membres *longueur* et *nbPlaces* depuis *Voilier* (c'est heureusement pas le cas!) nous pourrions imaginer mettre à disposition une méthode spécifique *afficherVoilier*. De toute façon ce ne serait pas une bonne solution. Mais alors? La

solution est à la fois plus simple et plus propre. Nous allons redéfinir la méthode *afficher* dans *Voilier*. Elle appellera *afficher* d'*Embarcation* pour réaliser le travail de base commun, qu'elle complétera par l'affichage de ses propres informations.

Le code de cette méthode devient:

```
void Voilier::afficher ()
{
  Embarcation::afficher();
  cout << "Voilure: " << surfaceVoile;
  if ( leste )
    cout << ", leste\n";
  else
    cout << ", non leste\n";
}
```

Notez l'indication de l'appel d'*afficher* d'*Embarcation*:

```
Embarcation::afficher();
```

Préciser qu'il s'agit de la méthode *afficher* d'*Embarcation* est important pour éviter un appel récursif infini! Pour réaliser cet appel, il y aura à nouveau conversion implicite de *Voilier* vers *Embarcation*.

De manière plus générale, nous pouvons toujours redéfinir une méthode d'une classe de base dans une classe dérivée et, si nécessaire, d'appeler explicitement la méthode de même nom de la classe de base en utilisant l'opérateur de résolution de portée.

❑ *Cas des membres "données"*

Pour les membres données, le comportement diffère! Si dans la classe dérivée vous définissez un membre portant le même nom qu'un membre de sa classe parent, vous ajoutez ce membre à la nouvelle classe. Les 2 cohabitent, qu'ils soient du même type ou d'un type différent. Imaginons les déclarations de classes, totalement artificielles, suivantes:

```
class C1
{
public:  //[1]
  int a;
  C1 ( int p = 1 ) { a = p; }
};
class C2 : public C1
{
public:
  float a;
  C2 ( float p = 2.5 ) { a = p; }
};
```

Si nous déclarons:

```
C1 o1;  C2 o2;
```

o1.a désigne la valeur entière 1 alors que *o2.a* désigne la valeur réelle 2.5. En d'autres termes, par défaut, on désigne le membre qui porte ce nom dans la classe effective de l'objet. Toutefois, pour un objet de la classe dérivé, on accède au membre de même nom de la classe parent en utilisant l'opérateur de résolution de portée.

Ainsi:

o2.C1::a

désigne la valeur 1 soit le membre de o2 hérité de sa classe parent.

[1] Pour simplifier les écritures nous déclarons volontairement tout **public**, ce qui en pratique ne se justifie pas!

❑ *Affectation et passage en paramètre*

Nous avons déjà signalé qu'un objet d'une classe dérivée est aussi un objet de la classe parent, mais que l'inverse n'est pas vrai. Nous avons aussi vu qu'une méthode **public** de la classe parent peut sans problème[1] s'appliquer à un objet d'une classe dérivée. Quelque soit le niveau d'héritage, il y a conversion implicite. Il en va de même du passage en paramètre; d'ailleurs, appliquer une méthode à un objet correspond à un passage de l'objet en paramètre implicite.

Le programme principal de notre premier exemple nous montre en plus que nous pouvons considérer l'affectation de la même manière:

```
uneEmbarcation = unVoilier;
```

De manière plus générale, nous pouvons affecter à un objet (une variable) d'une classe parent un objet d'une classe dérivée[2]. Bien entendu, la variable affectée ne gardera de l'objet source que la partie commune des membres *données*, à savoir la partie héritée, le reste étant perdu!

❑ *Les constructeurs*

Dans notre exemple de programme, nous avons écrit du code avec, comme objectif, de présenter certaines possibilités et surtout de permettre l'introduction de variantes plus intéressantes. Dans cette optique, il semblerait maintenant raisonnable de revenir à des notions anciennes et d'utiliser des constructeurs.

Pour la classe *Embarcation* de notre exemple, rien de spécial à signaler, tout se passe comme nous l'avons présenté lors de l'introduction de la notion de constructeur. Ceci reste vrai pour toutes les classes "racines" que nous pourrions créer.

Par contre, pour les classes dérivées, la situation se complique quelque peu. En effet, le constructeur doit en principe pouvoir accéder aux membres *données* dont il a hérités de son parent. Toutefois ceci n'est pas possible de manière usuelle, ces éléments étant normalement privés! Si tel n'était pas le cas le constructeur s'écrirait normalement. En fait nous allons nous

[1] Pour autant que cela corresponde à la logique de ce que l'on veut faire!

[2] A nouveau quelque soit le niveau d'héritage.

retrouver dans la même situation que celle présentée pour les classes possédant des objets membres étant eux-mêmes des classes. Rappelons qu'il faut alors compléter la ligne d'en-tête du constructeur en question par ":" suivi du nom du constructeur parent, avec entre parenthèses les paramètres à lui transmettre.

Dans notre exemple cela peut donner un code du genre:

```
Voilier ( unsigned int nbPlaces = 0,
          float longueur = 0.0,
          float surfaceVoile = 0.0,
          bool  leste = false ) : Embarcation ( nbPlaces, longueur )
{
  this->surfaceVoile = surfaceVoile;
  this->leste        = leste;
}
```

Nous avons donné ici des valeurs par défaut aux paramètres, ce qui nous paraît raisonnable, entre autre pour pouvoir disposer de l'équivalent d'un constructeur sans paramètre. Toutefois, d'un point de vue purement syntaxique, il ne s'agit pas une obligation.

Le(s) constructeur(s) parent(s) est(sont) appelé(s) dans le bon ordre hiérarchique de dépendance(s), en partant de la racine et remontant jusqu'au constructeur que nous écrivons. Il en sera d'ailleurs de même lorsque les destructeurs seront appelés!

Les paramètres retransmis au constructeur de la classe parent peuvent être des expressions. Ainsi on peut imaginer pour le constructeur de la classe *Voilier*, même si cela n'a pas beaucoup de sens du point de vue de la logique:

```
Voilier ( unsigned int nbPlaces = 0,
          float longueur = 0.0,
          float surfaceVoile = 0.0,
          bool  leste = false )
              : Embarcation ( nbPlaces + 1, longueur * 2 )
{ …
```

❑ *Hiérarchie de classes*

Nous avons déjà signalé que nous pouvons construire des hiérarchies de classes. Cela signifie qu'une classe dérivée peut elle-même servir de parent à une nouvelle classe dérivée. Bien que nous aurons à nuancer cette affirmation avec la mise en œuvre d'autres formes d'héritage que **public**[1] ainsi qu'avec la notion d'héritage multiple, nous pouvons pour l'instant dire que tout se passe normalement, comme nous l'avons décrit jusqu'à présent.

Relevons simplement:

- La classe disposera indirectement de l'accès aux membres **public** du parent de son parent.
- Si sa classe parent redéfinit un membre public nous gardons l'accès à celui de son ancêtre par l'intermédiaire de l'opérateur de résolution de portée.
- Lors de l'appel d'un constructeur, les opérations se réaliseront dans le bon ordre, à savoir: d'abord le parent du parent, puis celui de son parent et finalement son propre constructeur.
- Le constructeur de la classe n'a pas à s'occuper de transmettre des paramètres au constructeur du parent de son parent puisque le constructeur de son parent s'en charge!

[1] Voir propagation des protections dans la suite de ce chapitre.

Limites "logique" de l'héritage

Rappelons qu'à travers une hiérarchie de classes, une classe hérite et donc indirectement met à disposition les membres **public** de ses parents. Si ce comportement semble logique au niveau de la structure du programme, cela ne l'est pas toujours au sens de la logique "humaine".

A titre d'exemple:

- Si un classe ancêtre met à disposition une méthode livrant un résultat booléen, permettant de comparer 2 objets de la classe, afin de déterminer s'ils sont égaux.
- Si les classes dérivées ne redéfinissent pas cette méthode pour leur propre usage.

Dans ces conditions, appliquer la méthode à un objet d'une classe dérivée implique l'appel de la méthode parent. Celle-ci ne va comparer que les informations propres au parent et livrera vrai si elles concordent, même si les valeurs des membres spécifiques à la classe dérivée diffèrent. C'est ce que nous démontrerons dans le programme exemple qui suit avec la méthode *egal* définie uniquement dans la classe *Embarcation*.

Ce problème revient de manière courante avec la définition d'opérateurs!

Le point par l'exemple

Nous avons maintenant introduit un certain nombre de notions de base sur l'héritage, qui doivent nous permettre d'écrire nos premiers programmes réellement orienté "programmation objet". Avant de compléter ces éléments, remettons à jour notre exemple de programme en utilisant un maximum des possibilités présentées. Dans cet exemple, pour en réduire la taille, nous ne gardons du point de vue des fonctionnalités que le strict minimum nécessaire à illustrer les points désirés. Toutefois, afin de bien comprendre les mécanismes mis en jeu, nous avons ajouté plusieurs instructions d'affichage. Nous vous recommandons donc, non seulement de bien étudier le code du programme, mais aussi de regarder en parallèle les résultats affichés:

```
#ifndef __EMBARCATION__
#define __EMBARCATION__
#include <iostream>
using namespace std;
/*
  Gestion de base d'un objet embarcation
  HERITAGE2/Embarcation.h
*/
class Embarcation
{
   unsigned int nbPlaces;
   float        longueur;
public:
  Embarcation ( unsigned int nbPlaces = 0, float longueur = 0.0 )
  {
    this->nbPlaces = nbPlaces;
    this->longueur = longueur;
    cout << "* Construction d'une embarcation:\n";
    this->afficher();
  }
  ~Embarcation (  )
  {
    cout << "* Destruction d'une embarcation:\n";
    this->afficher();
  }

  /* Affiche les caracteristique d'une embarcation */
  void afficher ();

  /* Comparer 2 embarcations */
  bool egal ( Embarcation & );
};
#endif



/*
  Gestion de base d'un objet embarcation
  HERITAGE2/Embarcation.cpp
*/
#include "Embarcation.h"
#include <iostream>
using namespace std;

/* Affiche les caracteristiques d'une embarcation */
void Embarcation::afficher ()
{
  cout << "Longueur: " << longueur << "m; nombre de places: "
       << nbPlaces << endl;;
}
```

```
/* Comparer 2 embarcations */
bool Embarcation::egal ( Embarcation & bateau )
{
  return bateau.nbPlaces == nbPlaces && bateau.longueur == longueur;
  cout << bateau.nbPlaces << endl;
  cout << nbPlaces << endl;
  cout << bateau.longueur << endl;
  cout << longueur << endl;

}



#ifndef __VOILIER__
#define __VOILIER__
#include "Embarcation.h"
#include <iostream>
using namespace std;
/*
  Classe Voilier, herite de Embarcation
  HERITAGE2/Voilier.h
*/
class Voilier : public Embarcation
{
   /* Adjonction aux caracteristiques d'une embarcation */
   float surfaceVoile;
   bool  leste;
public:
  Voilier ( unsigned int nbPlaces     = 0,
            float        longueur     = 0.0,
            float        surfaceVoile = 0.0,
            bool         leste        = false )
                    : Embarcation ( nbPlaces, longueur )
  {
    this->surfaceVoile = surfaceVoile;
    this->leste        = leste;
    cout << "*** Construction d'un voilier:\n";
    this->afficher();
  }

  ~Voilier (  )
  {
    cout << "* Destruction d'un voilier:\n";
    this->afficher();
  }

  /* Affiche les caracteristiques d'un voilier */
  void afficher ();
};
#endif
```

```
/*
  Classe Voilier, herite de Embarcation
  HERITAGE2/Voilier.cpp
*/
#include "Voilier.h"
#include <iostream>
using namespace std;

/* Affiche les caracteristiques d'un voilier */
void Voilier::afficher ()
{
  Embarcation::afficher();
  cout << "Voilure: " << surfaceVoile;
  if ( leste )
    cout << ", leste\n";
  else
    cout << ", non leste\n";
}
```

```
#ifndef __MULTICOQUES__
#define __MULTICOQUES__
#include "Voilier.h"
#include <iostream>
using namespace std;

/*
  Classe Multicoques, herite de Voilier
  HERITAGE2/Multicoques.h
*/
class Multicoques : public Voilier
{
   /* Adjonction des caracteristiques d'un Multicoques */
   unsigned int nbCoques;

public:
  Multicoques ( unsigned int nbPlaces     = 0,
                float        longueur     = 0.0,
                float        surface      = 0.0,
                bool         leste        = false,
                unsigned int nbCoques     = 1 )
                    : Voilier ( nbPlaces, longueur, surface, leste )
  {
    this->nbCoques = nbCoques;
    cout << "*** Construction d'un multicoques:\n";
    this->afficher();
  }
```

```
  ~Multicoques (  )
  {
    cout << "* Destruction d'un multicoques:\n";
    this->afficher();
  }

  /* Affiche les caracteristiques d'un Multicoques */
  void afficher ();
};
#endif



/*
  Classe Multicoques, herite de Voilier
  HERITAGE2/Multicoques.cpp
*/
#include "Multicoques.h"
#include <iostream>
using namespace std;

/* Affiche les caracteristiques d'un Multicoques */
void Multicoques::afficher ()
{
  Voilier::afficher();
  cout << "Nombre de coque(s) " << nbCoques << endl;
}




/*
  Programme exemple: Test des notions d'heritage
 HERITAGE2/TestBateaux
*/
#include "Embarcation.h"
#include "Voilier.h"
#include "Multicoques.h"
#include <cstdlib>
#include <iostream>
using namespace std;
int main ( )
{
  Embarcation uneEmbarcation ( 5, 5.3 );
  Voilier     unVoilier ( 3, 4.5, 15.0, true );
  Multicoques unMulticoques( 3, 4.5, 15.0, true, 2 );
  cout << "\n\nProgramme \"Bateaux\"\n\n";
  cout << "Caracteristiques de l'embarcation:\n";
  uneEmbarcation.afficher ();
  cout << "Caracteristiques du voilier:\n";
  unVoilier.afficher ();
```

```
  cout << "Caracteristiques du multicoques:\n";
  unMulticoques.afficher ();

  /* La comparaison nous montre les limites de l'heritage */
  if ( uneEmbarcation.egal ( unVoilier ) )
   cout << "\n-Le voilier est identique a l'embarcation\n";
  else
    cout << "\n-Le voilier n'est pas identique a l'embarcation\n";
  if ( unVoilier.egal ( unMulticoques ) )
    cout << "\n-Le multicoques est identique au voilier\n";
  else
    cout << "\n-Le multicoques n'est pas identique au voilier\n";

  uneEmbarcation = unVoilier;
  cout << "Caracteristiques de l'embarcation apres affectation:\n";
  uneEmbarcation.afficher ();

  cout << "\nFin du programme...";
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Cette version nous fournit les résultats suivants:

```
* Construction d'une embarcation:
Longueur: 5.3m; nombre de places: 5
* Construction d'une embarcation:
Longueur: 4.5m; nombre de places: 3
*** Construction d'un voilier:
Longueur: 4.5m; nombre de places: 3
Voilure: 15, leste
* Construction d'une embarcation:
Longueur: 4.5m; nombre de places: 3
*** Construction d'un voilier:
Longueur: 4.5m; nombre de places: 3
Voilure: 15, leste
*** Construction d'un multicoques:
Longueur: 4.5m; nombre de places: 3
Voilure: 15, leste
Nombre de coque(s) 2
```

```
Programme "Bateaux"

Caracteristiques de l'embarcation:
Longueur: 5.3m; nombre de places: 5
Caracteristiques du voilier:
Longueur: 4.5m; nombre de places: 3
Voilure: 15, leste
Caracteristiques du multicoques:
Longueur: 4.5m; nombre de places: 3
Voilure: 15, leste
Nombre de coque(s) 2

-Le voilier n'est pas identique a l'embarcation

-Le multicoques est identique au voilier
Caracteristiques de l'embarcation apres affectation:
Longueur: 4.5m; nombre de places: 3

Fin du programme...Appuyez sur une touche pour continuer...
* Destruction d'un multicoques:
Longueur: 4.5m; nombre de places: 3
Voilure: 15, leste
Nombre de coque(s) 2
* Destruction d'un voilier:
Longueur: 4.5m; nombre de places: 3
Voilure: 15, leste
* Destruction d'une embarcation:
Longueur: 4.5m; nombre de places: 3
* Destruction d'un voilier:
Longueur: 4.5m; nombre de places: 3
Voilure: 15, leste
* Destruction d'une embarcation:
Longueur: 4.5m; nombre de places: 3
* Destruction d'une embarcation:
Longueur: 4.5m; nombre de places: 3
```

Quelques points complémentaires

- Pour que l'héritage puisse fonctionner, la classe parent doit disposer d'un constructeur par défaut sans paramètre, ou, tout au moins, d'un constructeur dont tous les paramètres possèdent une valeur par défaut.
- Une classe dérivée peut n'ajouter aucun nouveau membre donnée à sa classe parent. Le but d'une telle opération consiste à offrir une différence de comportement sur les objets créés!

Propagation des protections

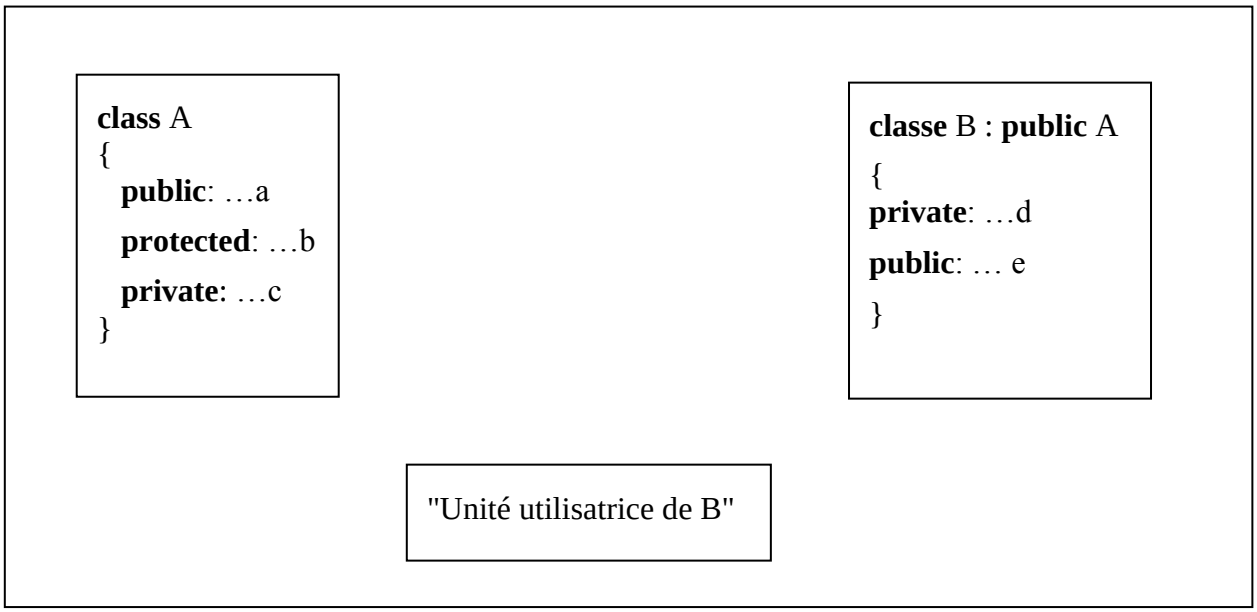
Nous savons, pour l'instant, que les éléments d'une classe peuvent jouir de 2 modes de visibilité, donc de protection, à savoir **private** et **public**. Les éléments **public** sont accessibles par n'importe qui du monde extérieur à la classe, alors que les éléments **private** ne le sont que depuis l'intérieur de la classe[1].

En réalité il existe un troisième niveau de protection, le mode protégé, introduit par le mot réservé **protected**, qui s'utilise comme **public** et **private** dans la définition d'une classe. Du point de vue du mode extérieur les membres **protected** se comportent exactement comme les membres **private**. Par contre, dans une classe dérivée ces éléments correspondent à des membres **public**. Toutefois cette propriété de visibilité ne sera pas exportée aux utilisateurs de la classe dérivée[2].

Exemple:

[1] Il faut considérer les fonctions amies comme internes à la classe; elles ont donc accès à ses éléments **private**.

[2] Relevons que cette possibilité ouvre une brèche dans le principe d'encapsulation!



Avec la structure ci-dessus:

- dans la classe *A* on peut utiliser *a*, *b*, *c*.
- dans la classe *B* on peut utiliser *a*, *b*, *d*, *e*
- dans l'unité utilisatrice de *B* on peut utiliser *a*, *e*

De plus, à de la création d'une classe dérivée, nous avons toujours spécifié qu'elle héritait de manière **public** de sa classe parent:

```
classe B : public A { …
```

Il s'agit certainement de la situation le plus courante, qui revient à dire que les membres de la classe parent conservent leurs propriétés de visibilité.

Plus rare, mais également possible, l'héritage peut se faire de manière privée:

```
classe B : private A { …
```

Dans ce cas, au sein de la classe *B*, les éléments de la classe *A* restent accessibles comme nous l'avons défini pour l'héritage **public**. Par contre, les unités utilisatrices de la classe *B* n'auront plus accès aux membres **public** de *A*[1].

Toutefois, la classe dérivée peut "remettre à disposition" une ou plusieurs méthodes de son parent en déclarant son prototype dans la zone **public** de la classe dérivé.

Relevez aussi que, dans de telles conditions, vous perderez la propriété qui permet d'affecter un objet de la classe dérivée à un objet de sa classe parent.

De plus, maintenant cet héritage peut se réaliser selon le mode protégé:

```
classe B : protected A { …
```

Dans ce cas, pour une unité utilisatrice de *B*, tous les membres hérités de *A* réagissent comme étant **priate**. Pour elle il n'y a donc pas de différence par rapport à un héritage **private**. Par contre le comportement diffère lorsque nous dérivons une nouvelle classe à partir de *B*. Les membres **private** le resteront, par contre ceux **public** et **protected** se comporteront tous 2 comme des éléments **protected** dans la nouvelle classe.

[1] A moins qu'elles ne soient aussi utilisatrices de manière directe de la classe *A*, auquel cas elle peut les utiliser, mais uniquement pour les objets de classe *A*!

Table des matières