Le langage de programmation C++

Patrick Smacchia 2002 patrick@smacchia.com

1 Introduction

- ✓ Ce cour ne constitue qu'une introduction au langage C++, et ne peut être vu comme une référence.
- ✓ LA référence du C++ est **[Stroustrup]**, écrit par directement par le père de ce langage.
- ✓ Cependant la plupart des possibilités offertes par ce langage seront abordées, avec un niveau de détail inégal, selon la fréquence d'utilisation de la possibilité.

- ✓ Le cour se divise comme suit :
  - o Un bref rappel du langage C, pour que le lecteur s'assure que sa connaissance de ce langage soit assez solide pour aborder le C++.
  - o Des détails sur la compilation C/C++.
  - o Les mécanismes simples de C++, qui n'en sont pas moins importants. Ils constituent les réponses à certains détails du C rédhibitoires.
  - o La programmation orientée objet avec C++ est l'ajout majeur au C++.
  - o Les mécanismes avancés de C++, qui sont des techniques évoluées pour faciliter la tâche du programmeur.
  - o Une introduction à la librairie standard générique (Standard Template Library STL).

2 Historique

- ✓ L'histoire du langage **C++**, est très liée à l'histoire du langage **C**.
- ✓ Le langage **C** lui-même est historiquement lié au systèmes d'exploitations type **UNIX**.
- ✓ 1969 : la première version du système UNIX voit le jour dans les laboratoires Bell. Le principal problème est qu'il est codé en assembleur.
- ✓ Peu après l'apparition d'UNIX, **Ken Thompson** créa un nouveau langage de programmation, nommé **B**, qui s'avéra être un peu trop simple et trop dépendant de l'architecture.
- ✓ 1971 : On commence à travailler sur le successeur de **B**, nommé dans l'ordre des choses **C**. On s'accorde pour dire que **Dennis Ritchie** est le véritable créateur du **C**.
- ✓ La puissance du **C** était véritable : ce langage n'avait pas été fait spécifiquement pour une sorte de machine ou pour une sorte de système. C'est donc un langage hautement portable. De plus, son rayon d'action s'étend du bas niveau de la machine (le **C** peut générer du code aussi rapide que du code assembleur), au haut niveau des langages orientés problèmes.
- ✓ Le **C** devint vite tellement populaire que tout le monde en a fait sa version. Le comité **ANSI** (*American National Standards Institute)* l'a donc normalisé en 1983.
- ✓ Le **C** avait donc de nombreux avantages. Mais il lui manquait ce qui caractérise la plupart des compilateurs modernes : la programmation orientée objets.
- ✓ 1980 : Le **Dr. Bjarne Stroustroup** (voir [Stroustroup]) de Bell Laboratories crée le **C++**.
- ✓ Les contraintes étés :
  - o un langage permettant la programmation orientée objets
  - o tout en restant hautement performant
  - o en en touchant un large public (donc en étant compatible avec le **C ANSI**).
- ✓ Il créa alors le **C++** sur la base du **C**, en prenant soin de préserver la compatibilité : tout programme **C** peut être compilé en **C++**.
- ✓ Il n'y a pas a l'heure actuelle de standard **ANSI C++** en tant que tel, mais on a ajouté a l'**ANSI C** des spécifications concernant le **C++**.
- ✓ Aujourd'hui, les langages **C** et **C++** sont deux des langages les plus utilisés de la planète, et tirent leur force de leur flexibilité, leur performance et surtout, de leur immense popularité.

3 Rappels sur le langage C

3.1 Organisation des fichiers sources

- ✓ Il y a trois types de fichier source, avec leurs extensions :
  - o Les fichiers sources qui ne contiennent que du code C. Leur extension est **`.c`**
  - o Les fichiers sources qui contiennent du code C++ et C. Leur extension peut être **`.cc`** ou **`.C`** (sous unix) ou **`.cpp`** (sous windows).
  - o Les fichiers d'en-tête ou header. Leur extension est **`.h`** ou **`.hpp`**
- ✓ Seul les fichiers sources C ou C++ sont compilés.
- ✓ Les fichiers d'en-tête servent à partager des déclarations (de variables de fonctions, de classes…) entre les fichiers C/C++.
- ✓ Ils sont inclus par les fichiers C/C++ au moyen de la directive préprocesseur `#include`.
- ✓ Comme les fichiers d'en-tête peuvent aussi inclure d'autres fichiers d'en-tête il y a risque de collision des déclarations. Pour éviter ceci très souvent les fichiers d'en-tête utilisent la compilation conditionnelle du préprocesseur, afin d'être sûr de ne pas inclure plus d'une fois un fichier d'en-tête dans un fichier C/C++. Par exemple :

```
Declaration.h (fichier d'en-tête)
#ifdef METTRE_UNE_MACRO_BIEN_LONGUE_ET_COMPLIQUEE_DECLARATION_H
#define METTRE_UNE_MACRO_BIEN_LONGUE_ET_COMPLIQUEE_DECLARATION_H
...
ici les déclarations
...
#endif METTRE_UNE_MACRO_BIEN_LONGUE_ET_COMPLIQUEE_DECLARATION_H
```

- ✓ Naturellement le nom de la macro utilisée doit être différente pour tous les fichiers d'en-tête.

3.2 Le préprocesseur

- ✓ Toute compilation de source est précédée d'une phase de mise en forme. Celle-ci est effectuée par un préprocesseur. Il n'effectue que des traitements simples, mais en aucun cas n'est chargé de la compilation. Toutes les directives adressées au préprocesseur sont précédées par le signe `#`.

- ✓ **Constantes symboliques :**
  Syntaxe: `#define nom_constante valeur_constante`
  Syntaxe: `#define nom_constante`
  A chaque fois que le nom de la constante sera rencontré, sa valeur lui sera substituée dans le source. Si l'on affecte pas de valeur, la valeur 1 est automatiquement affectée.

- ✓ **Constantes prédéfinies :**
  Il existe quelques constantes qui sont prédéfinies dans le langage utilisé. Celles-ci peuvent être utilisées par le programmeur, mais jamais redéfinies. En fonction du compilateur elles sont plus ou moins nombreuses. En voici quelques-unes :

nom	**valeur**
`__LINE__`	numéro de ligne courante du fichier
`__FILE__`	nom du fichier source courant
`__DATE__`	date de la compilation
`__TIME__`	heure de la compilation
`__CPLUSPLUS__`	vaut 1 si le source courant est en C++

- ✓ **Macros** :
  Syntaxe : `#define nom_macro(liste_arg) expression`
  L'expression est substituée à la macro dans le code source, en remplaçant les arguments en lieu et place dans l'expression par ceux passés à l'identifiant.

- ✓ **Insertion de fichiers** :
  Syntaxe : `#include nom_fichier`
  Insère un fichier à l'endroit où la directive est placée. Si le nom du fichier est encadré par des doubles quotes, la recherche de ce fichier se fait dans le répertoire courant. Si ce nom est encadré par des chevrons `< ... >` la recherche est faite dans les répertoires définis par une variable d'environnement (définie par le système d'exploitation, telle que INCLUDE_PATH par exemple).

- ✓ **Compilation conditionnelle** :
  Syntaxe : `#ifdef symbole bloc1 #else bloc2 #endif`
  Syntaxe : `#ifdef symbole1 bloc1 #elif symbole2 bloc2 #else bloc3 #endif`
  Syntaxe : `#ifndef symbole bloc1 #else bloc2 #endif`
  On peut activer ou désactiver tel ou tel bloc de code en fonction de la présence ou non de symboles prédéfinis.

- ✓ **Annulation d'une définition** :
  Syntaxe : **`#undef`** `nom_symbole`
  Cette directive permet d'annuler la définition d'une constante ou d'une macro.

- ✓ **Opérateur #** :
  Si un argument d'une macro est précédé du signe `#`, l'argument transmis est inséré comme une chaîne de caractères. Plus aucune substitution n'est effectuée sur cette chaîne, même si celle-ci contient des noms de macro ou de symboles. Si le résultat contient plusieurs chaînes qui se suivent, elles sont fusionnées en une seule. Par exemple :

```
#define CHAINE(a,b) "chaîne a :" #a " chaîne b :" #b
cout << CHAINE(FOO1,FOO2);
//appel: a = "FOO1" et b = "FOO2"
//puis : "chaîne a :" "FOO1" " chaîne b :" "FOO2"
//puis : "chaîne a :FOO1 chaîne b :FOO2"
```

- ✓ **Opérateur ## :**
  L'opérateur `##` permet de concaténer des chaînes de caractères. Il élimine de plus les espaces entre les chaînes concaténées. Le résultat étant une chaîne, plus aucune substitution n'est réalisée. Par exemple :

```
#define pref(a) p ## a
#define valeur Z
#define pvaleur resultat

cout << pref(valeur); // appel : a = "valeur"
// puis : pref(a) = "pvaleur"
// enfin : pref(a) = resultat
```

- ✓ **Message d'erreur** :
  Syntaxe : **`#error`** `chaine_a_afficher`
  On peut faire émettre des messages d'erreur au préprocesseur, qui arrête le processus. Cela permet par exemple de détecter un fonctionnement incohérent suivant des symboles. Lorsque le préprocesseur rencontre cette directive, il affiche la chaîne de caractères et rend la main.

3.3 Les commentaires

- ✓ On a la possibilité (et le devoir !) de commenter le code source.
- ✓ En C les commentaires se trouve entre **`/*`** et **`*/`**. Par exemple :
  `/* commentaire */`
- ✓ En C++ on peut commenter la fin d'un ligne avec **`//`** .Par Exemple :
  `int MonEntier=0; // Déclaration de mon entier`
- ✓ La plupart des éditeurs de code source reconnaissent les commentaires et offre la possibilité de les visualiser différemment (autre couleur…)

3.4 Les types prédéfinis

- ✓ C/C++ est un langage typé . Cela signifie que chaque entité (fonction, objet variable…) a un type. Grâce à ce type le compilateur peut vérifier la validité des opérations et produire des liens entre opérations et entités.

- ✓ **`void`** : le type vide. Utilisé pour faire des procédures et des pointeurs sur des données non typées.
- ✓ **`char`** : les caractères.
- ✓ **`int`** : les entiers dont la taille (et donc la plage de valeur) peut varier en utilisant `short` et `long`. On peut aussi avoir une plage de valeur des entiers positif ou nul avec **`unsigned`**, ou une plage de valeur centrée sur 0 avec **`signed`** (aussi utilisable avec le type `char`)
- ✓ **`float`** : les réels
- ✓ **`double`** : les réels en double précision
- ✓ Notez qu'en C/C++ il n'y a pas de type de base pour les chaînes de caractères. Elles sont représenté comme un tableau dont les éléments sont des caractères.
- ✓ La taille des instances des types n'est pas spécifiée dans la norme. Cependant la plupart du temps, sur un processeur 32 bits on a :
  - o `char` 1 byte utilise la norme ASCII
  - o `short int` 2 bytes `signed` [-32768,+32767] `unsigned` [0,65535]
  - o `long int` 4 bytes `signed` [-2 147 483 648 , +2 147 483 647]
    `unsigned` [0,4 294 967 295]
  - o `float` 4 bytes [3.4 E-38, 3.4 E+38]
  - o `double` 8 bytes [1.7 E-308, 1.7 E+308].

3.5 Les structures de contrôle

- ✓ La structure conditionnelle : **`if`**`(condition) opération` **`else`** `opération`
  - o Les opérateurs de comparaison sont : `{==, !=,<,>,<=,>=}` ATTENTION à ne pas utiliser = au lieux de ==.
  - o Les opérateurs logique : `{&&,||,!}`
- ✓ La boucle **`for`**`(initialisation ;test ;itération) opération.`
- ✓ La boucle **`while`**`(condition) opération`
- ✓ La boucle **`do`** `opération` **`while`**`(condition)`
- ✓ Le branchement conditionnel :
  `switch``(valeur)` **`case`** `cas1 : opération . . .` **`default`** `opération`
- ✓ Le saut **`goto`** `etiquette . . . etiquette:`
- ✓ Les commandes de rupture de séquence :
  - o **`continue`** permet de terminer immédiatement la boucle courante d'un `while do for,` mais de continuer l'enchaînement des boucles.
  - o **`break`** permet de sortir d'un `while do for switch`
  - o **`return`** permet de sortir de la fonction courante en communiquant éventuellement la valeur de retour.

3.6 Les variables

- ✓ On peut simplement déclarer une variable avec la syntaxe :

```
type nom_de_variable
```

- ✓ Lors de la déclaration on peut affecter une valeur à la variable (ceci est fortement conseillé pour TOUTES les variables) :

```
type nom_de_variable = valeur
```

- ✓ La variable est physiquement allouée sur la pile. Elle est désallouée lorsque l'exécution du programme sort de la portée courante (i.e rencontre le }).
- ✓ La variable est visible dans les sous-portées de la portée ou elle a été crée. Elle peut être masquée, dans une sous portée, par une autre variable de même nom mais ceci est à proscrire absolument, sous peine d'avoir un code illisible.
- ✓ On peut aussi définir des variables hors de toutes les portées. Ce sont des **variables globales**. En général l'utilisation de variables globales montre une faiblesse du design de l'application. **EN GENERAL, IL NE FAUT PAS UTILISER DE VARIABLES GLOBALES.**
- ✓ La **classe de stockage** d'une variable permet de spécifier sa durée de vie et sa place en mémoire. Voici les classes de stockage en C :
  - o **`auto`** : la classe de stockage par défaut. Ce mot clé est facultatif.
  - o **`extern`** : permet d'utiliser une variable dans un autre fichier C/C++ que celui ou elle est déclarée.
  - o **`static`** : cette classe de stockage permet de créer des variables dont la portée est le bloc d'instructions en cours, mais qui ne sont pas détruites à la sortie de ce bloc. Les variables statiques gardent leurs valeurs pour la prochaine fois où l'on passe dans le bloc. Ca peut être utile mais assez dangereux, surtout dans un environnement multi threaded. Généralement il vaut mieux éviter cette facilité. On peut aussi utiliser ce mot clé devant une variable ou une fonction globale. Elles seront spécifiques au fichier courant et ne pourront pas être utilisées dans un autre fichier (même avec le mot clé `extern` pour les variables).
  - o **`register`** : permet d'indiquer au compilateur que la variable doit être stockée dans un registre du processeur, pour plus de rapidité. Ne doit être utilisé que si l'on connaît bien le langage machine.
  - o **`volatile`** : indique que cette variable doit être rechargée par le processeur à chaque accès, car elle a pu être modifiée par un autre thread. En général on n'utilise ceci que lors de la programmation système.

3.7 Les fonctions

- ✓ Le langage C autorise de séparer la **déclaration** et la **définition** (i.e le corps) **d'une fonction**. Ceci est constamment utilisé, notamment dans les fichiers d'en-tête, pour appeler une fonction dans un fichier C/C++ autre que celui où est le corps de la fonction. En effet une fonction doit déjà être déclarée (mais pas forcément définie) pour pouvoir être appelée.
- ✓ La syntaxe de la déclaration d'une fonction est la suivante :

```
type_de_retour(falcutatif) Nom_de_la_fonction(
            type_parametre1 parametre1,
            . . .
            type_parametreN parametreN);
```

- ✓ La syntaxe du corps de la fonction est la suivante :

```
type_de_retour(falcutatif) Nom_de_la_fonction(
            type_parametre1 parametre1,
            . . .
            type_parametreN parametreN)
{ Ici corps de la fonction }
```

- ✓ L'appel de fonction en C/C++ supporte la **récursivité** (i.e une fonction peut s'appeler).
- ✓ **Les paramètres sont passés par valeur** (i.e la valeur du paramètre est copiée dans une variable spécialement allouée sur la pile lors de l'appel de la fonction).
- ✓ **On peut simuler un passage par variable** (aussi nommé passage par référence) en utilisant un pointeur sur la variable à passer.

3.8 Les opérateurs

- ✓ Le C dispose des 4 fonctions de l'arithmétique : '**+**' '**-**' '*****' '**/**'
- ✓ Sur les types entiers on peut utiliser l'opérateur modulo (reste de la division, ou congruence) : '**%**'
- ✓ Il y a aussi les opérations binaires : '**|**' est le ou binaire, '**&**' est le et binaire, '**^**' le ou exclusif binaire, '**~**' la négation binaire, '**<<**' et '**>>**' représente un décalage de bits vers la gauche ou vers la droite du nombre de bits égal à la valeur de la seconde opérande.
- ✓ Il y a les opérateurs d'incrémentation '**++**' et de décrémentation '- -', préfixés ou suffixés :

```
int i=2;
int j = ++i;  // après cette instruction, i=3 et j=3
int k = i++;  // après cette instruction, i=4 et k=3
```

- ✓ Il y a les opérateurs d'affectation composées :

```
int i=6;
i += 5;  // après cette instruction, i=11
i -= 1;  // après cette instruction, i=10
i /= 2 ; // après cette instruction, i=5
i op= valeur   i = i op valeur
```

- ✓ Enfin il y a l'opérateur ternaire d'évaluation conditionnelle ?:dont voici un exemple :

```
Test ? expression si vrai : expression si fausse
int i = ( (5 > 3) ? 6 : 7 ); // après cette instruction i = 6
```

3.9 La fonction main()

- ✓ Un programme C/C++ commence toujours par l'exécution de la fonction **int main()**, qui accepte pour paramètres (facultatifs) les chaînes de caractères passées en ligne de commandes, lors de l'appel du programme comme suit :
  - o 1er paramétre : un entier, égal au nombre de paramètres.
  - o 2eme paramètre : un tableau de chaînes de caractères (la première chaîne étant le nom du programme). Voir la section sur les tableaux, 3.11.
- ✓ La fonction retourne un entier qui est le code de retour du programme.

3.10 Les pointeurs

- ✓ Chaque variable admet une adresse mémoire qui la localise physiquement dans le programme (ou processus).
- ✓ Pour tous type de variable, il existe un type dual, le type pointeur du type concerné. Une variable de type pointeur est en fait l'adresse d'une variable du type concerné.

- ✓ Un pointeur peut ne pointer sur rien du tout. Dans ce cas il est **EXTREMEMENT IMPORTANT** que sa valeur soit nulle (0). En effet la majorité des bugs des programmes C/C++ viennent de pointeurs non nuls qui ne pointent sur aucune variable valide.
- ✓ La déclaration d'un pointeur sur le type `toto` se fait comme suit :

```
toto * pointeur
```

Par exemple :

```
long * pUnEntier = 0;
```

- ✓ On peut obtenir un pointeur sur une variable par **l'opérateur d'indirection &**.
Par exemple :

```
long UnEntier = 98;
long * pUnEntier = &UnEntier;
```

- ✓ On peut accéder à l'objet pointer par **l'opérateur de déréférencement *.**
Par exemple :

```
long UnEntier = 98;
long * pUnEntier = &UnEntier;
long UnAutreEntier = *pUnEntier;
// UnAutreEntier vaut ici 98
```

- ✓ On utilise couramment un pointeur vers un pointeur, (notamment dans les tableau voir 3.11). On appelle ceci un **double pointeur.** Par exemple :

```
long UnEntier = 98;
long * pUnEntier = &UnEntier;
long ** ppUnEntier = &pUnEntier ;
```

- ✓ Il est très important d'avoir une convention de nommage des pointeurs et des doubles pointeurs. En général, pour le nom d'un pointeur on a un p minuscule comme premier caractère, et pp pour un double pointeur. Dans les programme Microsoft on utilise LP (Long Pointer).

3.11 Les tableaux (arrays)

- ✓ On peut déclarer un tableau à une dimension de variables de même type. Par exemple :

```
long  TableauDentier[13] ;
```

- ✓ La taille du tableau ne peut être fixée dynamiquement.
- ✓ L'accès au premier élément est :

```
long  PremierEntier = TableauDentier[0];
```

D'où l'accès au dernier élément s'écrit :

```
long  DernierEntier = TableauDentier[12]; // ATTENTION PAS 13 !!
```

- ✓ En interne, les variables du tableau sont stockées les une à la suite des autres. Ce qui implique qu'un tableau est un pointeur sur le premier élément :

```
long  PremierEntier = TableauDentier[0];
```
équivalent à :
```
long  PremierEntier = *TableauDentier;
```

- ✓ Il existe une arithmétique des pointeurs :
p + i = Adresse contenue dans p + i * taille en octets d'un élément de p, d'où :
```
long  DernierEntier = *(TableauDentier+12);
```
p2-p1 = Adresse contenue dans p2 – Adresse contenue dans p1 / taille des éléments pointés par p1 et p2 en octets.
- ✓ Notez que la fonction C **`sizeof()`** retourne la taille d'un type ou d'une variable en octets :

```
long Entier = 0;
int taille1 = sizeof(Entier);  // taille 1 = 4
int taille2 = sizeof(long);    // taille 2 = 4
```

- ✓ On peut initialiser un tableau comme suit :

```
long  TableauDentier[4] = {23 , 45, 78 , 98} ;
```

- ✓ Pour créer des tableaux à 2 dimensions il faut faire des tableaux de tableaux. Pour gérer ceci on utilise les doubles pointeurs :

```
char * TableauDeChainesDeCaracteres[5] ;
// TableauDeChainesDeCaracteres est de type char **
```

- ✓ Enfin notez que C ne s'aperçoit ni à la compilation, ni à l'exécution du dépassement des limites d'un tableau. C'est donc au programmeur d'être très vigilant.

3.12 Les structures

- ✓ La définition de types complexes passe par la notion de structure. La syntaxe est :

```
struct [nom de structure]
{
      type nom ;
      . . .
      type nom ;
} ;
```

- ✓ Le nom de la structure est facultatif.
- ✓ Voici un exemple de déclaration et d'utilisation de structure :

```
struct Employe
{
      unsigned char     Age;
      long              Salaire;
};
struct Employe Fabrice;//struct nécessaire en C mais pas en C++
Fabrice.Age = 32 ;
Fabrice.Salaire = 3000 ;
```

- ✓ On peut aussi définir des champs de bits dans les structures. La façon dont les bits sont stockés dépend du compilateur :

```
struct ChampsDeBits
{
      unsigned char     Drapeaux:1;
      unsigned char     Valeur :6;
};
struct ChampsDeBits MonChamps;
MonChamps.Drapeaux = 1 ;  // doit être soit 0 soit 1
MonChamps.Valeur = 23 ;   // doit être entre 0 et 63
```

3.13 Les unions

- ✓ Les unions ont la même syntaxe que les structures avec le mot clé `union` à la place de `struct`. La différence est que les champs d'une union occupent le même espace mémoire. Voici un exemple :

```
union entier_ou_reel
{
      long  entier;
      float reel;
};
union entier_ou_reel MonUnion;
MonUnion.entier = 876 ;
MonUnion.reel   = 3.1415 ;   // écrase la valeur de 'entier'
```

- ✓ Contrairement aux structures les unions sont très peu utilisées.

3.14 Les énumérations

- ✓ Les énumérations sont des types basés sur les entiers. On a la possibilité de nommer les entiers. Par exemple :

```
enum LesEntiers{un = 1,deux = 2 ,trois ,quatre} ;
```

- ✓ Notez qu'ici la valeur entière des mots `trois` et `quatre` sont choisies par le compilateur.
- ✓ La déclaration de variables de type énumération suit les mêmes règles que la déclaration d'unions et ou de structures.

3.15 Les alias de type

- ✓ On peut définir des alias de type pour simplifier les types complexes.
- ✓ La syntaxe est la suivante. Imaginer que vous déclarez une variable d'un type. Alors il suffit de mettre le mot clé **`typedef`** devant la déclaration et vous avez un alias de type dont le nom est le nom de la supposée variable. Par exemple :

```
typedef unsigned long EntierNonSigné;
EntierNonSigné UnEntierPositif = 98764;

typedef long TypeTableauDeTreizeEntiers[13];
TypeTableauDeTreizeEntiers  TableauDentier;
TableauDentier[12] = 7;
```

3.16 Le transtypage (cast)

- ✓ On peut en C convertir explicitement une variable d'un type vers un autre type:

```
short Entier1 = 3;
short Entier2 = 2;
float Division1 = Entier1/Entier2;  // Division1 vaut 1.00000
float Division2 = ( (float) Entier1)/( (float) Entier2);
// Division2 vaut 1.50000
```

- ✓ Pour la première division le compilateur fait appel à la division des entiers, puis le résultat est transtypé en flottant. Pour la deuxième division le compilateur fait appel à la division des flottants et le résultat est lui-même un flottant.
- ✓ Notez que le transtypage du résultat de la première division, d'un entier en un flottant, est automatique. Bien que très pratique le transtypage automatique peut se révéler dangereux, et mener à des résultats erronés.

3.17 Les pointeurs sur fonction

- ✓ On peut définir un pointeur sur une fonction. L'idée est de pouvoir appeler la fonction à travers le pointeur.
- ✓ Voici un exemple pour illustrer la syntaxe :

```
void (*pSurFonction)(short,double);
void  f1(short,double);
void  f2(short,float);
short f3(short,double);
void f()
{
 pSurFonction = &f1;    // Assignement du pointeur pSurFonction
 pSurFonction(34,78.9);  // Appel de f1()
 pSurFonction = &f2 ;   // erreur de compilation
 pSurFonction = &f3 ;   // erreur de compilation
}
```

- ✓ On aurait pu définir un alias sur ce type de pointeur de fonction :

```
typedef void (*TypepFonc)(short,double);
void  f1(short,double);
void f()
{
 TypepFonc pSurFonction = &f1;
 pSurFonction(34,78.9);  // Appel de f1()
}
```

- ✓ Il peut être efficace d'utiliser les pointeurs sur fonctions pour obtenir une certaine généricité. Par exemple un algorithme de tri repose entièrement sur la fonction de comparaison des éléments. On peut donc rendre générique un algorithme de comparaison en lui fournissant un pointeur sur la fonction de comparaison des éléments.

3.18 L'allocation mémoire

- ✓ Il existe une fonction pour allouer de la mémoire au système d'exploitation (**`void* malloc(int taille_en_octet)`**) et une pour la lui restituer (**`void free(void * pointeur_retourné_par_malloc)`**).

- ✓ Elles utilisent des pointeurs, puisque l'espace mémoire n'est pas connu à la compilation et ne peux donc pas être identifié.
- ✓ De plus ces pointeurs n'ont pas de type (type `void`).
- ✓ Pour utiliser ces fonctions il faut inclure le fichier `stdlib.h`.
- ✓ Si on veut typer la mémoire allouées (par exemple on alloue 4 bytes pour un `long`) il faut transtyper le pointeur retourné par `malloc()`. Par exemple :

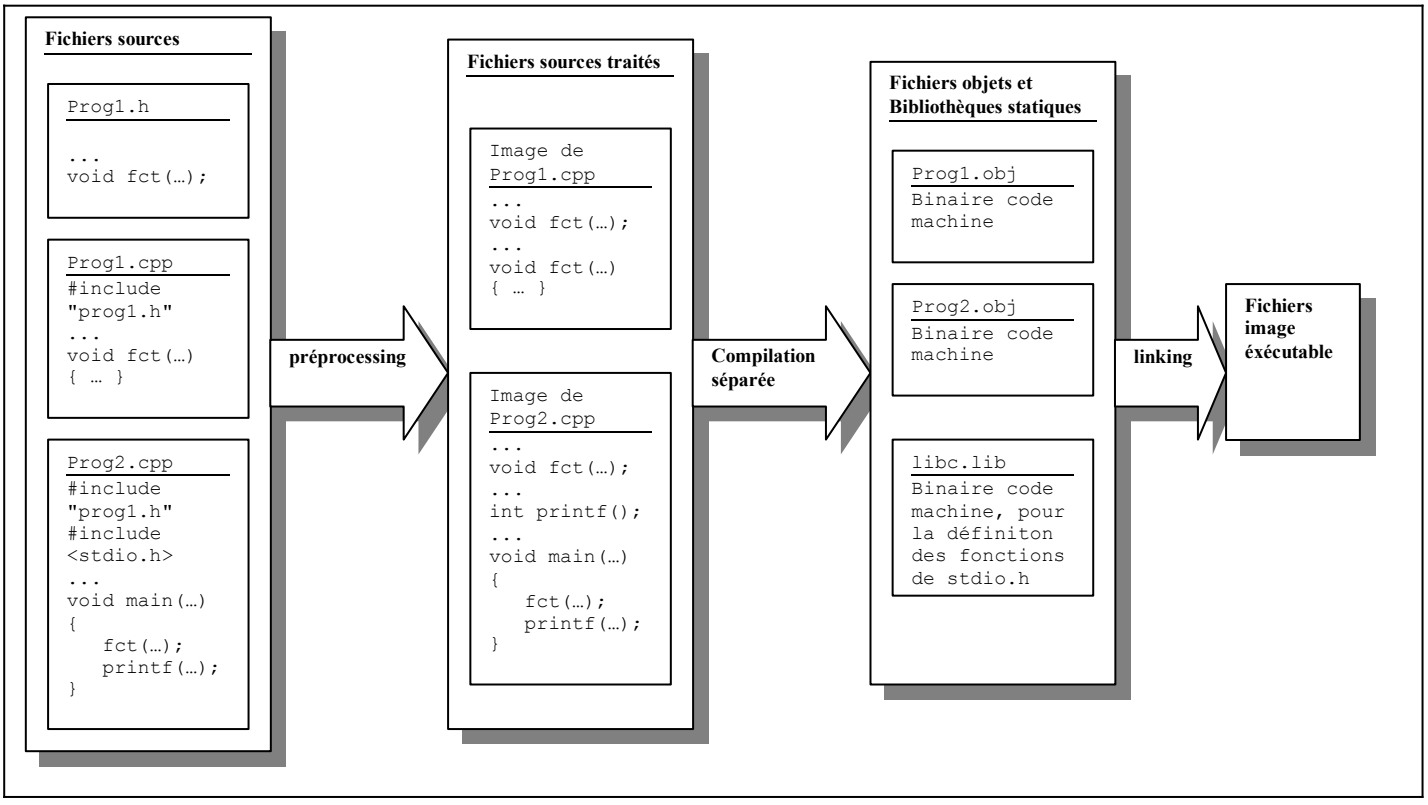
```
#include <stdlib.h>
long * pEntier =(long*) malloc(4); // ou malloc(sizeof(long));
*pEntier = 98;
free(pEntier);
// ici pEntier pointe sur rien et est potentiellement dangereux
pEntier = 0;
```

- ✓ Notez que le pointeur est positionné à 0 pour éviter qu'on accède à de la mémoire désallouée.
- ✓ Il faut faire très attention à toujours libérer la mémoire allouée, car sinon le programme 'gonfle' et risque de planter ou même de faire planter la machine. Ce problème est connu sous le nom de '**fuite de mémoire**' (**memory leak** en anglais) est constitue un problème majeur dans le développement d'applications en C/C++. Il existe des outils permettant de détecter automatiquement les fuites mémoire.
- ✓ D'autres langages comme Java ou C# n'ont pas ce problème puisque la mémoire est désallouée automatiquement lorsque le système prouve qu'on ne peut plus y avoir accès. Ce processus est le **ramasse miette** (**garbage collector** en anglais).

4 La compilation

4.1 Etapes de la compilation

- ✓ Au départ on ne dispose que des fichiers sources C/C++.
- ✓ La première étape est le **traitement des fichiers sources avant compilation** (*préprocessing* en anglais) (le *préprocesseur* en C/C++ voir 3.2). Le préprocesseur fabrique de nouveaux fichiers sources C/C++, avec les macros remplacées, les commentaires supprimés, les fichiers d'entête inclus…
- ✓ La deuxième étape est la **compilation séparée**. Le résultat pour chaque fichier source compilé est un *fichier objet*. Un fichier objet contient la traduction en langage machine du code du fichier source correspondant. Les bibliothèques statiques utilisées pour inclure des fonctionnalités aux programmes sont des fichiers objets.
- ✓ La dernière étape est **l'édition de lien** (linking en anglais). On y effectue le regroupement de tous le code des fichiers objets (sources et bibliothèque), et la résolution des références inter-fichier. Le produit est un *fichier image* exécutable par la machine.
- ✓ En résumé :



4.2 Les outils de compilation et l'optimisation

- ✓ Toutes les étapes de la compilation peuvent être réunies soit au niveau du compilateur, soit au niveau d'un programme appelé **make**.
- ✓ Le principe de make est le suivant :
  - o Lecture séquentielle dans le fichier makefile des opérations à effectuer.
  - o Décision ou non d'exécution pour chaque opération.
- ✓ En effet on peut optimiser la compilation au moyen des techniques suivantes :
  - o D'une compilation à une autre, il se peut que seul quelques fichiers sources aient été modifiés. Dans ce cas il est profitable de garder les fichiers objets correspondants aux fichiers sources inchangés. Notez que plus les déclarations sont fragmentées sur plusieurs fichiers d'entête, plus cette technique sera efficace. En effet la modification d'un fichier d'entête inclus par tous les fichiers sources implique une re-compilation totale. Cela confirme un des principes fondamental du C/C++ : **chaque fichier source ne doit inclure que strictement ce qu'il a besoin.**
  - o La plupart des compilateurs réduisent l'édition de liens en sauvant des informations dans le fichier image exécutables. L'éditeur de lien ne travaille ainsi qu'avec les fichiers objets modifiés depuis la dernière édition de lien. Notez que tous ceci se base sur la date de dernière modification des fichiers, d'ou il est important d'avoir l'OS à l'heure.
- ✓ La syntaxe des opérations dans le fichier makefile n'est pas standardisée et en général, différente d'un système à un autre. Elle ne sera donc pas abordée ici. Voir par exemple **[Casteyde]** pour plus de détails.
- ✓ De plus certains environnements de développement comme Visual C++ masquent complètement le fichier makefile, au moyen de menus optionnels plus ou moins intuitifs.

4.3 Contraintes de la compilation séparée

- ✓ Les entités suivantes doivent toujours être déclarées (mais pas forcément définis) avant utilisation :
  - o Les types (ou classes en C++)
  - o Les variables
  - o Les fonctions

  L'éditeur de lien se charge de faire correspondre les définitions de ces entités, aux endroits ou elles sont utilisées.
- ✓ Les types et classes peuvent être déclarés comme suit :

```
struct UnTypeDeStructure;
class CMaClasse;
```

- ✓ Les variables globales (à éviter cependant) définies dans fichier source peuvent être utilisées dans un autre fichier source en utilisant le mot clé `extern` :

```
extern UneVariableGlobale;
```

- ✓ Les fonctions sont déclarées comme vu en 3.7.

4.4 Les compilateurs disponibles

- ✓ Il faut savoir que les mécanismes les plus avancés du C++ (et aussi les moins utilisés) ne sont pas tous supportés par les différents compilateurs.
- ✓ De part leurs complexités, ces mécanismes interviennent nécessairement dans des cas de figure complexes rares. Malgré leurs lacunes d'implémentation de la norme, les compilateurs restent donc tout à fait utilisables.
- ✓ Les compilateurs les plus courants sont :
  - o ***egcs*** version 1.1.2 de la Free Software Foundation distribué avec Red Hat Linux V5.1 (fin 98).
  - o Microsoft ***Visual C++*** 6.0 (début 99) 7.0 (début 02)
  - o Borland ***C++ Builder*** v3.0 (fin 98).
- ✓ *egcs* est le plus proche de la norme malgré certaines lacunes.
- ✓ Il n'en est pas de même pour *Visual C++* et *C++ Builder* qui parfois transgressent allègrement la norme. Par exemple l'utilisation du RTTI (voir 7.3) est facultative dans *Visual C++* !!!
- ✓ En général on utilise *egcs* sous les systèmes type Unix et *Visual C++* sous les systèmes type Windows.

5 Les mécanismes simples de C++

5.1 Notion de référence

5.1.1 Référence sur une variable

- ✓ Une référence est un nom alternatif sur une variable.
- ✓ Toutes les opérations sur une référence agissent sur la variable référencée :

```
int i = 1;
int & r = i; // r et i désigne maintenant le même entier
i=2; // r = 2 et i = 2
r=3; // r = 3 et i = 3
i++; // r = 4 et i = 4
r--; // r = 3 et i = 3
```

- ✓ Le compilateur s'assure que les références sont toujours initialisées :

```
int i = 1;
int & r1 = i; // OK r1 est initialisé
int & r2;     // erreur de compilation : r2 non initialisé
extern int & r3; // OK r3 est initialisé dans un autre fichier
```

- ✓ **Une fois initialisée une référence ne peut plus changée**, puisque tous les opérateurs agissant sur la référence, agissent en fait sur la variable référencée.
- ✓ On peut considérer une référence comme un pointeur constant, qui est déréférencé à chaque utilisation. De plus, contrairement aux pointeurs, les références sont toujours initialisées. Les deux programmes suivants sont équivalents (mis à part que le pointeur n'est pas constant):

```
int i = 1;          int i = 1;
int &r = i;         int *p = &i;
r++;                (*p)++;
r=4;                (*p)=4;
```

- ✓ Les deux utilisations principales des références sont:
  - o Le passage par références des paramètres des fonctions (voir ci après).
  - o Les opérateurs surchargés (voir 6.7).
- ✓ On utilise très peu les références dans le corps d'un programme.

5.1.2 Le passage par références des paramètres des fonctions

- ✓ Comme on l'a vu dans 3.7 le C n'autorise que le passage par valeurs des paramètres d'une fonction. On peut cependant passer la variable elle même et non une copie, en utilisant un pointeur sur la variable.
- ✓ Le C++ autorise le passage des paramètres à une fonction par références. Il suffit de préciser lors de la déclaration de la fonction et dans son prototype l'utilisation des références pour qu'on puisse bénéficier du passage par adresse en conservant l'écriture du passage par valeur. Par exemple :

```
void fct( int & ParamParRef, int ParamParVal , int * pParam)
{
      ParamParRef = 1;
      ParamParVal = 2;
      pParam = 3 ;
}
int main()
{
      int i = 4;
      int j = 5;
      int k = 6 ;
      fct(i,j,&k);
      // ici i = 1 , j = 5 , k = 3
}
```

- ✓ Attention toutefois au danger de cette écriture pour l'utilisateur. Si on ne connaît pas le prototype de `fct()` on ne peut savoir que la valeur de i a été potentiellement changée. Cela constitue la faiblesse principale du mécanisme de références face au mécanisme de pointeurs, puisque dans ce dernier cas, le client de la fonction a l'information du changement potentiel.
- ✓ Comme on l'a vu le principal avantage du passage de la variable elle-même est la performance, puisqu'il n'y a pas de recopie. On peut donc très bien vouloir passer une variable par référence sans vouloir la modifier. On peut obliger le compilateur à vérifier si un paramètre est effectivement non modifié avec le mot clé **`const`** :

```
void fct( const int & ParamParRef)
{
      ParamParRef = 1; // <- erreur de compilation !!!
}
```

5.2 Surcharge (ou surdéfinition) d'une fonction

- ✓ On parle de surdéfinition ou de surcharge lorsqu'un même symbole possède plusieurs significations, la sélection se faisant en fonction du contexte. Par exemple, l'opérateur "+" est surdéfini. On emploie le même opérateur, qu'il s'agisse d'une addition entre des entiers ou d'une addition entre des réels.
- ✓ Dans le cas de fonctions, il peut y avoir plusieurs fonctions avec le même nom, mais une signature différente (le paramètre de retour n'est pas pris en compte):

```
char fct(int a);
char fct(double a);
char fct(int a, int b);
int  fct(int a, int b); // erreur de compilation: seul le
                        // paramètre de retour change
```

- ✓ Le compilateur choisit quelle fonction appeler d'après le contexte d'appel :

```
void fct(int a);
void fct(double a);
. . .
fct( 3.5 ); // appelle la deuxième fonction
int i = 2 ;
fct( i ); // appelle la première fonction
```

- ✓ Attention, les types suivant sont considérés comme équivalents :
  - o Un type et une référence sur un type .
  - o L'ensemble d'un type et ses alias (voir 3.15) .
  - o Un pointeur sur un type et un tableau d'éléments de ce même type (voir 3.11).
- ✓ Certains cas peuvent être considérés comme ambiguë. Dans ces cas le compilateur choisit les types les plus étendus.

```
void fct( long  a , double b );
void fct( short a , float  b );
fct( 1, 6.7); // appel la première fonction, long plus étendu que
              // short et double plus étendu que float
```

Cependant dans l'exemple suivant le compilateur ne peut plus exactement appliquer cette règle, et applique d'autres règles détaillées dans [Stroustrup] :

```
void fct( long  a , float  b );
void fct( short a , double b );
fct( 1, 6.7); // qui appeler ?
```

- ✓ Le mécanisme de surcharge de fonction est puissant et très utilisé, mais nous déconseillons son utilisation lorsqu'il faut aller plus en détails dans les règles du compilateur. En effet, le code deviendrait vite illisible pour la plupart des programmeurs.

5.3 Arguments par défaut d'une fonction

- ✓ Le C++ permet de donner une valeur par défaut à des arguments d'une fonction.
- ✓ Par défaut dans le sens où l'appelant de la fonctionne leur affecte pas de valeurs. En fait ceci peut être assimilé à la surcharge d'une fonction où les arguments manquant ont une valeur par défaut. Un exemple :

```
f(long a , long b , long c = 0 , long d = 0);
...
f(2,3,4,5);
f(2,3,4); // équivalent à f(2,3,4,0)
f(2,3);   // équivalent à f(2,3,0,0)
f(2);     // erreur de compilation
```

- ✓ Notez que les arguments par défaut sont nécessairement les arguments à droite dans la déclaration des arguments d'une fonction. Sinon le compilateur ne pourrait pas s'en sortir dans certains cas.

5.4 Les types bool et wchar_t

- ✓ Le C++ introduit le type **`bool`** pour définir les booléens.
- ✓ En interne une variable de type `bool` est un char. Le booléen est considéré faux si le caractère est égal à 0, et vrai pour les 255 autres valeurs.
- ✓ Le C++ réserve les 2 mots clés **`true`** et **`false`** pour affecter une valeur à une variable de type `bool`. Par exemple :

```
bool b1 = true;
int i=2, j=3;
bool b2 = (i>j);
if( b2 ) ...
```

- ✓ Le C++ introduit le type **`wchar_t`.** Il représente un ensemble étendu de caractères, pouvant contenir de nombreux jeux de caractères locaux.
- ✓ En C++ `wchar_t` est un mot clé réservé et non un simple `typedef` comme en C (dans le fichier `<stddef.h>`).

5.5 Utilisation des variables

- ✓ L'utilisation des variables en langage C++est très semblable à celle du langage C.
- ✓ Cependant la déclaration des variables en C++ a été assoupli, dans la mesure où elle peut se faire à l'endroit où l'on a besoin de la variable (et pas seulement au début des blocs comme c'est le cas en langage C). La portée d'une variable reste limitée au bloc contenant sa déclaration.

5.6 Les fonctions inline

- ✓ Le compilateur traduit un appel de fonction par un saut en mémoire vers la zone de code correspondant à la fonction. L'avantage est que le code d'une fonction n'est présent qu'une seule fois en mémoire, et peut être appelé autant de fois qu'il le faut. Le désavantage est que cela nécessite une gestion interne assez lourde (le passage des arguments sur la pile…), responsable d'une certaine lenteur d'exécution.

- ✓ La définition d'une fonction inline (mettre le mot clé `inline` devant) permet au compilateur de recopier le code de la fonction aux points d'appel au lieu de générer une instruction de saut. Le programme est donc plus rapide mais aussi plus gros.

- ✓ De plus les fonctions inline doivent respecter certaines contraintes :

  - o Elles ne doivent pas être récursives.

  - o On ne peut pas obtenir de pointeur sur ces fonctions.

  - o Elles doivent être complètement définies avant leur appel. Il vaut donc mieux écrire le corps d'une fonction inline avec sa déclaration.

- ✓ Enfin il vaut mieux respecter ces deux règles avant de mettre une fonction inline :

  - o Le corps de la fonction doit être très court (4 ou 5 instructions).

```
inline long max( long  a , long  b ) { return ((a>b) ? a : b);}
```

  - o On peut tolérer plus d'instruction s'il y a beaucoup de paramètres.

5.7 Le mot clé const et l'opérateur const_cast( )

- ✓ On a vu que lors du passage par références des paramètres d'une fonction (5.1.2), que l'on peut rendre constant dans le corps de la fonction un paramètre de la fonction. Il suffit de mettre le mot clé **`const`** devant la déclaration.
- ✓ Cette possibilité peut s'étendre à toutes déclarations de variables.
- ✓ Dans le cas de constantes globales (comme les constantes mathématiques) il vaut mieux utiliser une variable constante qu'un `#define`. En effet, la variable étant typée, le compilateur peut détecter des erreurs.

```
#define PI 3.14159265358979323 // à proscrire !!
const double PI = 3.14159265358979323;  // à faire
```

- ✓ Le type const T est considéré comme différent du type T. Le compilateur autorise le transtypage du type T vers const T mais pas l'inverse :

```
void f1( const long & a );
void f2( long & a );
. . .
const long i = 2;
f1(i);
f2(i); // erreur de compilation : impossible de convertir du type
       //'const long' vers 'long'
```

- ✓ L'utilisation des pointeurs avec le mot clé `const` doit être bien compris. En effet un pointeur peut être constant et/ou la variable pointée peut être constante. Voici la syntaxe à utiliser :

```
long a = 6, b=7;
long * const p1 = &a;
p1 = &b;    // erreur de compilation, p1 est un pointeur constant
```

```
*p1 = 2;    // OK

const long * p2 = &a;
p2 = &b;    // OK
*p2 = 2;    // erreur de compilation, p2 est un pointeur
                  // sur une constante

const long * const p3 = &a;
p3 = &b;    // erreur de compilation, p3 est un pointeur constant
*p3 = 2;    // erreur de compilation, p3 est un pointeur
                  // sur une constante
```

- ✓ L'opérateur **`const_cast`** est un opérateur de transtypage C++ qui permet de réduire la contrainte d'un pointeur ou d'une référence vis-à-vis des mots clés `const` et `volatile`.
- ✓ Sa syntaxe est `const_cast<type_cible>(expression)` ou `expression` est un pointeur ou une référence, et type cible le type moins contraint.
- ✓ Notez que `const_cast` ne peut être utilisé pour les transtypage C/C++ courant comme la conversion d'un entier en un flottant. Voici un exemple :

```
const int i = 9;
int * p = const_cast<int*>(&i);
(*p)+=10; // après ceci i vaut 19
```

5.8 Le mot clé mutable

- ✓ Le mot clé **`mutable`** permet de rendre un champ d'une structure constante accessible en écriture. Voici un exemple :

```
struct T
{
long a;
mutable long b;
};
. . .
const T MaStructure = { 5,6 };
MaStructure.a = 7;  // erreur de compilation
MaStructure.b = 8;  // OK
```

6 La programmation orientée objet (POO) avec C++

6.1 Remarques sur la programmation objet

- ✓ Le concept de la **programmation fonctionnelle** est construit autour de la notion de fonction. Tout programme est un ensemble de fonctions s'appelant entre elles.
- ✓ Le concept de la **programmation objet** est construit autour des notions d'objet et de classe. **Une classe est un type de donnée** (au même titre que `int` ou une structure en C). **Un objet est une instance d'une classe**.
- ✓ Dans la programmation objet, tout programme peut être vu comme un ensemble d'objets qui interagissent entre eux.
- ✓ Avec sa compatibilité avec le C, le C++ ne pouvait être un langage objet (comme EIFFEL par exemple). En effet, le C est spécialisé pour la programmation fonctionnelle. Cependant le C++ intègre la plupart des concepts de la programmation objet. On parle de **programmation orientée objet ou POO**.
- ✓ Nous allons détailler comment C++ traite la POO. Cependant ce cours n'est pas un cours sur la POO. En effet, la plupart des concepts seront abordés, mais ça ne suffit pas pour devenir un bon architecte en POO.
- ✓ Pour avoir de bonnes notions d'architecture en POO nous vous conseillons **[Design Patterns]** (figures de style ou motifs en Français). Cette ouvrage rassemble une vingtaine de problématiques courantes dans l'architecture objet, et proposent des solutions élégantes.

6.2 Les classes

6.2.1 Notions et vocabulaire

- ✓ Une **classe** est un type de donnée, au même titre que `int` ou une structure en C.
- ✓ Les notions de variable d'un type et **d'instance d'une classe** sont en fait identiques.
- ✓ Un **objet** est une instance d'une classe. On peut donc le voir comme une variable.
- ✓ Une classe est principalement constituée de deux entités :
  - L'ensemble de ces **attributs**.
  - L'ensemble de ces **méthodes**.

  On les appelle les **membres** de la classe.
- ✓ Le notion d'**attribut** est identique à la notion de champs d'une structure. Chaque objet renferme donc une quantité d'information sous forme de variables, qui en fait sont elles aussi des objets.
- ✓ La notion de **méthode** se rapproche de celle de fonction, à ceci prés qu'une méthode est appelée sur un objet de la classe. Une méthode agit sur l'objet sur lequel elle est appelée. Elle peut aussi bien lire ou écrire les attributs, qu'effectuer une action sur l'objet.
- ✓ Tout comme les variables, les objets sont construits et détruits. Une méthode est automatiquement appelée lors de la construction. Elle est appelée **constructeur**. Elle permet par exemple d'initialiser l'objet, ou d'allouer des ressources. Il peut y avoir plusieurs constructeurs pour une même classe car il peut y avoir plusieurs façons de construire un objet.
- ✓ Une méthode est automatiquement appelée lors de la destruction d'un objet. C'est le **destructeur**. Par exemple il désalloue des ressources. Contrairement au constructeur il ne peut y avoir qu'un seul destructeur.

6.2.2 Déclaration de classes et l'opérateur de résolution de portée

- ✓ La déclaration d'une classe se fait comme suit :

```
class nom_de_la_classe
{
      type nom;
      . . .
      type nom;
      methode(…);
      . . .
      methode(…);
};
```

- ✓ Le code des méthodes peut être directement écrit à l'intérieur de la déclaration de la classe, ou à l'extérieur :

```
class UneClasse
{
 long MonAttribut;
 long MaMethode(long a) { return ( a + MonAttribut);}
};

class UneClasse
{
 long MonAttribut;
 long MaMethode(long a);
};
long UneClasse::MaMethode(long a) { return ( a + MonAttribut);}
```

- ✓ On appel l'opérateur `::` l'opérateur de résolution de porté. Si il n'y a pas de nom de classe devant le nom de cet opérateur, on considère que c'est la porté globale. Voici un exemple d'utilisation :

```
void fct(long a);       // une fonction globale
void Foo::fct(long a)   // une méthode la classe Foo
{
 fct(a);                // Foo::fct() s'appelle récursivement
 ::fct(a) ;             // appelle la fonction globale fct()
}
```

6.2.3 L'encapsulation

- ✓ L'encapsulation est un puissant concept qui permet de maîtriser la visibilité qu'on a des membres d'une classe vu de l'extérieur de la classe.
- ✓ L'utilisateur d'une classe, (c'est à dire un de ceux qui écrit du code qui instancie des objets de la classe), n'a donc accès qu'à un nombre restreint d'attributs et de méthodes. Cela réduit d'autant la complexité d'utilisation des objets.
- ✓ Chaque attribut et méthode a un niveaux de visibilité parmi :
    - o Le niveau de visibilité **publique** : L'attribut (resp. méthode) peut être lu et écrit (resp. appelée) partout dans le code, à partir d'une instance de la classe.
    - o Le niveau de visibilité **privé** : L'attribut (resp. méthode) peut être lu et écrit (resp. appelée) seulement dans le code des méthode de la classe.

- Le niveau de visibilité **protégé** : L'attribut (resp. méthode) peut être lu et écrit (resp. appelée) seulement dans le code des méthode de la classe et des méthodes des classes dérivées. (voir 6.5 pour la notion de classes dérivées).

✓ Voici un exemple pour exposer la syntaxe avec les mot clés **`public`**, **`protected`** et **`private`** :

```
class UneClasse
{
public:
 long UnAttributPublique;
 long UneMethodePublique(long a);
protected:
 char UnAttributProtege;
 short UneMethodeProtegee(char * p);
private:
 char* UnAttributPrive;
 short UneMethodePrivee(double d);
};
```

✓ Notez que par défaut les membres d'une classe déclarée avec le mot clé `class` sont privés. En effet, on peut déclarer une classe avec le mot clé `struct`. Dans ce cas les membres sont par défaut publiques. Ceci montre qu'en C++ la notion de structure et de classe est identique.

✓ On peut aussi utiliser le mot clé `union` pour définir une classe. Les attributs sont, comme en C, au même emplacement mémoire. Cependant l'utilisation des classes union est rare, et admet des contraintes qu'on ne détaillera pas. (voir [Stroustrup]).

6.2.4 Accès aux membres d'une classe

✓ Comme en C avec les champs d'une structure, on utilise l'opérateur `'.'` ou `'->'` selon que l'on a une instance ou un pointeur sur une instance. Dans les méthodes de la classes on accède aux membres tel quel. Un exemple :

```
struct CFoo
{
 long m_Attribut;
 long UneMethode (long a)
{
  m_Attribut = a;// accès à l'intérieur d'une méthode de la classe
       UneAutreMethode();
 }
 void UneAutreMethode();
};

void main()
{
   CFoo UnObjet;
   CFoo * pUnObjet = &UnObjet ;
   UnObjet.m_Attribut = 3;
   pUnObjet->m_Attribut = 4;
   UnObjet.UneMethode(3);
   pUnObjet->UneMethode(4);
}
```

6.2.5 Le pointeur this

- ✓ Dans toutes les méthodes (non statiques voir 6.2.11) de toutes les classes, C++ défini automatiquement un pointeur nommé **`this`**. C'est un pointeur, du type de la classe, qui pointe vers l'objet courant.
- ✓ On s'en sert surtout pour communiquer un accès à l'objet courant, à une entité extérieure à la classe.

```
class CFoo; // déclaration de la classe
void fct( CFoo * pFoo ); // une fonction globale
struct CFoo
{
 long m_Attribut;
 long UneMethode (long a)
 {
   this->m_Attribut = a;// équivalent à : 'm_Attribut = a;'
         fct(this); // communique l'adresse de l'objet courant à
                    // l'extérieur de la classe
 }
};
```

6.2.6 La surcharge des méthodes et les arguments par défaut

- ✓ C++ autorise la surcharge de méthodes, et les arguments par défaut des méthodes. Les règles sont exactement les mêmes que pour la surcharge de fonctions (voir 5.2) et les arguments par défaut de fonction (voir 5.3). Un exemple :

```
class CFoo
{
private:
 long m_Attribut;
public:
 long UneMethode(long a){ m_Attribut = a;}
 long UneMethode(double d){ m_Attribut = (long) d;}
};
```

6.2.7 Les constructeurs

- ✓ Comme nous l'avons vu, une méthode est automatiquement appelée lorsque l'objet est construit. On l'appelle le constructeur.
- ✓ En C++, une méthode constructeur porte le nom de la classe, et ne retourne rien (même pas le type `void`).
- ✓ Il peut y avoir :
  - o Pas de constructeur : Dans ce cas le compilateur fournit automatiquement un constructeur par défaut, qui n'accepte pas de paramètre.
  - o Un seul constructeur : Dans ce cas ce sera toujours lui qui sera appelé.
  - o Plusieurs constructeur : Dans ce cas ils diffèrent selon leurs signatures, le constructeur est surchargé.
- ✓ Voici un exemple avec 2 constructeurs :

```
class CFoo
{
private:
 long m_Attribut;
public:
 CFoo(long a){ m_Attribut = a;}
 CFoo(double d){ m_Attribut = (long) d;}
```

```
};
...
      CFoo UnObj;//erreur de compilation, pas de constructeur disponible
      CFoo UnObj(56); // constructeur qui accepte un long appelé
      CFoo UnObj(56.87); // constructeur qui accepte un double appelé
```

- ✓ Il existe une syntaxe pour initialiser les attributs dans un constructeur. Voici un exemple avec 3 constructeurs (Notez que c'est comme cela que l'on initialise un attribut constant) :

```
      class CFoo
      {
      private:
       const long   m_Attribut;  // un attribut constant
       char * m_pChaine;
      public:
       CFoo(long a,char* pChaine): m_Attribut(a),m_pChaine(pChaine) {}
 CFoo(long a): m_Attribut(a),m_pChaine(0) {}
 CFoo(char* pChaine): m_Attribut(0),m_pChaine(pChaine) {}
      };
```

- ✓ Les rôles principaux du constructeur sont :
  - o Initialisation des attributs.
  - o Allocation des ressources (zone mémoire, connexions, attributs dynamiquement alloués…).

6.2.8 Le Destructeur

- ✓ Comme nous l'avons vu, une méthode est automatiquement appelée lorsque l'objet est détruit. On l'appelle le destructeur.
- ✓ En C++, une méthode destructeur porte le nom de la classe, précédé du caractère `'~'`, ne retourne rien (même pas le type `void`), et n'accepte aucun argument.
- ✓ Il peut y avoir :
  - o Pas de destructeur : Dans ce cas le compilateur fournit automatiquement un destructeur par défaut.
  - o Un destructeur : Dans ce cas, il prend la place du destructeur par défaut. En effet, une classe a toujours exactement un destructeur.
- ✓ Voici un exemple avec un attribut alloué dynamiquement:

```
      class CFoo
      {
      private:
       long * m_pAttribut;
      public:
       CFoo(long a);
 ~CFoo();
};
CFoo::CFoo(long a)
      {
       m_pAttribut  = (long*) malloc(sizeof(long));
       *m_pAttribut = a;
      }
CFoo::~CFoo()
{
 free( m_pAttribut );
      }
```

- ✓ Le rôle principal du destructeur est la désallocation des ressources allouée dynamiquement par l'objet (souvent dans le constructeur).

6.2.9 Constructeur de copie

- ✓ Il existe un constructeur spécial appelé constructeur de copie. Comme son nom l'indique il est appelé lorsqu'un objet est initialisé à partir d'un autre objet de même type. Il y a copie d'objet.
- ✓ Le C++ prévoit toujours un constructeur de copie par défaut pour toutes les classes. Son rôle est de copier simplement les attributs.
- ✓ Parfois on doit coder explicitement le constructeur de copie . C'est le constructeur qui admet pour seul paramètre '`const NomDeLaClasse &`' . Voici un exemple où l'on doit coder absolument le constructeur de copie:

```
      class CFoo
      {
      private:
       long * m_pAttribut;
      public:
       CFoo(long a);
 ~CFoo();
 CFoo( const CFoo & Source );
};
CFoo::CFoo(long a)
      {
       m_pAttribut  = (long*) malloc(sizeof(long));
       *m_pAttribut = a;
      }
CFoo::~CFoo()
{
 free( m_pAttribut );
      }
      CFoo::CFoo( const CFoo & Source ) // Constructeur de copie
      {
       m_pAttribut  = (long*) malloc(sizeof(long));
       *m_pAttribut = *Source.m_pAttribut;
      }
```

- ✓ Cet exemple montre bien qu'un constructeur de copie est requis si la classe admet des attributs alloués dynamiquement. Dans l'exemple précédent, si on avait utilisé le constructeur de copie par défaut, les objets copiés auraient partagé l'attribut dynamique avec l'objet source, ce qui n'est en général pas ce que l'on veut.
- ✓ Notez que ce n'est pas le constructeur de copie qui est appelé lors de l'affectation d'un objet (i.e a=b ;). C'est une méthode spéciale appelée opérateur d'affectation (voir 6.7.3).

6.2.10 Constructeur de transtypage

- ✓ Les constructeurs sont aussi utilisés dans les conversions de type dans lesquelles le type cible est la classe du constructeur.
- ✓ Par défaut, dans une classe utilisateur, aucun constructeur de transtypage n'est définit. C'est à l'utilisateur de le définir, avec pour argument du constructeur un objet du type source.
- ✓ On peut obliger que le transtypage soit explicite avec le mot clé `explicit`. Voici un exemple :

```
class CEntier
{
private:
  int val;
public:
  CEntier()               {val = 0;}
  //transtypage de long vers Centier (explicite)
  explicit CEntier(long e){val = e;}
  //transtypage de char* vers Centier (pas forcément explicite)
  CEntier(char * e)       {val = *e;}
};

int main()
{
  long  e1 = 9;
  char  e2 = 7;
  CEntier UnEntier;  // le constructeur sans argument est appelé
  UnEntier = e1;  // erreur de compilation: e1 doit être casté
  UnEntier = (CEntier) e1; // pas d'erreur
  UnEntier = &e2; // le constructeur de transtypage d'un char* vers
                  // CEntier est appelé implicitement
  return 0;
}
```

6.2.11 Membres statiques

- ✓ On a la possibilité de déclarer des attributs, des variables, et des méthodes statiques avec le mot clé `static`. Cela indique au compilateur que ces membres appartiennent à la classe et non aux objets instanciés.
- ✓ Un **attribut statique** doit être initialisé hors de la déclaration de la classe. Un tel attribut ressemble à une variable globale, à ceci prés que la règle des niveaux de visibilité joue :

```
class CFoo
{
private:
 static long a;
public:
 static long b;
void UneMéthode(){ a=2;b=3; }
};
long CFoo::a = 5; // initialisation de a
long CFoo::b = 6; // initialisation de b
void f()
{
 CFoo::a = 8 ;  // erreur de compilation : a est privé
 CFoo::b = 9 ;
}
```

- ✓ Voici un exemple qui montre les propriétés d'une **variable statique** dans une méthode :

```
class CFoo
{
public:
 long Increment()
 {
  static long a=0; // a vaut 0 SEULEMENT lors du premier appel
```

```
                    // de la méthode Increment()
   a++;
   return a;
  }
 };

 int main()
 {
  CFoo foo1;
  CFoo foo2;
  long aa = foo1.Increment(); // aa vaut 1 après cette instruction
  aa = foo2.Increment(); // aa vaut 2 après cette instruction
  return 0;
 }
```

- ✓ Les **méthodes statiques** ne sont pas applicables sur un objet. Elles n'ont pas le pointeur `this` et par conséquent ne peuvent accéder qu'aux attributs statiques de la classe. Notez les deux façons d'appeler une fonction statique :

```
class CFoo
{
public:
 static long a;
 long b;
 static void Increment()
 {
  a++;
  b++;// erreur de compilation :
      // ne peut accéder à un attribut non-statique
 }
};
long CFoo::a = 0;
void f()
{
 CFoo Obj;
 CFoo::Increment();// appel à la fonction statique
 Obj.Increment() ; // un autre appel à la fonction statique
}
```

6.2.12 Les méthodes constantes

- ✓ C++ offre la possibilité de définir des méthodes constantes, c'est à dire que le compilateur garantit que leurs appels ne modifient aucun attribut de l'objet.
- ✓ Il suffit de rajouter le mot clé `const` à la fin de la méthode.
- ✓ Le compilateur n'autorise que l'appel de méthodes constantes sur un objet constant.

```
class CFoo
{
  public:
    long m_Attribut;
    void MaMéthode() const
    {
      m_Attribut++;  // erreur de compilation :
                     // la méthode est constante
    }
}
```

6.2.13 Pointeur sur les membres d'une classe

- ✓ Les attributs et les méthodes d'une classe peuvent être vus comme des variables et des fonctions en C++. A ce titre on peut définir des pointeurs sur les membres d'une classe.
- ✓ On distingue le cas des membres statiques des membres non-statiques.
- ✓ Pour les membres non statiques la syntaxe est du type :
  `définition classe ::* pointeur`
- ✓ Naturellement, pour accéder au membres pointés il faut un objet. Voici un exemple :

```
struct CFoo
{
   int m_Entier;
   long fct(int a){ return (long) ++a;}
};
typedef int  CFoo:: * TpCFooInt;
typedef long (CFoo::*TpCFooFct)(int);
void main()
{
  CFoo Obj;
  TpCFooInt pAttr = &CFoo::m_Entier;
  TpCFooFct pFct  = &CFoo::fct ;

  Obj.*pAttr = 9;
  long r = (Obj.*pFct)(6);  // les parenthèses sont requises!!
  // équivalent à  (&Obj)->*pAttr = 9;
  //               long r = (&Obj->*pFct)(6);
}
```

- ✓ Pour les membres statiques c'est plus simple puisqu'ils n'appartiennent pas à un objet mais à la classe entière. Voici un exemple :

```
struct CFoo
{
   static int m_Entier;
   static long fct(int a){ return (long) ++a;}
};
int CFoo::m_Entier = 0;

typedef long (*TpCFooFct)(int);  // pointeur sur fct normal
void main()
{
  int * pAttr = &CFoo::m_Entier;
  TpCFooFct pFct  = &CFoo::fct ;
  *pAttr = 9;  // variable et fct pointées, appels normaux
  long r = (*pFct)(6); // les parenthèses sont requises!!
}
```

6.3 Création dynamique des objets : Opérateurs new et delete

- ✓ Le langage C++ introduit deux nouveaux opérateurs de gestion mémoire : **`new`** et **`delete`**. Le premier effectue l'allocation, le second réalise la libération. L'avantage de l'utilisation de ces opérateurs par rapport à **`malloc()`** et **`free()`** vu au 3.18 est multiple.
- ✓ Ces opérateurs ont été crée pour la création et la destruction des objets, mais on peut les utiliser avec tout autres type. Voici un exemple :

```
class CFoo
{
public:
 long * pVal;
 CFoo(long arg){ pVal = new long(arg); }
 ~CFoo()       {delete pVal; pVal =0;}
};
...
CFoo * pObj = new CFoo(56);
...
// travail avec pObj
...
delete pObj;
pObj = 0; // NE JAMAIS OUBLIER DE METTRE A 0 LES PTR NON VALIDES
```

- ✓ Notez que le constructeur est appelé automatiquement après l'allocation de l'objet.
- ✓ Notez que le destructeur est appelé automatiquement avant la désallocation de l'objet
- ✓ Enfin tout objet alloué avec l'opérateur `new` doit être desalloué avec l'opérateur `delete` de même que une zone mémoire allouée avec la fonction `malloc()` doit être désallouée avec la fonction `free()`.
- ✓ En C++ , les problèmes de fuite de mémoire décris en 3.18 surviennent aussi avec les objets alloués dynamiquement avec `new`. Sans un bon design, un programme C++ qui alloue dynamiquement des objets devient vite une usine à gaz qui gonfle en mémoire !

6.4 Les tableaux d'objets

6.4.1 Introduction

- ✓ De la même manière que C peut définir des tableaux de variables de même type, C++ autorise l'utilisation de tableaux d'objets de même type.
- ✓ Il faut être conscient que les constructeurs sont appelés dans l'ordre croissant des adresses, et les destructeurs dans l'ordre décroissant des adresses.
- ✓ Il faut un constructeur sans paramètres. C'est en effet lui qui est appelé par défaut. Il reste cependant possible de transmettre des valeurs à des constructeurs ayant des paramètres, par le biais d'un initialiseur.

```
class CFoo
{
  long m_Val;
public:
  CFoo (long Val = 0): m_Val(Val) {}
  ~CFoo (){}
};
void f()
{
  CFoo Tab[3] = { 5, 6 } ;
  // Tab[0].m_Val = 5 ; Tab[1].m_Val = 6 ; Tab[2].m_Val = 0
  ...
} // Détruit les 3 CFoo avec 3 appels au destructeur
```

6.4.2 Création dynamique des tableaux d'objets : new [ ] et delete [ ]

- ✓ Contrairement au C, on peut aussi créer dynamiquement des tableaux d'objets. La taille du tableau peut être spécifiée dynamiquement.
- ✓ Ceci est possible grâce à l'opérateur **`new[]`**. De même les tableaux dynamiques sont désalloués avec l'opérateur **`delete[]`**. Ces opérateurs appellent les constructeurs et destructeurs de la même façon que lors de la création des tableaux d'objets non dynamiques.

```
class CFoo
{
  long m_Val;
public:
  CFoo (long Val = 0): m_Val(Val) {}
  ~CFoo (){}
};
void f()
{
  int i = 5;
  CFoo * pTab = new CFoo[i] ;
  ...
} // Détruit les 5 CFoo avec 5 appels au destructeur
```

6.5 Héritage et dérivation

6.5.1 Objectif : Réutilisation de code

- ✓ Une lois fondamentale en programmation est que **la complexité dépend linéairement du nombre d'instruction**. Un programme deux fois plus gros qu'un autre est donc deux fois plus long à écrire, tester et maintenir.
- ✓ Le mécanisme de dérivation ou d'héritage, est un des points clés de la programmation objet. Il est la réponse au problème de la réutilisation de code.
- ✓ Concrètement, on part du constat que dans un programme, différentes classes ont des fonctionnalités identiques.
    - o Dans un programme de gestion de personnel d'une entreprise, il peut y avoir une classe pour les secrétaires, une classe pour les techniciens, une classe pour les cadres… Toutes ces classes ont ceci de commun qu'une instance représente un employé, avec les attributs `nom`, `age`, `adresse`, `salaire`… et les méthodes `Evalue( )`, `ModifieLesHoraires( )`, `Augmente( )`…
    - o Dans un programme de dessin, il peut y avoir une classe pour les cercles, une classe pour les rectangles, une classe pour les triangles… Toutes ces classes ont ceci de commun qu'une instance a les attributs `CouleurDuTrait`, `TailleDuTrait`… et les méthodes `Dessine()`, `Translate( )` , `Rotate( )` , `Grossi( )`…
    - o Dans un programme qui communique avec différents protocoles de communication, chaque point de communication a, indépendamment du protocole sous-jacent, les attributs `NbOctetsEnvoyés/reçu`, `DateDeCréation`… et les méthodes, `PingLePointDistant( )`, `EnvoieUnStream( )`, `DeconnecteEtFerme()`…
- ✓ L'idée de la réutilisation est de cerner ces similitudes, et de les encapsuler dans une classe appelée **classe de base** (par exemple `CEmployé`, `CFigureGéométrique` et `CPointDeCommunication`).
- ✓ Une classe dérivée dérive d'une classe de base, elle héritera des membres de la classe de base. Concrétement si `CTechnicien` hérite de `CEmploye`, `CTechnicien` a les attributs `Nom`, `Age`… et les méthodes `Evalue()`, `ModifieLesHoraires()`…
- ✓ Une difficulté du design objet est donc de bien cerner les similitudes (appelés aussi *communalités*) entre classes. L'autre difficulté étant de définir la façon dont les objets interagissent entre eux.
- ✓ Voici quelque phrases à se dire pour confirmer l'intuition.
    - o Un Technicien **est un** employé, un Rectangle **est une** figure géométrique, une socket **est un** point de communication…
    - o Un Technicien **a un** nom, un Rectangle **a une** couleur de trait, une socket **a une** date de création…
    - o Un Technicien **peut être** évalué, un Rectangle **peut être** translaté, une socket **peut être** utilisée pour envoyer un stream…

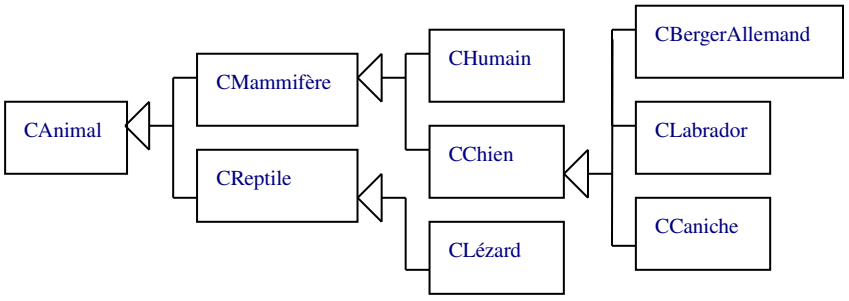
6.5.2 Héritage simple et membres protégés

✓ C++ autorise à une classe de dériver d'une autre classe. Voici un exemple qui montre aussi l'utilisation du mot clé `protected` (défini au 6.2.3).

```
class CEmploye
{
 private :
   long PrévisionDaugmentation ;
 protected :
   long Salaire;
 public :
   char * Nom;
   long Age ;
   void Augmente() { Salaire += PrévisionDaugmentation ; }
};

class CTechnicien : CEmploye  // CTechnicien hérite de CEmploye
{
 public :
   long CalculeDuCoutDuneUneMission( CMission & Mission )
   {
     PrévisionDaugmentation += 10000 ; // erreur de compilation :
                 // l'attribut PrévisionDaugmentation est
                 // privé et ne peut être accédé de CTechnicien
     return Salaire/169 * Mission.NbHeure ; // OK l'attribut
                 // Salaire est protégé est peut donc être accédé
                 // par CTechnicien
   }
}
...
CTechnicien Roger;
Roger.Age += 32 ;  // OK
Roger.Salaire += 10000 ; // erreur de compilation :
// on ne peut accéder aux membres protégé à l'extérieur d'une
// classe et de ses dérivées
```

✓ Notez qu'une classe de base peut être aussi une classe dérivée. Tout ceci peut être vu dans un schéma de dérivation. Par exemple :



✓ La flèche signifie `(la classe pointeur) hérite de (la classe pointée).`
✓ Il existe des normalisation pour les schémas objets, la plus courantes étant la normalisation UML (Unified Modelling Langage) voir **[UML]**.

6.5.3 Contrôle d'accès aux classes de base

- ✓ On peut affiner le contrôle d'accès qu'à une classe dérivée sur les membres d'une classe de base.
- ✓ En effet lors de l'héritage d'une classe de base on peut préciser un des 3 mots clés pour préciser le contrôle d'accès : `public, protected, private`, par exemple : `class CTechnicien : protected CEmploye`
- ✓ Par défaut, si aucun contrôle d'accès n'est précisé, C++ utilise le contrôle d'accès `public` pour les classes déclarées avec le mot clé `struct` et `private` pour les classes déclarée avec le mot clé `class`.
- ✓ Le contrôle d'accès agit différemment sur les membres de la classe de base selon qu'ils sont `public, protected, private.`
- ✓ Résumons ceci dans un tableau :

Niveau de visibilité des attributs et méthodes de la classe de base dans la classe dérivée		Contrôle d'accès de la classes de base		
		Public	protected	private
Niveau de visibilité des attributs et méthodes dans la classe de base	public	Public	protected	private
	protected	Protected	protected	private
	private	Accès interdit	Accès interdit	Accès interdit

6.5.4 Méthodes virtuelles et polymorphisme

- ✓ En programmation objet on est souvent confronté au problème suivant : On crée des objets, instances de plusieurs classes dérivés d'une classe de base, puis on veut leur appliquer un traitement de base, c'est à dire défini dans la classe de base.
- ✓ Concrètement :
  - o On veut obtenir une description de tous les employés. (traitement de base : obtenir une description d'un employé, quelque soit sa catégorie).
  - o On veut dessiner toutes les figures géométriques (traitement de base : dessiner une figure géométrique, quelque soit le type de figure).
  - o On veut déconnecter puis fermer tous les points de communication (traitement de base : déconnecter puis fermer un point de communication quelque soit le protocole de communication sous jacent).
- ✓ Il est très utile de rassembler les objets qui doivent subir le traitement de base, puis de leur faire subir le traitement de base dans une boucle. Toute l'élégance de cette méthode vient du fait que **le traitement de base s'applique sur un objet sans savoir exactement sa classe, on ne connaît que sa classe de base : c'est le polymorphisme**.
- ✓ En C++ ce qu'on a appelé traitement de base, s'appelle **méthode virtuelle**.
- ✓ Un classe de base qui admet au moins une méthode virtuelle est une **classe polymorphe**.
- ✓ Voici un exemple ou l'on affiche une description pour chaque employés :

```
#include <iostream.h>
class CEmploye
{
 public :
   char * m_pNom;
   CEmploye(char * pNom): m_pNom(pNom) {}
   virtual void GetDescription(){
         cout << "Nom: " << m_pNom << "  ";}
};
//-----------------------------------------------------
```

```
class CTechnicien : CEmploye
{
 public:
      CTechnicien(char * pNom):CEmploye(pNom) {}
      virtual void GetDescription(){
         CEmploye::GetDescription();
         cout << "Fonction: Technicien";}
};
//-------------------------------------------------------
class CSecretaire : CEmploye
{
 public:
      CSecretaire(char * pNom):CEmploye(pNom) {}
      virtual void GetDescription(){
          CEmploye::GetDescription();
          cout << "Fonction: Secretaire";}
};
//-------------------------------------------------------
int main()
{
      CEmploye * pTab[3] = {0,0,0};
      pTab[0] = (CEmploye*)new CTechnicien("B.Durant");
      pTab[1] = (CEmploye*)new CSecretaire("W.Dupont");
      pTab[2] = (CEmploye*)new CTechnicien("E.Martin");
      for( int i = 0 ; i<3; i++)
            pTab[i]->GetDescription();
      for( i = 0 ; i<3; i++) {delete pTab[i];pTab[i]=0;}
  return 0;
}
```

- ✓ Voici la sortie du programme :

```
Nom: B.Durant  Fonction: Technicien
Nom: W.Dupont  Fonction: Secretaire
Nom: E.Martin  Fonction: Technicien
```

- ✓ La fonction `GetDescription()` est virtuelle car elle est déclarée avec le mot clé **`virtual`**.
- ✓ `GetDescription()`, est toujours appelée sur un pointeur de type `CEmploye`. Pourtant on voit bien que le programme passe par les fonctions `GetDescription()` de `CTechnicien` et `CSecretaire`. C'est la magie du polymorphisme. Il évite un test de type fastidieux et participe à la maintenance.
- ✓ En effet lors de l'ajout de la classe `CCadre` il faudrait maintenir le test, alors qu'avec le polymorphisme il n'y a absolument rien à faire pour que la fonction `CCadre::GetDescription()` soit appelée.
- ✓ Notez l'emploi de l'opérateur de résolution de portée, dans les classes dérivées, pour appeler `CEmploye::GetDescription()`.
- ✓ Notez la façon utilisée par les constructeurs des classes dérivées `CTechnicien` et `CSercretaire` pour appeler le constructeur de la classe de base `CEmploye`.
- ✓ Notez que lorsqu'une fonction a été déclarée comme virtuelle à un niveau d'arborescence de dérivation, il n'y a plus besoin de mettre le mot clé `virtual` dans les classe dérivées.
- ✓ Notez que cet exemple utilise les flots d'entrée / sortie de la STL. (`cout <<` ...)

6.5.5 Méthodes virtuelles pures, classes abstraites et interface

- ✓ Dans l'exemple précédent, afficher la description d'un employé (i.e son nom) a un sens. Il est donc utile de mettre du code dans la méthode virtuelle `GetDescription()` de la classe de base `CEmployee`.
- ✓ Il arrive que la sémantique de la classe de base soit si abstraite, qu'on n'ai pas de code à mettre dans la méthode virtuelle.
- ✓ Par exemple pour la classe `CFigureGeometrique` il n'y a rien à mettre dans la méthode virtuelle `Dessine()` puisqu'on ne sait pas quelle type de figure géométrique on instancie.
- ✓ De même pour la classe `CPointDeCommunication` il n'y a rien à mettre dans la méthode `DeconnecteEtFerme()` puisqu'on ne connaît pas le protocole de communication sous jacent.
- ✓ On pourrait très bien déclarer une fonction virtuelle sans code à l'intérieur, mais C++ permet de signifier qu'il n'y a pas de code à l'intérieur de ces méthodes que l'on appelle **méthodes virtuelles pures**.
- ✓ Si une classe admet **au moins une** méthode virtuelle pure on dit que c'est une **classe abstraite**. Ce qui veut dire qu'il ne peut y avoir d'objet de cette classe (puisque certaines méthodes ne sont pas codées).
- ✓ En revanche il peut y avoir des classes dérivées qui implémentent **toutes** les méthodes virtuelles pures. Elles ne sont donc plus abstraites et peuvent être instanciées.
- ✓ De plus on peut bénéficier du polymorphisme avec des pointeurs de classes abstraites. Voici un exemple :

```
class CFigureGeometrique
{
 public :
   virtual void Dessine()=0; // =0 signifie méthode virtuelle pure
};//-------------------------------------------------------
class CCercle : CFigureGeometrique
{
 public:
      CPoint m_Centre;
      double m_Rayon;
      Ccercle( CPoint Centre, double Rayon) :
            m_Centre(Centre), m_Rayon(Rayon) { }
      void Dessine (){
      // dessine un cercle à partir de son centre et de son rayon
      }
};//-------------------------------------------------------
class CRectangle : CFigureGeometrique
{
 public:
      CPoint m_Sommet1;
      CPoint m_Sommet2;
      Cpoint m_Sommet3;
      CRectangle( CPoint Sommet1, CPoint Sommet2, CPoint Sommet3):
      m_Sommet1(Sommet1),m_Sommet2(Sommet2),m_Sommet3(Sommet3) {}
      void Dessine (){
      // dessine un rectangle à partir de 3 de ses sommets
      }
};//-------------------------------------------------------
```

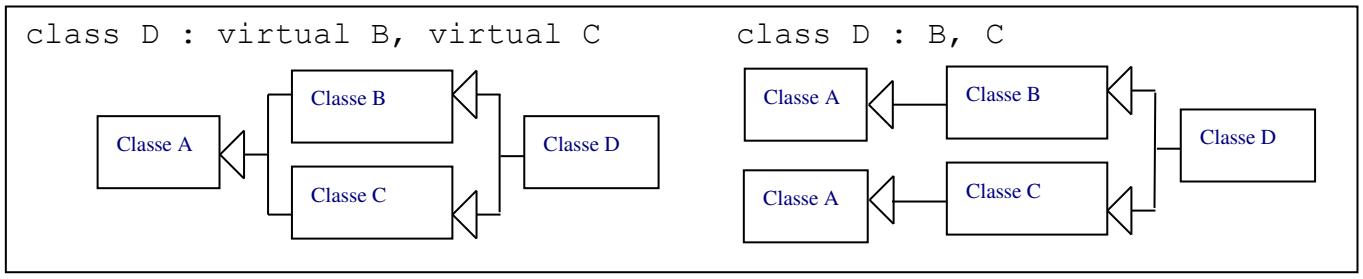
```
int main()
{
      CFigureGeometrique * pTab[3] = {0,0,0};
      pTab[0] = (CFigureGeometrique *)new CCercle((0,0),3.2);
      pTab[1] = (CFigureGeometrique *)new
                                      CRectangle((0,0),(0,2),(1,2));
      pTab[2] = (CFigureGeometrique *)new CCercle((1,1),4.1);
      for( int i = 0 ; i<3; i++)
            pTab[i]->Dessine();
      for( i = 0 ; i<3; i++) {delete pTab[i];pTab[i]=0;}

      CFigureGeometrique UneFigure; // erreur de compilation
      // ne peut instancier une classe abstraite !
      CFigureGeometrique* pUneFigure = new CFigureGeometrique();
      // erreur de compilation :
      // ne peut instancier une classe abstraite !
      return 0;
}
```

- ✓ Le polymorphisme est encore plus évident avec une classe abstraite puisqu'on est sûr que ce ne peut être la méthode virtuelle pure qui est appelée puisqu'elle n'existe pas !
- ✓ Une classe qui n'a que des méthodes virtuelles pures et aucun attribut est appelée une **interface**. Ce concept est très puissant et très utilisé. Il est à la base de l'architecture Microsoft COM, sur laquelle repose tout les systèmes type NT/XP.

6.5.6 Héritage multiple

- ✓ C++ autorise à une classe d'avoir plusieurs classes de base. Par exemple :
  `CFigureGeometrique : CZoom, CDeplace, Csave, CLoad`
- ✓ Bien que puissante, cette possibilités soulèvent beaucoup de problèmes, qu'il vaut mieux maîtriser avant de l'utiliser. D'ailleurs certains langages POO comme Java ou C# ne permettent pas l'héritage multiple.
- ✓ Un gros problème est si les classes de bases héritées ont une même classe de base :



- ✓ Une instance de D contient t-elle une ou deux instances de A ? En fait vous avez le choix. Si vous ne voulez qu'une instance de A (figure de gauche) il faut **hériter virtuellement** des classes B et C.
- ✓ Un autre problème est l'ordre d'appel des constructeurs et destructeurs des classe de base. Pour les constructeurs c'est l'ordre définit lors de la spécification de la classe. Pour les destructeurs c'est l'ordre inverse.
- ✓ Un autre problème est si dans la hiérarchie il y a des attributs ou des méthodes homonymes. On résout ce problème en précisant la classe à utiliser avec l'opérateur de résolution de portée.
- ✓ Un autre problème dans le cas de l'héritage virtuel est si les classes B et D initialisent A de 2 façons différente. En fait C++ autorise la classe D d'initialiser A, bien que D ne soit pas dérivée directement de A.

- ✓ Une utilisation intéressante de l'héritage multiple est d'hériter de plusieurs interfaces. En effet, tous les problèmes exposés n'ont pas lieu d'être avec les interfaces puisqu'elles n'ont pas de constructeur avec paramètres, ni attributs, ni classes de base.

- ✓ A part cette utilisation particulière nous déconseillons l'héritage multiple, qui produit du code illisible et difficile à maintenir. Cependant pour un exposé complet voir **[Stroustrup]**.

6.5.7 Destructeurs virtuels

- ✓ Une règle du C++ est :

LE DESTRUCTEUR D'UNE CLASSE DE BASE DOIT ETRE VIRTUEL DU MOMENT QUE LA CLASSE DE BASE EST POLYMORPHE (I.E A AU MOINS UNE METHODE VIRTUELLE)

- ✓ Dans le cas contraire, le destructeur d'une classe dérivée n'est pas appelé par polymorphisme, avec tous les problèmes de déallocation que cela comporte.

6.6 Fonctions méthodes et classes amies d'une classe

- ✓ En C++ on peut violer le mécanisme d'encapsulation définie par les niveaux de visibilité des membres.
- ✓ Concrètement, une classe peux décider de donner un accès publique à tous ces membres à :
  - o Une fonction particulière.
  - o Une méthode d'une classe particulière.
  - o Toute les méthodes d'une classe particulière.
- ✓ Ceci reste dangereux, et est à utiliser avec précaution. En général une classe rend amie une entité qui l'utilise entièrement. La classe et l'entité sont fortement couplées. Par exemple une fonction qui affiche les attributs de la classe, doit être amie de la classe.
- ✓ Voici un exemple pour la syntaxe :

```
class CAmie;
class CHote;
class CSemiAmie
{
   char * AfficheHote( Chote & hote );
   ...
};
class CHote
{
   friend char * CSemiAmie::AfficheHote( CHote &);
   friend char * FctAmie(CHote &);
   friend class CAmie;
   ...
};
char * FctAmie( CHote & hote );
```

- ✓ Notez que la classe et ses entités amies doivent avoir une connaissance mutuelle.
- ✓ Notez que l'amitié entre classe n'est pas réflexive. A peut être amie de B sans que B soit amie de A.
- ✓ Notez que l'amitié n'est pas transitive. A amie de B et B amie de C n'implique pas A amie de C.
- ✓ Notez que l'amitié a une classe de base n'a aucune influence implicite sur les classes dérivées.

6.7 La surcharge des opérateurs (operator overloading)

6.7.1 Introduction

- ✓ Le langage C++ étend la notion de surcharge aux opérateurs du langage.
- ✓ Concrètement on peut par exemple définir une classe `CFraction` ou `CComplexe`, et surcharger les opérateurs `+,-,*,\` afin d'étendre ces opérations au type concerné.
- ✓ Par convention, un opérateur `op` reliant deux opérandes `a` et `b` est vu comme une fonction prenant en argument un objet de type `a` et un objet de type `b`.
- ✓ Ainsi, l'écriture `a op b` est traduite en langage C++ par un appel `op(a,b)`.
- ✓ Par convention également, un opérateur renvoie un objet du type de ses opérandes.
- ✓ Un opérateur est représenté par un nom composé du mot operator suivi de l'opérateur (`operator+()  operator/()`…).
- ✓ En C++ les opérateurs suivants peuvent être surchargés :

```
+       -       *       /       &       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ->*     ,
->      []      ()      new     new[]   delete  delete[]
```

- ✓ Les opérateurs suivant ne peuvent être surchargés :

```
::      .       .*      ?:
```

- ✓ Lorsqu'on surcharge les opérateurs il est très important qu'ils aient une signification très proche de leur signification habituelle (ne pas utiliser l'opérateur + pour multiplier 2 complexes !!)
- ✓ De plus il est essentiel d'être cohérent : '`a += b ;`' et '`a = a + b`' doivent donner la même valeur à `a`.

6.7.2 Surcharge interne et externe des opérateurs

- ✓ C++ autorise deux méthodes pour surcharger un opérateur :
  - o La surcharge **interne** : l'opérateur est une **méthode** de la classe.
  - o La surcharge **externe** : l'opérateur est une **fonction amie** de la classe.
- ✓ Un exemple de surcharge interne :

```
class CComplexe
{
 private:
   double m_dRe,m_dIm; // partie réelle et imaginaire
 public:
   CComplexe(double dRe = 0.0,double dIm = 0.0):
                                    m_dRe(dRe),m_dIm(dIm){}
   CComplexe operator+ (const CComplexe comp){
      return CComplexe( m_dRe + comp.m_dRe, m_dIm + comp.m_dIm);
   }
};
...
CComplexe a(1,2),b(2,3),c;
c = a+b;  // operator+() est appelée sur la première opérande, a
```

- ✓ Même exemple avec une surcharge externe :

```
class CComplexe
{
 friend CComplexe operator+ (const CComplexe a, const CComplexe b);
 private:
   double m_dRe,m_dIm; // partie réelle et imaginaire
 public:
   CComplexe(double dRe = 0.0,double dIm = 0.0):
                                    m_dRe(dRe),m_dIm(dIm){}
};
CComplexe operator+ (const CComplexe a, const CComplexe b){
      return CComplexe( a.m_dRe + b.m_dRe, a.m_dIm + b.m_dIm);
   }


...
CComplexe a(1,2),b(2,3),c;
c = a+b;
```

- ✓ La librairie Standard C++ (STL) définit le type `complex`, il ne faut donc pas utiliser l'exemple.
- ✓ Notez que les opérandes sont passées sous forme de références constantes.
- ✓ Notez qu'un opérateur interne est toujours appelé sur la première opérande, ce qui confère un avantage aux opérateurs externes, qui eux sont symétriques. En effet, C++ peut appliquer les règles de transtypage sur les deux opérandes.
- ✓ Savoir utiliser la surcharge interne ou externe, dépend du contexte et de l'opérateur. Par exemple pour additionner un `double` et un `CComplexe` il vaut mieux définir l'opérateur en externe puisqu'on, ne peut toucher au type `double`.
- ✓ On présente par la suite quelques opérateur à utiliser plus subtilement. Néanmoins cette présentation est assez brève pour un sujet assez vaste. Pour plus de détails consultez **[Stroustrup]** .

6.7.3 L'opérateur d'affectation (operator=)

- ✓ Il existe **la règle de forme canonique** en C++, **lorsqu'on a besoin d'au moins l'une de ces méthode :**
  - **Le constructeur de copie**
  - **Le destructeur**
  - **L'opérateur d'affectation**

  Il est important de coder les trois.
- ✓ En effet, si l'on a besoin d'une de ces méthodes, cela veut dire qu'un des attributs de la classe est dynamique, auquel cas il faut coder les trois méthodes et ne pas compter sur les méthodes par défaut de C++.
- ✓ Il y a trois règles à respecter lorsqu'on code un opérateur d'affectation :
  - Prévoir le cas ou on s'auto-affecte.
  - Retourner une copie de l'objet courant (`return *this`) afin de pouvoir enchaîner les opérateurs d'affectation.
  - Il ne faut pas utiliser le constructeur de copie dans l'opérateur d'affectation et vice-versa. Par contre il est utile de coder une fonction `Copie()` appelée par le constructeur de copie et l'opérateur d'affectation.
- ✓ Voici un exemple d'application de ces règles :

```
CComplexe &  CComplexe::operator=( const Complexe & Source )
{
   if( &Source != this )  // cas d'auto affectation
   {
      m_dRe = Source.m_dRe ;
      m_dIm = Source.m_dIm ;
   }
   return *this; // retour de l'objet courant
}
```

6.7.4 L'opérateur de transtypage

- ✓ On a vu qu'on peut réaliser des constructeurs pour faire le transtypage (explicite ou non) (voir 6.2.10) d'une classe source vers une classe cible. Il est nécessaire que ce constructeur soit dans la classe cible.
- ✓ Il se peut que le code de la classe cible ne soit pas modifiable. Dans ce cas, on peut définir l'opérateur de transtypage dans la classe source.
- ✓ L'opérateur de transtypage ne doit être utilisé que dans ce cas, car il requiert une copie de l'objet retourné vers l'objet cible en plus (donc moins performant).
- ✓ De plus si le constructeur de copie et l'opérateur de transtypage sont disponibles, le compilateur choisira le constructeur de copie, a moins que le constructeur de copie soit explicite et que le transtypage ne le soit pas. Ceci relève néanmoins d'un mauvais design.
- ✓ Voici un exemple oû on veut transtyper un `CComplexe` en un `const char *` (par exemple pour l'afficher). On ne peut pas faire autrement qu'utiliser un opérateur de transtypage puisqu'on ne peut ajouter un constructeur de transtypage au type `char*` :

```
CComplexe::operator char* const () const;
```

6.7.5 L'opérateur de comparaison

- ✓ Il ne faut retenir qu'une seule chose pour surcharger un opérateur de comparaison : Il faut retourner une variable de type `bool`.

```
bool CEntier::operator>( const Entier & ent) const;
bool CEntier::operator==( const Entier & ent) const;
```

6.7.6 Les opérateurs d'incrémentation et de décrémentation

- ✓ C++ fait la différence entre l'utilisation de ces opérateur en suffixe ou en préfixe.
- ✓ De plus ces opérateurs travaillent sur l'objet lui même.
- ✓ Il a donc fallut établir une convention pour pouvoir différencier l'utilisation de ces opérateurs en préfixe ou en suffixe, dont voici un exemple :

```
class CFlottant
{
 private:
   double m_d;
 public:
   CFlottant(double d = 0.0) :m_d( d ){}
   CFlottant operator++(int)  // Opérateur ++ suffixé
   {                          // on retourne la valeur puis on
     CFlottant tmp(m_d);      // l'incrémente.
     m_d += (double)1.0;      // Notez l'argument int à ne
     return tmp;              // pas utiliser
```

```
   }                          //
   CFlottant &operator++(void)// Opérateur ++ préfixé
   {                          // on incrémente puis on retourne
     m_d += (double)1.0;      // une référence sur l'objet
     return *this;            // lui même
   }                          //
};
```

6.7.7 L'opérateur fonctionnel

- ✓ L'opérateur fonctionnel `( )` peut être surchargé.
- ✓ Cela permet d'avoir des objets qui se comportent comme des fonctions.
- ✓ Cet opérateur est très utilisé dans la librairie standard.
- ✓ Cet opérateur est n-aire, c'est à dire qu'il peut prendre un nombre quelconque d'opérandes
- ✓ Voici un bref exemple qui montre une utilisation concrète de cet opérateur :

```
struct CMatrice
{
   double &operator()(unsigned short col, unsigned short lin);
   double operator()(unsigned short col, unsigned short lin) const;
   ...
};
...
CMatrice M;
M(3,4) = 3.14;    // 1er  opérateur fonctionnel appelé : modif
double d = M(1,3);// 2eme opérateur fonctionnel appelé : accès
```

- ✓ Les deux opérateurs fonctionnels définis permettent de modifier la valeur d'un élément de la matrice, ou d'y avoir accès.

6.7.8 Les opérateurs de déréférencement, d'indirection et de sélection

- ✓ Les opérateurs de déréférencement `'*'`, d'indirection `'&'` et de sélection `'->'` peuvent être surchargés. Cela permet d'encapsuler l'accès à un objet. Voici un exemple :

```
struct CEncapsulee
{
   int i;
};
CEncapsulee o;
struct CEncapsulante // cette classe encapsule la variable o
{
  CEncapsulee * operator->(void) const { return &o; }
  CEncapsulee * operator&(void)  const { return &o; }
  CEncapsulee & operator*(void)  const { return o; }
};
void main()
{
  CEncapsulante e;
  e->i = 2;    // o.i = 2
  (*e).i = 3;  // o.i = 3
  CEncapsulee * p = &e;
  p->i = 4;    // o.i = 4
}
```

6.7.9 Autres opérateurs

- ✓ Les paragraphes précédent montrent comment surcharger pratiquement tous les opérateurs mis à part les opérateurs d'allocation dynamique de mémoire `new` `delete` `new[]` et `delete []`.

- ✓ La surcharge de ces opérateurs est la plus difficile et dépasse le cadre de ce cours. Pour un exposé détaillé voir **[Stroustrup]** ou **[Casteyde]**.

7 Les mécanismes avancés de C++

7.1 Les espaces de nommage (namespace)

7.1.1 Introduction

- ✓ Un problème survient souvent dans les grosses architectures utilisant de nombreuses librairies et fonctions développées séparément :

  Il y a souvent collision entre les identifiants.
- ✓ Peut être par manque d'imagination, surtout par respect de conventions, les programmeurs ont tendance à donner des noms identiques à leurs classes et fonctions.
- ✓ Par exemple un logiciel de cartographie anglophone, définira la classe `map` comme un point d'accès à une carte alors que la librairie standard définie la classe `map` comme une classe d'association entre objets (c'est du vécu…).
- ✓ Heureusement les espaces de nommage résolvent totalement ce problème.

7.1.2 Espaces de nommage nommés

- ✓ Concrètement, on nomme une région déclarative (une portion du code) en utilisant le mot clé **`namespace`** :

```
#include <map> // ce fichier de la librairie standard définit
               // une classe 'map', à l'intérieur d'un espace de
               // nommage nommé std
namespace MesMaps
{
  class map
  {
    // moi aussi j'ai le droit de faire une classe map!!
    ...
    void Methode();
  };
  void fct(map m,double);
}

void MesMaps::fct(MesMap::map m, double d){...}
void MesMaps::map::Methode(){...}

MesMaps::map m1,m2;     // instancie ma classe
std::map<int,double> m; // instancie la classe map de la librairie
                        // standard
```

- ✓ Cette exemple montre bien comment cela se passe. A l'intérieur de notre espace de nommage `MesMaps` on donne les noms que l'on veut à nos classes et nos fonctions.
- ✓ A l'extérieur de l'espace de nommage on accède à nos classes et nos fonctions en les préfixant par `MesMaps::`.
- ✓ On peut imbriquer les espaces de noms. Si C dans B dans A et la fonction `fct()` est dans C on y accède avec `A::B::C::fct()`.
- ✓ On peut scinder un espace de nommage pourvu qu'on garde le même nom :

```
namespace A {int i ;}
namespace B {int j ;}
```

```
namespace A {int j ;}  // extension de l'espace de nommage A
```

7.1.3 La déclaration using

- ✓ La déclaration `using` permet l'utilisation d'un identificateur déclaré dans un espace de nommage sans avoir à utiliser l'opérateur de résolution de portée :

```
namespace A{  int i; int fct(int j);  }
using A::i;
using A::fct;
...
i=fct(67);   // utilise i et fct de l'espace de nommage A
```

- ✓ Bien évidemment cela peut générer des collisions d'identificateurs.
- ✓ On peut aussi utiliser la déclaration `using` dans un autre espace de nommage mais c'est à éviter :

```
namespace A{int i;}
namespace B{using A::i;}
B::i = 6;   //c'est bien le i déclaré dans A
```

7.1.4 La directive using namespace

- ✓ La directive `using` permet d'utiliser tous les identificateurs d'un espace de nommage.

```
namespace A{  int i; int fct(int j);  }
using namespace A;
...
i=fct(67);   // utilise i et fct de l'espace de nommage A
```

- ✓ Il semblerait que cette pratique enlève le bénéfice de l'utilisation des espaces de nommage, mais dans le cas où il n'y a pas de collisions d'identificateurs, c'est beaucoup moins fastidieux que d'utiliser la déclaration `using` pour tous les identificateurs.

7.1.5 Alias d'espace de nommage

- ✓ Lorsqu'un espace de nommage porte un nom très compliqué, il peut etre avantageux d'utiliser un alias de ce nom :

```
namespace nom_alias = nom_complique_a_renomer ;
```

- ✓ Encore une fois il faut faire attention à ne pas générer des collisions d'identificateurs.

7.1.6 Les espaces de nommage anonymes

- ✓ On peut aussi définir des espaces de nommage anonymes (sans nom). C'est comme si on déclarait des variables et des fonctions globales statiques.
- ✓ Concrètement elles ne pourront pas être utilisées dans un autre fichier, même avec le mot clé `extern` pour les variables.

7.2 Gestion des exceptions

7.2.1 La problématique :Gérer toutes les erreurs dans un programme

- ✓ Los des appels systèmes ou hors du programme courant, les programmes doivent faire face à des situations exceptionnelles indépendantes du programmeur :
  - o Accès à un fichier qui n'existe pas ou plus.
  - o Demande d'allocation de mémoire alors qu'il n'y en a plus.
  - o Accès à un serveur qui est crashé.
  - …
- ✓ Ces situations, qui ne sont pas des bugs mais que l'on peut appeler erreurs, engendrent un arrêt du programme si elles ne sont pas traitées.
- ✓ Pour les traiter on peut tester les codes d'erreurs retournés par les fonctions critiques, mais ceci présente deux inconvénients :
  - o Le code devient lourd, puisque chaque appel à une fonction critique est suivie de nombreux tests.
  - o Le programmeur doit prévoir toutes les situations possibles dès la conception du programme, ainsi que définir les réactions du programme et les traitements à effectuer
- ✓ En fait ces inconvénients sont majeurs, et il a fallu trouver d'autres solutions à ce problème : c'est la gestion des exceptions.

7.2.2 Principe de la gestion des exceptions

- ✓ Voici les étapes dans la gestion d'une exception :
  - o Un erreur est trouvée.
  - o On construit un objet qui contient, éventuellement, des paramètres descriptifs de l'erreur.
  - o Une exception est lancée, paramétrée par l'objet.
  - o Deux possibilités peuvent survenir à ce moment :
    - Un gestionnaire d'exception rattrape l'exception, l'analyse, et à la possibilité d'exécuter du code, par exemple pour sauver des données.
    - Il n'y a pas de gestionnaire d'exception prévu pour ce type d'exception. Le programme se termine.
- ✓ Voici un exemple :

```
struct CErrorAccesServeur
{
  unsigned long  m_AddressIP;
  unsigned short m_Port;
  CErrorAccesServeur(unsigned long AddressIP,unsigned short Port)
    : m_AddressIP(AddressIP), m_Port(Port) {}
};
void main(void)
{
  try
  {
     ...
     if( ! PingServeur( AddressIP , Port ) )
        throw CErrorAccesServeur(AddressIP , Port);
```

```
      ...
   }
    catch(CErrorAccesServeur &e) // gestionnaire d'erreur de ce type
    {
       // traitement de l'erreur, par exemple affichage de
       // l'adresse et du port
       cout << "Acces refusé : AdressIP :" << e.m_AddressIP <<
                                          "Port :" << e.m_Port ;
       // autre traitement, arrêt du programme, ou on réessaye...
    }
    catch(...)  // gestionnaire de toutes les erreurs non rattrapées
    {
       // ce gestionnaire rattrape toutes les erreurs, non déjà
       // rattrapées par un gestionnaire d'erreur précédent
       // Il est très pratique, par exemple pour assurer qu'un
       // serveur, ne plante jamais, mais on ne peut récupérer
       // aucune information sur l'erreur rattrapée
    }
}
```

- ✓ La syntaxe de traitement des exceptions ne fait donc appel qu'à trois mots clés : `try`, `throw`, `catch`.
- ✓ Cette syntaxe est la même que celle d'autres langages comme ADA ou Java.
- ✓ Notez que l'on peut imbriquer les blocs try/catch.
- ✓ Notez que un gestionnaire d'exception peut lever une exception avec le mot clé `throw`.

7.2.3 Algorithme du choix du gestionnaire d'exception

- ✓ Lorsqu'une exception levée rencontre un gestionnaire d'exception elle ne sera prise en compte que si une des conditions suivante est vérifiée :
  - o Le type exact de l'objet mentionné dans `throw` est rencontré.
  - o Un type correspondant à une classe de base de l'objet mentionné dans `throw` est rencontré.
  - o Un type correspondant à un pointeur sur une classe dérivée du type mentionné dans `throw` est rencontré.
  - o Le gestionnaire d'exception est de type `catch(...)`.
- ✓ Enfin notez que le mot clé `throw` peut lever une exception sans objet associé. L'exception ne sera donc rattrapée que si elle rencontre un gestionnaire de type `catch(…)`.

7.2.4 Désallocation des ressources dynamiques lors d'un exception

- ✓ Il faut bien comprendre que lorsqu'une exception est lancée, elle remonte pas à pas la pile des fonctions appelées jusqu'à ce qu'elle soit rattrapée par un gestionnaire d'exception approprié.
- ✓ Ceci implique que les objets déclarées dans les fonctions seront détruits (à part les objets statiques).
- ✓ Malheureusement, les objets alloués dynamiquement dans ces fonctions ne seront pas détruits :

```
void fct()
{
  long i   = 8 ;    // allocation statique d'un long
  long * p = new long(9);  // allocation dynamique d'un long
  throw;      // une exception est lancée
```

```
  delete p;   // ce code n'est pas exécuté, par conséquent l'objet
  p=0;        // alloué dynamiquement n'est pas libéré
}  // même lorsque l'exception est lancée, les objets statiques i
   // et p sont détruits, mais pas l'espace pointé par p
void main()
{
  try
  {
    fct();
  }
  catch(...)
  {
  }
}
```

- ✓ Heureusement il existe une astuce pour résoudre ce problème.
- ✓ Elle consiste à définir une classe qui contient un pointeur vers l'objet allouée dynamiquement. Dans le destructeur de la classe, on libère l'objet pointé.
- ✓ Voici le même exemple corrigé:

```
struct long_ptr
{
   long * m_p;
   long_ptr( long * p ): m_p( p ) {}
   ~long_ptr() { if( m_p != 0 ) delete m_p; }
};
void fct()
{
  long i   = 8 ;    // allocation statique d'un long
  long * p = new long(9);  // allocation dynamique d'un long
  long_ptr pptr(p) ;
  throw;      // une exception est lancée
}  // même lorsque l'exception est lancée, les objets statiques i
   // p et pptr sont détruits, par conséquent le destructeur de
   // pptr est appelé et l'objet alloué dynamiquement est détruit
```

- ✓ Notez que l'on a jamais à coder une classe comme `auto_ptr`, puisque la librairie standard (STL) contient la classe `auto_ptr` qui remplit ce rôle, et fait bien plus encore.

7.2.5 Exception lancée dans un constructeur

- ✓ Il est possible de lancer une exception dans un constructeur. Dans ce cas l'objet sera automatiquement détruit, sans appel au destructeur.
- ✓ Cependant, si des objets ont été alloués dans le constructeur avant le lancement de l'exception ils ne seront pas détruits.
- ✓ C++ prévoit une syntaxe spéciale pour ce cas dont voici un exemple :

```
struct CObj
{
   long * m_plong;
   CObj() throw(int);
   ~CObj() { if(m_plong != 0 ) delete m_plong;}
};
CObj::CObj()  throw(int)
try
{
   m_plong = new long(67);
```

```
   throw int(5);
}
catch(int &i)
{ // dans ce cas le gestionnaire d'exception
   if( m_plong != 0)   // renvoie automatiquement l'exception
      delete m_plong;
}
```

- ✓ Les constructeurs des classes de base peuvent eux aussi envoyer des exceptions. Dans ce cas il faut utiliser la syntaxe suivante :

```
CObj::Cobj() try : ClasseDeBase(paramètre) {...} catch ...
```

- ✓ Plus de détails sont dans **[Stroustrup]**.

7.2.6 Exception lancée dans un destructeurs

- ✓ Il existe une règle simple en C++ :

  NE JAMAIS LANCER D'EXCEPTION DANS UN DESTRUCTEUR

- ✓ Les raisons sont assez profondes (voir **[FAQ C++]** par exemple) , mais si cette règle n'est pas respectée, le programme peut planter assez 'salement'.

7.3 Identification dynamique des types (RTTI : Run Time Type Information)

7.3.1 Transtypage dynamique en C++

- ✓ Le transtypage dynamique permet de convertir une expression en un pointeur ou une référence d'une classe, ou un pointeur sur `void`.
- ✓ Il faut bien comprendre que l'on parle de transtypage dynamique, c'est à dire que seules les classes polymorphes sont concernées. Contrairement aux autres transtypages, le compilateur ne sait pas quel est le type de l'expression (parce que par exemple c'est un pointeur d'une classe de base sur une classe dérivée).
- ✓ Le transtypage dynamique est réalisé à l'aide de l'opérateur `dynamic_cast` dont la syntaxe est, `dynamic_cast<type>(expression)` où `type` est le type cible, et `expression` l'expression à transtyper.
- ✓ De plus, on a pas besoin du transtypage dynamique pour convertir un pointeur sur un objet de classe dérivée, en pointeur de classe de base.
- ✓ On a donc besoin du transtypage dynamique que lorsque l'on veut passer d'un pointeur (ou une référence) sur un objet d'une **classe de base <u>polymorphe</u>**, **vers** un pointeur sur un objet d'une **classe dérivée** :

```
struct CBase
{
  virtual int fct() { return 6;}
};
struct CDerive : public CBase
{
  int fct() { return 7;}
};
void main()
{
  CBase * pBase = (CBase*) new CDerive ;
  CDerive * pDerive = dynamic_cast<CDerive*>(pBase);
  if( pDerive )  { /* ca a marché*/ }
  delete pBase; pBase=0; pDerive = 0;
}
```

- ✓ Si `dynamic_cast` ne peut résoudre un transtypage dynamique il retourne le pointeur nul. Il faut donc toujours tester le retour de `dynamic_cast`, puisqu'on ne peut être certains du résultat.
- ✓ Notez qu'à part pour le type cible `void*`, `dynamic_cast` ne peut utiliser les types prédéfinies du langage.
- ✓ Si `dynamic_cast` est appelé sur un pointeur nul, il retourne un pointeur nul.
- ✓ Noter que lorsqu'il y a héritage multiple, `dynamic_cast` ne peut toujours résoudre certains cas simples. Plus de détails dans **[Stroustroup]**.

7.3.2 Identification de type

- ✓ L'opérateur `typeid(expression)` accepte une variable ou le nom d'une classe.
- ✓ `typeid` renvoit un objet de type `type_info` . Cet objet caractérise le type de l 'expression.
- ✓ Si un pointeur est passé on aura le `type_info` du type pointeur.
- ✓ Si on veut avoir le `type_info` de l'objet pointé, il faut déréférencer le pointeur.

- ✓ **Tout le bienfait de `typeid()` est que dans le cas d'un pointeur sur une classe de base déréférencée, il calcule dynamiquement le `type_info` de la classe dérivé et le retourne. Par exemple :**

```
#include <typeinfo>// ce fichier doit être inclus pour le RTTI
struct CBase
{
  virtual int fct() { return 6;}
};
struct CDerive : public CBase
{
  virtual int fct() { return 7;}
};
void main()
{
  CBase * p = (CBase*) new CDerive ;
  If( typeid( *p ) == typeid( CDerive ) ) { /*passe par ici */}
  delete p; p=0;
}
```

- ✓ Voici la définition de `type_info`:

```
class type_info
{
 public:
   virtual ~type_info();
   bool operator==(const type_info &rhs) const;
   bool operator!=(const type_info &rhs) const;
   bool before(const type_info &rhs) const;
   const char * name() const;
 private:
   type_info(const type_info &rhs);
   type_info &operator=(const type_info &rhs)
};
```

- ✓ Un objet de type `type_info` ne peut donc être ni copié ni affecté. On doit donc se limité au `type_info` renvoyé par `typeid()`.
- ✓ On a la possibilité de déterminer l'égalité ou non de deux objets `type_info`.

7.3.3 Quand utiliser le RTTI?

- ✓ La réponse basique est **LE MOINS POSSIBLE !**
- ✓ En effet il existe quelques problématiques bien connues où on ne peut contourner cette technique (par exemple lors d'une sauvegarde et d'un rapatriement d'objets dans un fichier binaire voir **[Stroustrup]**).
- ✓ Souvent les 'nostalgiques du C' veulent utiliser le RTTI pour switcher parmi les types possibles d'un pointeur, pour appeler une fonction sur l'objet pointé : CETTE PRATIQUE EST A PROSCRIRE, en effet le polymorphisme fait ça beaucoup mieux !!!
- ✓ Il faut vraiment se renseigner sur les motifs objets **[Design Pattern]** avant d'utiliser le RTTI. Souvent ils évitent son utilisation.

7.3.4 Opérateurs de transtypage statiques

- ✓ Le C++ a deux opérateurs de transtypage statique, **`static_cast`** et **`reinterpret_cast`**.
- ✓ **`static_cast`**`<type> expression` remplace l'ancien opérateur de transtypage `(type)` du C (qui reste utilisable en C++). Avec deux fonctionnalités en plus :
  - o On peut convertir d'un type entier vers des énumération.
  - o On peut convertir d'une classe de base vers une classe dérivée (l'inverse étant automatique).

  `static_cast` ne fait aucun vérification contrairement à dynamic_cast, et c'est son énorme défaut, puisque la responsabilité du programmeur est engagée**. Il vaut mieux ne jamais l'utiliser**, et utiliser l'ancien transtypage `(type)` du C qui ne permet pas ceci. ou `dynamic_cast` qui au moins vérifie la validité du transtypage.
- ✓ **`reinterpret_cast`**`<type> expression` effectue une conversion intrinsèquement dangereuse et mal définie, entre deux types pointeurs, ou entre un type pointeur ou un type entier. l'ancien opérateur de transtypage `(type)` du C (qui reste utilisable en C++). Vous l'aurez compris, **il ne faut jamais l'utiliser !**

7.4 La généricité (template ou modèle)

7.4.1 Introduction

- ✓ La **généricité** est un concept qui peut s'appliquer sur des fonctions ou sur des classes. Ce concept s'appelle **template** en anglais.
- ✓ Concrètement on peut faire dépendre une fonction ou une classe d'un ou plusieurs type, donc une liste de type. Par exemple :
  - o Une fonction qui retourne le minimum des 2 arguments d'entrées, quelquesoit le type des arguments d'entrée, du moment qu'il surcharge un opérateur de comparaison.
  - o Une classe qui définit le comportement d'une pile, quelquesoit le type `T` des éléments de la pile. Les opérations `push(T element)` et `T pop()` doivent être implémentées.
- ✓ Au moment où l'on utilise un modèle de fonction, ou de classe, le compilateur génère le code pour la liste de type fournie. Cette étape est appelée **génération des templates**. Dans le fichier exécutable on a donc bien le code de la fonction ou de la classe, dupliqué pour chaque liste de type utilisée.
- ✓ Une liste de type se présente sous la forme : `<class type1,...,typename typeN>`. Comme on le voit on peut indifféremment utiliser les mots clés `class` ou **`typename`**.
- ✓ On a la possibilité de définir des types par défaut. Comme pour la surcharge des fonctions ces types se trouvent à gauche dans la liste :
  `<typename U,class V,class W=int, typename X=double>`

- ✓ Ce cours n'a pas la prétention d'exposer ce mécanisme compliquée dans les détails. En effet un certain nombres de questions se pose lorsqu'on utilise les template. Pour un exposé complet voir **[Stroustrup]**.

7.4.2 Les fonctions génériques

- ✓ La déclaration et la définition d'une fonction générique se fait exactement comme si elle était une fonction normale, à ceci prés quelle est précédée de la liste des types génériques. Voici un exemple de code d'une fonction générique qui retourne le maximum des ses 3 arguments, quelquesoit leur type du moment qu'il surcharge l'opérateur `'>'` :

```
template <typename T> T Max( T param1, T param2, T param3)
{
  if( param1 > param2 && param1 > param3 ) return param1 ;
  if( param2 > param1 && param2 > param3 ) return param2 ;
  return param3;
}
void main()
{
  int i = Max(1 , 3 , 5);
  double d = Max(1.1 , 3.1 , 5.1);
  // ici i vaut 5 et d vaut 5.1
}
```

- ✓ Malgré les apparences, il y a 2 fonctions `Max()` générée par le compilateur. Une ou le type `T` est `int` et une ou le type `T` est `double`.

7.4.3 Les classes génériques

- ✓ La déclaration d'une classe générique se fait exactement comme si elle était une classe normale, à ceci prés quelle est précédée de la liste des types génériques. Voici un exemple de code d'une classe dont les objets sont des vecteurs de dimension 3. Le type des éléments est génériques. On définit les opérations produit vectoriel '^' et produit scalaire '*':

```
template <typename T> class VecteurDim3
{
  private:
    T m_x,m_y,m_z;
  public:
    VecteurDim3(T x, T y, T z): m_x(x), m_y(y), m_z(z) {}
    // produit vectoriel
    VecteurDim3 operator^( VecteurDim3& arg ) const
    {
       VecteurDim3 tmp;
       tmp.m_x = m_y*arg.m_z - m_z*arg.m_y;
       tmp.m_y = m_z*arg.m_x - m_x*arg.m_z;
       tmp.m_z = m_x*arg.m_y - m_y*arg.m_x;
       return tmp ;
    }
    // produit scalaire
    T operator*( VecteurDim3& arg ) const
    {
          return m_x*arg.m_x + m_y*arg.m_y + m_z*arg.m_z;
    }
};
void main()
{
  VecteurDim3<double> v1(1,2,3);
  VecteurDim3<double> v2(4,5,6);
  VecteurDim3<double> v3(0,0,0);
  v3 = v1 ^ v2;
  double d = v3 * v2;
}
```

- ✓ Le type générique doit supporter les opérations '*' '+' '-'.
- ✓ Une particularité subsiste. Si la méthode est déclarée hors de la classe, elle doit elle aussi être déclarée générique :

```
template <typename T> class VecteurDim3
{
    ...
    T operator*( VecteurDim3& arg ) const ;
};
void main()
template <typename T>
T VecteurDim3<T>::operator*( VecteurDim3<T> & arg ) const
{
      return m_x*arg.m_x + m_y*arg.m_y + m_z*arg.m_z;
}
```

7.4.4 Les méthodes génériques

- ✓ Mis à part les destructeurs les méthodes d'une classe peuvent être génériques, que la classe soit elle-même générique ou pas.
- ✓ Si la classe n'est pas générique, on utilise la même syntaxe que pour une fonction générique.
- ✓ Si la classe est aussi générique, il faut spécifier 2 fois la syntaxe `template`, une fois pour la classe et une fois pour la fonction. Si la méthode générique est définie à l'intérieure de la classe, il n'est pas nécessaire de donner les paramètres génériques de la classe. Dans ce cas on utilise donc encore la même syntaxe que pour une fonction générique.
- ✓ Notez qu'une méthode ne peut être virtuelle et générique à la fois.
- ✓ Bien qu'il soit courant d'utiliser des fonctions et des classes générique,s il est plus rare de devoir utiliser des méthodes génériques, surtout si la classe est déjà générique. Avant d'utiliser les méthodes génériques, il vaut mieux se demander si l'on est sûr d'en avoir besoin.

7.4.5 Instanciation des templates

- ✓ Lorsque tous les types d'une liste générique de type sont spécifiés, le compilateur génére l'instanciation des templates, c'est à dire qu'il fabrique le code de la fonction, classe ou méthode, en utilisant les types fournis.
- ✓ Une contrainte assez lourde survient lorsqu'on instancie une classe générique (resp. qu'on appelle une fonction générique) : les corps de toutes les méthodes (resp. le corps de la fonctions) doit être définit.. En plus de la lourde contrainte imposées au programmeur, qui doit modifier ses habitudes, cela implique que :
    - Les instances des templates sont compilées pour chaque liste de type et pour chaque fichier objet, d'où une performance moindre du compilateur.
    - Le code des instances des templates sont en multiples exemplaires dans les fichiers objets, d'où une augmentation de la taille de l'exécutable.

  Notez que ces problèmes sont maintenant plus ou moins bien résolus selon les compilateurs (fichiers d'entête précompilés, partage des instances de template lors de l'édition de liens…). Il est recommandé de consulter la documentation de votre compilateur.
- ✓ Notez que l'instanciation de template peut se produire de deux façons : implicitement ou explicitement.
    - L'instanciation de template implicite se produit par exemple lorsque le compilateur doit instancier une fonction générique à partir des types des paramètres d'appel et de retour, lors de l'appel d'une fonction générique dans le code. Si il y a ambiguité le compilateur retournera une erreur.
    - L'instanciation de template implicite est une technique permettant au programmeur de forcer le compilateur à instancier un template. Par exemple dans les exemples précédents, le programmeur peut rajouter une des lignes suivantes pour instancier explicitement la classe ou la fonction générique:

```
template int Max(int,int,int);
template VecteurDim3<double>;
```

- ✓ Il est vivement conseillé d'utiliser l'instanciation de template explicite. D'ailleurs certain compilateurs permettent d'interdire l'instanciation de template implicite.

7.4.6 Autres possibilités avec la généricité

- ✓ C++ autorise l'utilisation de fonctions ou de classes génériques amies. La démarche est la même que celle vu en 6.6. Par exemple on peut avoir les cas de figure :

```
template <class E> struct ma_classe
{
  friend class une_classe<int>;
  friend void une_fonction(float);
};

template <class E> struct ma_classe
{
  friend class une_classe<E>;
  friend void une_fonction(E);
};

template <class E> struct ma_classe
{
  template <class F> friend class nom_classe <F>;
  template <class G> friend type_retour nom_fonction<G>;
};
```

- ✓ On peut spécialiser une instance de template d'une classe ou une fonction générique pour certains types. Concrètement on redéfinit à l'aide d'un nouveau code le comportement. Par exemple pour la fonction générique Max() :

```
struct S
{
      int         val;
      void *      ptr;
};

template <> S Max( S param1, S param2, S param3)
{
  if( param1.val > param2.val && param1.val > param3.val ) return
param1 ;
  if( param2.val > param1.val && param2.val > param3.val ) return
param2 ;
  return param3;
}
```

- ✓ On peut utiliser des types génériques dans la liste des types d'un template !
- ✓ La norme C++ permet grâce au mot clé `export` de s'affranchir de la définition du corps d'une fonction ou du corps des méthodes d'une classe, avant son instanciation. Cependant aucun compilateur ne supporte cette spécificité à ce jour.
- ✓ Pour plus de détails sur ces possibilités complexes consulter **[Stroustrup]**.

8 La STL (la librairie générique standard)

- ✓ La STL, Standard Template Library ou la librairie générique standard, est un ensemble de librairies maintenant bien défini par la norme ISO.
- ✓ Le mot standard est utilisé pour signifier que ces librairies devraient être disponibles dans chaque implémentation du langage C++.
- ✓ Le mot générique est utilisé pour montrer que la plupart des fonctionnalités disponibles le sont au travers de fonctions et de classes génériques. Par exemple on peut définir des listes d'objets, le type des objets étant un paramètre générique.
- ✓ Concrètement la STL contient :
  - o Des types d'aides à la manipulation des chaînes de caractères.
  - o Des types numériques, comme les complexes.
  - o Des utilitaires généraux, comme les objets fonctions et les auto pointeurs..
  - o Des types d'aide à la gestion des flux d'entrée sortie.
  - o Des supports du langage comme le RTTI.
  - o Des outils d'aide à la gestion des particularités locales (calendrier…).
  - o Des conteneurs qui permettent de stocker et de manipuler des collections d'objets.
  - o Des itérateurs qui aident à travailler sur des collections d'objets.
  - o Des algorithmes qui aident à consulter et à manipuler des collections d'objets.
- ✓ Les déclarations requises pour utiliser la bibliothèque standard C++ se trouvent dans 32 fichiers en-tête :

```
<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>
```

- ✓ Les noms ci-dessus peuvent ne pas être des noms de fichier corrects, ou ne pas nommer des fichiers existants sur votre système ; la manière de les transformer en de vrais noms de fichier dépend de l'implémentation.
- ✓ Les noms de tous les éléments de la bibliothèque standard, sauf les macros et les opérateurs `new` et `delete`, sont membres de l'espace de noms `std` ou d'espaces de noms imbriqués dans `std`.
- ✓ Lorsque le risque de collision de nom est minime on peut donc utiliser la directive : `using namespace std;`
- ✓ Les éléments de la bibliothèque standard de C appartiennent aussi à la bibliothèque standard de C++ à travers les fichiers en-tête suivants, qui renvoient de manière évidente aux fichiers correspondants de la bibliothèque C :

```
<cassert> <cctype> <cerrno> <cfloat> <ciso646>
<climits> <clocale> <cmath> <setjmp> <csignal>
<cstdarg> <cstddef> <cstdio> <cstdlib> <cstring>
<ctime> <cwchar> <cwtype>
```

8.1 Les types complémentaires

8.1.1 Les chaînes de caractères : la classe string

- ✓ Fichiers en-tête concernés par la gestion de chaînes de caractères :
  `<string>`
  `<cstdlib>`
  `<cstring>`
  `<cctype>`
  `<cwtype>`
  `<cwchar>`
- ✓ Le fichier `<string>` définit le modèle de classe `char_traits` et surtout, la classe `string` (voir ci-dessous).
- ✓ Le modèle `char_traits` déclare un ensemble d'opérations de bas niveau (comparaison, recherche, déplacement, copie, etc.) sur les types pouvant jouer le rôle de caractère dans un objet `string`. Deux spécialisations de ce modèle sont particulièrement utiles : `char_traits<char>` et `char_traits<wchar_t>`.
- ✓ Les cinq autres fichiers listés ci-dessus définissent, comme leurs homologues de la bibliothèque C, les utilitaires tournant autour des `char` et des `char *` et, plus généralement, des séquences terminées par zéro.

- ✓ La classe `string` est un type simple à utiliser, sûr, complet et performant pour le traitement des chaînes de caractères, presque toujours très largement préférable au pénible type `char *` de C.
- ✓ En réalité, `string` n'est qu'un nom pour une classe qui est la spécialisation d'un modèle :

```
typedef basic_string<char> string;
```

- ✓ Le modèle `basic_string` offre de nombreuses fonctionnalités, communes aux chaînes de `char`, aux chaînes de `wchar_t`, voire aux chaînes d'autres types. Sa déclaration est touffue et difficile à lire ; pour cette raison, nous allons nous intéresser au cas des `char` uniquement.
- ✓ La classe `string` fonctionne *comme si* elle avait été ainsi définie :

```
class string
{
  public:
    // construction par défaut (chaîne vide)
    explicit string();

    // construction avec les n premiers éléments d'un tableau de caractères
    // (par défaut: tous les caractères)
    string(const char *s, size_t n);

    // construction par répétition d'un caractère
    string(size_t n, char c);

    // construction avec n caractères d'une autre chaîne (par défaut: jusqu'au bout)
    // à partir du caractère de rang pos (par défaut: le premier)
    string(const string &str, size_t pos = 0, size_t n = npos);

    // construction avec les caractères fournis par une itération
    // (au sens des itérateurs de la STL)
```

```
template<class inIter> string(inIter begin, inIter end);

// destruction
~string();

// affectation
string &operator=(const string &str);
string &operator=(const char *s);
string &operator=(char c);

// iterateurs
itérateur begin();
itérateur_constant begin() const;
itérateur end();
itérateur_constant end() const;
itérateur_inverse rbegin();
itérateur_inverse_constant rbegin() const;
itérateur_inverse rend();
itérateur_inverse_constant rend() const;

// nombre de caractères (ces deux fonctions sont les mêmes)
size_t size() const;
size_t length() const;

// nombre maximum de caractères
size_t max_size() const;

// changement du nombre de caractères : l'allongement se fait par recopie de c
// ou, par défaut '\0'
void resize(size_t n, char c);
void resize(size_t n);

// accès au ième caractère (reference est « char & », const_reference est
// « const char & »):
const char &operator[](size_t pos) const;
char &operator[](size_t pos);
const char &at(size_t n) const;
char &at(size_t n);

// concaténation avec une chaîne, un char * , un char ou le produit d'une itération
string &operator+=(const string &str);
string &operator+=(const char *s);
string &operator+=(char c);
string &append(const string &str);
string &append(const string &str, size_t pos, size_t n);
string &append(const char *s, size_t n);
string &append(const char *s);
string &append(size_t n, char c);
template<class inIter> string &append(inIter first, inIter last);
void push_back(const char);

// modification des caractères d'une chaîne
string &assign(const string&);
string &assign(const string &str, size_t pos, size_t n);
string &assign(const char *s, size_t n);
string &assign(const char *s);
string &assign(size_t n, char c);
template<class inIter> string &assign(inIter first, inIter last);

// insertion parmi les caractères d'une chaîne
string &insert(size_t pos1, const string &str);
string &insert(size_t pos1, const string &str,_t pos2, size_t n);
string &insert(size_t pos, const char *s, size_t n);
string &insert(size_t pos, const char *s);
string &insert(size_t pos, size_t n, char c);
itérateur insert(itérateur p, char c);
void insert(itérateur p, size_t n, char c);
template<class inIter>
void insert(itérateur p, inIter first, inIter last);

// suppression de caractères au milieu d'une chaîne
```

```
  string &erase(size_t pos = 0, size_t n = npos);
  itérateur erase(itérateur position);
  itérateur erase(itérateur first, itérateur last);

  // remplacement de caractères d'une chaîne
  string &replace(size_t pos1, size_t n1, const string &str);
  string &replace(size_t pos1, size_t n1,
  const string &str, size_t pos2, size_t n2);
  string &replace(size_t pos, size_t n1, const char *s, size_t n2);
  string &replace(size_t pos, size_t n1, const char *s);
  string &replace(size_t pos, size_t n1, size_t n2, char c);
  string &replace(itérateur i1, itérateur i2, const string &str);
  string &replace(itérateur i1, itérateur i2, const char *s, size_t n);
  string &replace(itérateur i1, itérateur i2, const char *s);
  string &replace(itérateur i1, itérateur i2, size_t n, char c);
  template<class inIter> string &replace(
  itérateur i1, itérateur i2, inIter j1, inIter j2);
  size_t copy(char *s, size_t n, size_t pos = 0) const;
  void swap(string);

  // obtention explicite du char * sous-jacent. Dans le cas de c_str, un caractère
  // nul est présent à la fin :
  const char *c_str() const; // explicit
  const char *data() const;

  // recherche de chaînes et de caractères dans une chaîne ;
  // fonctions donnant la position la plus à gauche (« première occurrence ») :
  size_t find (const string &str, size_t pos = 0) const;
  size_t find (const char *s, size_t pos, size_t n) const;
  size_t find (const char *s, size_t pos = 0) const;
  size_t find (char c, size_t pos = 0) const;

  // fonctions donnant la position la plus à droite (« dernière occurrence ») :
  size_t rfind(const string &str, size_t pos = npos) const;
  size_t rfind(const char *s, size_t pos, size_t n) const;
  size_t rfind(const char *s, size_t pos = npos) const;
  size_t rfind(char c, size_t pos = npos) const;

  // première occurrence d'un caractère d'un ensemble
  size_t find_first_of(const string &str, size_t pos = 0) const;
  size_t find_first_of(const char *s, size_t pos, size_t n) const;
  size_t find_first_of(const char *s, size_t pos = 0) const;
  size_t find_first_of(char c, size_t pos = 0) const;

  // dernière occurrence d'un caractère d'un ensemble
  size_t find_last_of (const string &str, size_t pos = npos) const;
  size_t find_last_of (const char *s, size_t pos, size_t n) const;
  size_t find_last_of (const char *s, size_t pos = npos) const;
  size_t find_last_of (char c, size_t pos = npos) const;

  // première occurrence d'un caractère qui n'est pas dans un ensemble
  size_t find_first_not_of(const string &str, size_t pos = 0) const;
  size_t find_first_not_of(const char *s, size_t pos, size_t n) const;
  size_t find_first_not_of(const char *s, size_t pos = 0) const;
  size_t find_first_not_of(char c, size_t pos = 0) const;

  // dernière occurrence d'un caractère qui n'est pas dans un ensemble
  size_t find_last_not_of (const string &str, size_t pos = npos) const;
  size_t find_last_not_of (const char *s, size_t pos, size_t n) const;
  size_t find_last_not_of (const char *s, size_t pos = npos) const;
  size_t find_last_not_of (char c, size_t pos = npos) const;

  // comparaison de chaînes
  int compare(const string &str) const;
  int compare(size_t pos1, size_t n1, const string &str) const;
  int compare(size_t pos1, size_t n1, const string &str,
  size_t pos2, size_t n2) const;
  int compare(const char *s) const;
  int compare(size_t pos1, size_t n1, const char *s,
  size_t n2 = npos) const;
};
```

- ✓ On notera que, contrairement à la classe `CString` de la bibliothèque *MFC*, il n'existe pas de conversion *implicite* d'une valeur de type `string` en une valeur `char *`.
- ✓ Pour une telle conversion, il faut *explicitement* appeler la fonction membre **`c_str`**.

- ✓ Voici deux exemples où l'utilisation de la classe `string` au lieu de `char *` simplifie grandement la tâche du développeur :
- ✓ Construction du chemin d'accès à un fichier version `char *` :

```
#include <string.h>
void main()
{
	char tmp[80]; // en espérant que ca suffit!!
	char * Lecteur = "c:/";
	char * Path = "home/mesdocuments/";
	char * NomFichier = "Fichier.txt";
	strcpy( tmp , Lecteur );
	strcat( tmp , Path );
	strcat( tmp , NomFichier );
	// fopen( tmp ,...
}
```

Construction du chemin d'accès à un fichier version `string` :

```
#include <string>
using namespace std;
void main()
{
	string Lecteur = "c:/";
	string Path = "home/mesdocuments/";
	string NomFichier = "Fichier.txt";
	string tmp = Lecteur + Path + NomFichier;
	// notez l'allocation dynamique de la mémoire
	// fopen( tmp ,...
}
```

- ✓ Attribut d'une classe type chaîne de caractères dynamique version `char *` :

```
#include <string>
class CPersonne
{
	char * m_Nom;
	void CopyNom(  const char * Nom )
	{
		m_Nom = new char[strlen(Nom)+1];
		strcpy( m_Nom , Nom );
	}
public:
	CPersonne( const char * Nom )
	{
		CopyNom( Nom );
	}
	//une variable allouée dynamiquement implique la régle des 3:
	// le destructeur,
	// le constructeur de copie,
	// l'opérateur d'affectation
	~CPersonne()
	{
		delete [] m_Nom;
	}
	CPersonne( const CPersonne & tmp )
	{
		CopyNom( tmp.m_Nom );
	}
	operator=( const CPersonne & tmp )
	{
		if( &tmp != this )
		{
			delete [] m_Nom;
			CopyNom( tmp.m_Nom );
		}
	}
};
```

Attribut d'une classe type chaîne de caractères dynamique version string :

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

class CPersonne
{
	string m_Nom;
public:
	CPersonne( const char * Nom ): m_Nom( Nom ) {}
	// le destructeur par défaut,
	// le constructeur de copie par défaut,
	// l'opérateur d'affectation par défaut
	// suffisent tout à fait
};
```

8.1.2 Les nombres complexes

- ✓ Il n'est pas question de présenter les nombres complexes ici, en revanche il est utile de se remémorer que l'ensemble des complexes est à la fois un corps, et un espace vectoriel de dimension 2 sur les réels.
  - o Un corps implique que les 4 opération arithmétiques + - * / existent sur les complexes.
  - o Un espace vectoriel de dimension 2 sur les réels implique que chaque nombre complexe peut être vu comme un couple de 2 nombres réels.
  - o De plus on peut contraindre les 2 composantes réelles d'un nombre complexe à être entières, auquel cas nous ne sommes plus dans un corps mais dans l'anneau des entiers de gauss, ou seulement les opérations + - * sont définies.
- ✓ Le fichier `<complex>` déclare un modèle et trois spécialisations prédéfinies (l'utilisateur peut ajouter ses propres spécialisations) :

```
template<class T> class complex;
template<> class complex<float>;
template<> class complex<double>;
template<> class complex<long double>;
```

- ✓ Ce fichier déclare également, soit comme fonctions membres, soit comme des fonctions indépendantes, la plupart des opérations mathématiques que l'on peut envisager de faire avec des nombres complexes :
  constructeurs, sélecteurs, opérations arithmétiques, comparaisons, travail avec la forme polaire, fonctions transcendantes, etc.

8.1.3 Les valarray

- ✓ Les programmeurs sont souvent confrontés au problème suivant :
  Appliquer un traitement sur un grand nombre de données de même type.
- ✓ Par exemple les algorithmes de compressions/décompressions multimédia rentre dans ce type de traitement.
- ✓ Pour optimiser ce traitement il faut l'effectuer en parallèle sur plusieurs processeurs, mais la répartition des tâches sur les processeurs se fait au niveau du système d'exploitation.
- ✓ L'idée du `valarray` est de créer un tableau de données, puis de faire subir le traitement au tableau lui-même. Le compilateur garantit que ce traitement est en fait effectué sur chaque élément du tableau, et d'une manière optimisée si il y a plusieurs processeurs.
- ✓ Le programme suivant crée un tableau de 30 éléments de type double qui valent chacun 3.14. Chaque élément est élevé au carré, puis on ajoute à chaque élément 2.1.

```
#include <valarray>
using namespace std;
void main()
{
      valarray<double> Tab(3.14,30);
      Tab *= Tab;
      Tab += 2.1;
}
```

- ✓ Naturellement une condition nécessaire est que les opérateurs utilisés sur le tableau soient disponibles sur le type des éléments.
- ✓ Voici quelques fonctionnalités du valarray :
  - o La méthode `sum` retourne la somme de tous les éléments.
  - o On peut utiliser les opérateurs binaires de comparaisons qui retournent alors un tableau de booléens.
  - o On peut utiliser d'autres opérateurs binaires comme '+' entre deux `valarray`. Le résultat est alors un `valarray` dont les éléments sont la somme des deux éléments de même index.
  - o On peut effectuer des décalages (resp. des rotations) des éléments en utilisant les méthodes `shift(int)` (resp. `cshift(int)`). Le paramètre entier représente le nombre de positions à sauter, vers la gauche s'il est positif vers la droite sinon. Les nouveaux éléments introduits dans le cas où il n'y a pas de remplaçant (seulement lors d'un décalage) prennent l'état spécifié par le constructeur par défaut.
- ✓ Dans ce type de traitement il arrive souvent que l'on ne veuille appliquer le traitement qu'à un sous ensemble des éléments du tableau. Il faut donc effectuer une sélection de plusieurs éléments. Cette sélection peut se réaliser de quatre façons différentes que nous ne détaillerons pas ici (voir **[Stroustrup]** ou **[Casteyde]**).

8.1.4 Les auto pointeurs

- ✓ Le fichier `<memory>` déclare plusieurs classes utilisées pour la gestion de la mémoire, mais la seule classe régulièrement utilisée par les programmeurs est `auto_ptr<>`. Les autres possibilités fournies par ce fichier sont surtout utilisées par la STL elle même.
- ✓ Voici les principales fonctionnalités d'un auto pointeur :
  - o Un auto pointeur contient un pointeur dynamiquement alloué par l'opérateur `new`.
  - o Un auto pointeur a la possibilité d'être propriétaire de son pointeur, auquel cas il ne doit pas y avoir d'autres auto pointeur propriétaires de ce pointeur.
  - o La copie d'un auto pointeur transfère la propriété du pointeur de l'auto pointeur source à celui de destination (que ce soit copie par affectation ou initialisation).
  - o Lorsque qu'un auto pointeur propriétaire est détruit, il appelle l'opérateur `delete` sur l'objet pointé.
  - o Toutes les méthodes que l'on peut appeler sur le pointeur peuvent être appelées sur l'auto pointeur car l'opérateur `->` est surchargé.
- ✓ Vous l'aurez compris, bien utilisé par un programmeur averti les auto pointeurs sont puissant, mais ils restent extrêmement dangereux.
- ✓ L'idée de base est de ne pas à avoir à appeler `delete` :

```
#include <memory>
#incldue <string>
using namespace std;
void main()
{
        string * p = new string(5);
        auto_ptr<string> p1( p );
        // travail avec p
        int size = p1->size() ;
        ...
}// p est automatiquement libéré lorsque p1 est détruit,
 // parce que p1 est propriétaire de p
```

- ✓ Il faut absolument éviter d'avoir 2 auto pointeurs propriétaires du même pointeur :

```
{
        int * p = new int(5);
auto_ptr<int> p1( p );
auto_ptr<int> p2( p );
}// ici p est libéré 2 fois et le programme plante
```

- ✓ De plus il vaut mieux éviter d'avoir à copier un auto pointeur, car le risque de ne plus savoir qui possède qui est très grand. Entre autre, il vaut mieux ne jamais passer un auto pointeur comme argument d'une fonction. On peut toujours récupérer le pointeur avec la fonction `auto_ptr<>::get()`. Si l'on veut récupérer le pointeur tout en supprimant la propriété on peut appeler la fonction `auto_ptr<>::release()`.
- ✓ En résumé, à moins d'être masochiste ou de vouloir saboter un projet, il vaut mieux ne jamais copier et ne jamais passer à une fonction un auto pointeur. L'utilisation d'un auto pointeur évite tout simplement d'avoir à appeler `delete` à la fin de la porté courante mais son utilité maximum est ailleurs :
  Dans le cas ou une exception est lancée, l'objet dynamiquement alloué sera quand même détruit.

8.1.5 Les objets fonctions (foncteurs)

- ✓ On désigne par objet fonction (ou foncteur) toute entité qui peut être appelée comme une fonction, c'est à dire :
  - o Une fonction
  - o Un pointeur sur une fonction ou une méthode
  - o Un objet dont la classe surcharge l'opérateur `( )`.
- ✓ Ici, seul le troisième cas nous intéresse.
- ✓ Concrètement vous utiliserez ce type d'objet pour pouvoir accéder à certaines fonctionnalités de la STL.
- ✓ Ces fonctionnalités ont besoin d'une fonction en paramètre (par exemple un tri a besoin d'une fonction de comparaison) mais ne peuvent accepter une fonction comme paramètre. En effet ces fonctionnalités sont génériques. Par conséquent elles acceptent des classes en paramètre. D'où le besoin de permettre à une classe de représenter une fonction.
- ✓ La STL nous propose de nombreux modèles de foncteurs dans le fichier <**functional**>, comme des opérations arithmétiques ou des opérations de comparaison:

```
// opérations arithmétiques
template<class T> struct plus;
template<class T> struct minus;
...
// comparaisons
template<class T> struct equal_to;
template<class T> struct greater;
...
```

- ✓ On a aussi accès aux 2 modèles de base :

```
template<class Arg, class Result> struct unary_function;
template<class Arg1, class Arg2, class Result> struct binary_function;
```

- ✓ Par exemple une implémentation de la class `plus` dérivant de `binary_function` serait:

```
template<class T> struct plus : binary_function<T, T, T>
{
        T operator()(const T &x, const T &y) const  { return x + y; }
};
```

- ✓ Consultez le chapitre sur les algorithmes 8.4.9 pour voir quand utiliser les foncteurs.

8.2 Les flux (ou flots) d'entrée/sortie

8.2.1 Introduction

- ✓ Un flux, en informatique, est un canal sur lequel on peut écrire ou lire des données.
- ✓ Accessibles en C++ avec les opérateurs << (sortie standard) et >> (entrée standard), leur emploi est particulièrement facile. Les opérateurs sont utilisés pour commander une lecture ou une écriture.
- ✓ Les opérateurs sont conçus pour prendre en premier paramètre le nom de l'objet représentant le flux et en second paramètre la valeur à écrire. Ces opérateurs renvoient une référence sur le flot, ce qui permet d'enchaîner les appels.
- ✓ Nous ne détaillerons pas les flux. Cependant il faut être conscient que la plupart du temps les programmeurs utilisent des flux de `char` voir de `wchar_t`, et qu'en fait les flots sont des classes génériques :

```
typedef basic_iostream<char> iostream ;
typedef basic_iostream<wchar_t> wiostream;
```

  Cependant nous nous contenterons ici seulement des flots de `char`.
- ✓ Nous ne détaillerons pas tous les fichiers réservés pour les flots. Nous nous limiterons au fichier <**iostream**>, et <**fstream**>.

8.2.2 Les flux standard

- ✓ Le fichier **<iostream>** déclare quatre flux de `char` standard :

```
extern istream cin;  //l'entrée standard  (clavier par défaut)
extern ostream cout; //la sortie standard (terminal par défaut)
extern ostream cerr; //un canal d'erreur  (terminal par défaut)
extern ostream clog; //connecté au canal d'erreur et bufferisé
```

- ✓ Ces flux sont ouverts lorsque le programme commence. Ils contrôlent les insertions et extractions de caractères dans ou depuis les unités **stdin**, **stdout** et **stderr** (déclarées dans **<cstdio.h>**).

8.2.3 La classe de flux de sortie : ostream

- ✓ La classe `ostream` définit un flot de sortie.
- ✓ Les 3 méthodes principales de `ostream` sont :
  - o `put(char c)` : écrit un caractère *c* sur le flot, renvoie le flot
  - o `write(char* c, int l)` : écrit *l* caractères sur le flot, depuis l'adresse *c* (pas de formatage), renvoie le flot. L'intérêt principal de cette méthode est quelle n'a pas besoin de formatage, donc peut être utilisée pour les fichiers binaires.
  - o `<< expression` : formate et écrit l'expression sur le flot (types de base automatiquement reconnus), renvoie le flot
- ✓ Voici des exemples d'utilisation:

```
cout << " Total : " << 123.67 << " euros.";
cout.put('h').put('e').put('l').put('l').put('o');
cout.write("hello",5).write(" ca va? ",8);
```

8.2.4 La classe de flux d'entrée : istream

- ✓ La classe `istream` définit un flot d'entrée.
- ✓ Les méthodes principales de `istream` sont :
  - o `>> variable` : extrait les caractères nécessaires du flot pour former une valeur compatible avec le type de la variable réceptrice, renvoie une référence sur le flot
  - o `get(char c)` : lit un caractère sur le flot (quel qu'il soit), et stocke dans *c*, renvoie le flot
  - o `get()` : extrait un caractère du flot et renvoie un entier; En particulier elle renvoie `EOF` si la fin du flot est atteinte.
  - o `getline(char* c, int l, char delim='\n')` : lit une chaîne d'au plus l caractères (si `delim` pas rencontré, limitée à `delim` sinon) et les range à l'adresse `c`; le caractère `'\0'` est ajouté en fin de chaîne
  - o `gcount()` : renvoie le nombre de caractères effectivement lus lors du dernier appel de `getline()`
  - o `read(char* c, int l)` : lit `"l"` caractères sur le flot et les range à l'adresse `"c"`, aucune supposition n'est faite sur la nature des caractères à lire (utilisé pour les fichiers binaires par exemple créés avec `ostream.write()`).
- ✓ Lors de l'acquisition avec l'opérateur `>>`, la fin de saisie est obtenue par appui sur la touche entrée si on lit sur `cin`.
- ✓ Voici un exemple d'utilisation :

```
int a ; double d ;
cout << "Entrez un entier puis un double: ";
cin >> a >> d;
```

8.2.5 Manipulation de fichiers

- ✓ Le fichier <**fstream**> déclare les classes `ifstream` et `ofstream` prévues pour la lecture et l'écriture de données à partir de fichiers.
- ✓ Voici un exemple d'utilisation :

```
// ecriture
ofstream fw("res.txt",ios::out);
fw << " Total : " << 123.67 << " euros.";
fw.close();
// lecture
#define MAX_SIZE 255
char buf[MAX_SIZE];
ifstream fr("res.txt",ios::in);
if (fr)
while (fr.getline(buf,MAX_SIZE))
cout << fr << "\n";
fr.close();
```

- ✓ Les mots d'états d'accès aux fichiers sont les suivants (pour en utiliser plusieurs il faut les séparer '|').

```
ios::in    lecture seule       ios::ate       se place en fin de fichier après ouverture
ios::out   écriture seule      ios::trunc     écrase le fichier s'il existe
ios::app   ajout de données    ios::nocreate  le fichier doit exister
                               ios::noreplace le fichier ne doit pas exister
```

8.2.6 Gestion des erreurs sur un flux

- ✓ A chaque flux est associé un ensemble de bits d'erreur dont voici les principaux :
  - o `goodbit` activé si tout va bien.
  - o `eofbit` activé si la fin du flux est atteinte.
  - o `failbit` activé si la prochaine opération d'E/S ne pourra se faire.
  - o `badbit` activé si le flux est irrécupérable.
- ✓ Si le flot est dans un état d'erreur, les opérations ne peuvent s'effectuer tant que l'erreur n'a pas été corrigée (indépendamment du flot) et que le bit d'erreur correspondant n'a pas été désactivé. Pour réaliser la seconde condition, on dispose de méthodes appartenant à la classe `ios` pour connaître la valeur d'un bit, et pour la modifier. Ces méthodes sont les suivantes :
  - o `eof()` renvoie l'état du bit `eofbit`
  - o `bad()` renvoie l'état du bit `badbit`
  - o `fail()` renvoie l'état du bit `failbit`
  - o `good()` renvoie 1 si aucun des 3 bits précédents n'est activé
  - o `rdstate()` renvoie le statut d'erreur du flot
  - o `clear(b)` active le bit passé en paramètre, et met tous les autres à 0. On fixe en fait le statut d'erreur du flot
- ✓ Si l'on souhaite activer un bit sans modifier la valeur des autres, on passera en paramètre à la fonction `clear()` le nom du bit à activer et le résultat de l'appel de la fonction `rdstate()` séparé par le caractère '|'. Ainsi on affecte l'ancien statut d'erreur plus un bit particulier.

8.2.7 Surcharge des opérateurs de redirection

- ✓ Pour toutes les classes vues précédemment, on peut donner une nouvelle signification aux symboles `<<` et `>>`. Pour cela, il faut respecter quelques consignes :
  - o Les opérateurs doivent recevoir un flux en premier argument.
  - o La valeur de retour est une référence sur le flux concerné.
  - o Il faut gérer le mieux possible les erreurs sur le flux.
- ✓ Le prototype d'une surcharge pourra donc être par exemple :

```
ostream & operator << (ostream&, expression_type_classe)
istream  &  operator >> (istream&, & type_classe)
```

8.2.8 Formatage des données

- ✓ On a vu que pour chaque type de base il existe un formatage par défaut.
- ✓ Grâce à des symboles, appelés **manipulateurs**, il est possible de sélectionner un autre formatage.
- ✓ Ces manipulateurs renvoient un flot, dont le fonctionnement est modifié par l'application du manipulateur.

- ✓ Deux types de manipulateurs existent, en fonction de la présence ou non de paramètres. Nous donnons dans un premier temps les prototypes de ces opérateurs, nous verrons ensuite la liste des manipulateurs et leur action :

```
istream & nom_manipulateur()
ostream & nom_manipulateur()
istream & nom_manipulateur(argument)
ostream & nom_manipulateur(argument)
```

- ✓ Les manipulateurs non paramétriques sont :
  - o `dec` E/S numérotation décimale
  - o `hex` E/S numérotation hexadécimale
  - o `oct` E/S numérotation octale
  - o `endl` S saut de ligne, vide le tampon
  - o `ends` S fin de chaine (end string)
  - o `flush` S vide le tampon
  - o `ws` S ignore les espaces (white space)
- ✓ Les manipulateurs paramétriques sont :
  - o `setbase(int)` E/S définit la base de conversion
  - o `resetiosflags(long)` E/S remet à zéro les bits désignés par l'argument
  - o `setiosflags(long)` E/S active les bits désignés par l'argument
  - o `setfill(int)` E/S fixe le caractère de remplissage
  - o `setprecision(int)` E/S précision des nombres flottants
  - o `setw(int)` E/S largeur d'affichage
- ✓ Voici un exemple d'utilisation :

```
cout << " 100 en décimal : " << dec << 100;
cout << " 100 en octal : " << oct << 100;
cout << " 100 en héxadécimal : " << hex << 100;
const double PI = 3.14159265358979323;
cout  << "PI à 4 décimales prés : " << setw(4) << PI;
```

8.3 Les locales

- ✓ On trouve ici des classes pour la prise en compte des particularités de chaque partie du monde. Cela concerne l'alphabet utilisé, l'expression de la date et l'heure, les unités monétaires, les autres mesures, etc.
- ✓ Pour donner une idée du contenu du fichier d'en-tête **<locale>** voici une liste simplifiée des classes qu'il introduit :

```
// représentation des particularités locales :
class locale;

// classification des caractères (fonctions « de confort »)
template<class charT> bool isspace (charT c, const locale &loc);
template<class charT> bool isprint (charT c, const locale &loc);
template<class charT> bool iscntrl (charT c, const locale &loc);
template<class charT> bool isupper (charT c, const locale &loc);
template<class charT> bool islower (charT c, const locale &loc);
template<class charT> bool isalpha (charT c, const locale &loc);
template<class charT> bool isdigit (charT c, const locale &loc);
template<class charT> bool ispunct (charT c, const locale &loc);
template<class charT> bool isxdigit(charT c, const locale &loc);
template<class charT> bool isalnum (charT c, const locale &loc);
template<class charT> bool isgraph (charT c, const locale &loc);
template<class charT> charT toupper (charT c, const locale &loc);
```

```
template<class charT> charT tolower (charT c, const locale &loc);

// classification des caractères
class ctype_base;
template<class charT> class ctype;
template<> class ctype<char>;
template<class charT> class ctype_byname;
template<> class ctype_byname<char>;

// nombres
template<class charT, class inIter> class num_get;
template<class charT, class OutputIterator> class num_put;
template<class charT> class numpunct;
template<class charT> class numpunct_byname;

// comparaison et « hashing »
template<class charT> class collate;
template<class charT> class collate_byname;

// date et heure
class time_base;
template<class charT, class inIter> class time_get;
template<class charT, class inIter> class time_get_byname;
template<class charT, class OutputIterator> class time_put;
template<class charT, class OutputIterator> class time_put_byname;

// symboles et nombres monetaires
class money_base;
template<class charT, class inIter> class money_get;
template<class charT, class OutputIterator> class money_put;
template<class charT, bool Intl> class moneypunct;
template<class charT, bool Intl> class moneypunct_byname;

// obtention de chaînes depuis des catalogues de messages
class messages_base;
template<class charT> class messages;
template<class charT> class messages_byname;
etc.
```

8.4 Les conteneurs

8.4.1 Introduction

- ✓ Les conteneurs sont des objets génériques représentant des collections d'objets.
- ✓ Le type des objets est un paramètre générique du conteneur.
- ✓ Les fichiers d'entête concernés sont :

```
<deque>  <list> <queue> <stack>
<vector> <map>  <set>   <bitset>
```

- ✓ On peut séparer les conteneurs en 3 familles :

  - o **Les séquences**, qui collectionnent des objets de même type en séquence. Il y a 3 conteneurs de séquences : les **listes** (`list`) les **vecteurs** (`vector`) et les **listes à double entrée** (`deque`).
    En association avec les séquences, la bibliothèque fournit des adaptateurs, qui sont des interfaces permettant d'utiliser les séquences pour implémenter des types de données abstraits : les **piles** (`stack`), les **files** (`queue`) et les **files de priorité** (`priority_queue`).

  - o **Les conteneurs associatifs**, qui collectionnent des objets de même type, chacun associé à une clé permettant d'y avoir accès. Il n'y a donc pas de notion de niéme élément, car il n'y a pas de relation d'ordre. Il y a deux types de conteneurs associatifs : ceux où la clé et l'objet ne font qu'un, les **ensembles** (`set`) et les **multi-ensembles** (`multiset`). Au contraire, dans les **tables associatives** (`map`) et les **tables associatives multiples** (`multimap`) la clé et la valeur associée sont séparées et peuvent être de types différents.

  - o **Les conteneurs de bits** qui sont pris en charge par deux types particuliers :les **vecteurs de bits** (`vector<bool>`) et les **ensembles de bits** (`bitset`).

8.4.2 Constructions, copies, destructions des éléments d'un conteneur

- ✓ Les objets sont littéralement possédés par le conteneur. Les conséquences sont :
  - o Lorsque le conteneur est détruit les objets contenus sont aussi détruits. Si ces objets sont de type pointeur, les objets pointés ne sont donc pas détruits, et peut être même inaccessible, auquel cas il y a une grave fuite de mémoire. Pour y remédier on peut faire des conteneurs d'auto pointeurs. En effet un auto pointeur détruit l'objet pointé lorsqu'il est détruit.
  - o Si il y a une copie du conteneur, les objets contenus sont aussi copiés. En général ce n'est pas ce que le programmeur veut, il faut donc être vigilant. De plus cela implique que les objets contenus supportent le constructeur de copie.
  - o Les objets contenus doivent aussi supporter l'opérateur d'affectation, qui est utilisé lors de la modification d'un élément de la collection.
- ✓ Voici un exemple :

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

```
struct CEntier
{
    int m_i;
    CEntier(int i = 0) : m_i(i)
    { cout << "CEntier(" << m_i <<") [" << this << "]\n"; }
    CEntier(const CEntier &p) : m_i(p.m_i)
    { cout << "CEntier(CEntier &) [" << this << " <-- "<< (void *)&p << "]\n"; }
    ~CEntier()
    { cout << "~CEntier() [" << this << "]\n"; }
};
void Display(vector<CEntier> v)
{
    for (int i = 0; i < v.size(); i++)
           cout << '(' << v[i].m_i << ") ";
    cout << '\n';
}
void main()
{
    cout << "Debut du programme\n";
    CEntier a(1), b(2), c(3);
    cout << "Creation du vecteur\n";
    vector<CEntier> *v = new vector<CEntier>;
    v->push_back(a);
    v->push_back(b);
    v->push_back(c);
    cout << "Appel de la fonction Display(vecteur)\n";
    Display(*v);
    cout << "Destruction du vecteur\n";
    delete v;v=0;
    cout << "Fin du programme\n";
}
```

✓ Affichage obtenu :

```
Debut du programme
CEntier(1) [0012FF70]
CEntier(2) [0012FF6C]
CEntier(3) [0012FF68]
Creation du vecteur
CEntier(CEntier &) [004918A0 <-- 0012FF70]
CEntier(CEntier &) [004918A4 <-- 0012FF6C]
CEntier(CEntier &) [004918A8 <-- 0012FF68]
Appel de la fonction Display(vecteur)
CEntier(CEntier &) [00491910 <-- 004918A0]
CEntier(CEntier &) [00491914 <-- 004918A4]
CEntier(CEntier &) [00491918 <-- 004918A8]
(1) (2) (3)
~CEntier() [00491910]
~CEntier() [00491914]
~CEntier() [00491918]
Destruction du vecteur
~CEntier() [004918A0]
~CEntier() [004918A4]
~CEntier() [004918A8]
Fin du programme
~CEntier() [0012FF68]
~CEntier() [0012FF6C]
~CEntier() [0012FF70]
```

8.4.3 Les méthodes communes aux conteneurs

✓ La plupart des conteneurs implémentent les méthodes suivantes :

- `bool` **`empty`**`() const;` renvoie `true` si le conteneur n'a pas d'élément.
- `size_t` **`size`**`() const;` renvoie le nombre d'élément dans le conteneur. Notez que `size()` et `empty()` sont implémentées par tous les conteneurs et retournent en un temps constant.
- `void` **`resize`**`(size_t nouvelle_taille, T valeur);` Changement de la taille du conteneur. Dans le cas d'un agrandissement, la `valeur` indiquée

(par défaut : la valeur par défaut du type `T`) est utilisée pour garnir les nouveaux emplacements.

- `size_t` **`capacity`**`() const;` nombre d'éléments que le conteneur peut maintenir sans nécessiter une réallocation.
- `void` **`reserve`**`(size_t n);` modifie la taille de l'espace réservé pour le conteneur (celui dont la taille est donnée par capacity).
- `size_t` **`max_size`**`() const;` nombre maximum d'éléments que le conteneur peut contenir.
- `T &`**`operator[]`**`( clé );const T &operator[]( clé ) const;`
  `T &`**`at`**`( clé );const T &at( clé ) const;`
  Accès indexé (dans le cas des tables associatives (`map`), `clé` est du type précisé lors de l'instanciation du modèle ; dans les autres cas, `clé` d'un type entier).
- `T &`**`front`**`(); const T &`**`front`**`() const;` Accès à l'élément qui est en tête.
- `T &`**`back`**`();const T &`**`back`**`() const;` Accès à l'élément qui est en queue.
- `void` **`push_front`**`(const T &);` `void` **`push_back`**`(const T &);`
  Ajout d'un élément en tête [resp. en queue].
- `void` **`pop_front`**`();` `void` **`pop_back`**`();` Suppression de l'élément en tête [resp. en queue].

✓ Voici un exemple d'utilisation de ces méthodes :

```
#include <iostream>
#include <vector>
using namespace std;
void main()
{
    vector<int> v;
    for (int i = 0; i < 18; i++){
            cout << "size: " << v.size() << " -- capacity: " << v.capacity()
            << " -- max: " << v.max_size() << "\n";
            v.push_back(i);}
}
```

✓ Affichage obtenu :

```
size: 0 -- capacity: 0 -- max: 1073741823
size: 1 -- capacity: 1 -- max: 1073741823
size: 2 -- capacity: 2 -- max: 1073741823
size: 3 -- capacity: 4 -- max: 1073741823
size: 4 -- capacity: 4 -- max: 1073741823
size: 5 -- capacity: 8 -- max: 1073741823
size: 6 -- capacity: 8 -- max: 1073741823
size: 7 -- capacity: 8 -- max: 1073741823
size: 8 -- capacity: 8 -- max: 1073741823
size: 9 -- capacity: 16 -- max: 1073741823
size: 10 -- capacity: 16 -- max: 1073741823
size: 11 -- capacity: 16 -- max: 1073741823
size: 12 -- capacity: 16 -- max: 1073741823
size: 13 -- capacity: 16 -- max: 1073741823
size: 14 -- capacity: 16 -- max: 1073741823
size: 15 -- capacity: 16 -- max: 1073741823
size: 16 -- capacity: 16 -- max: 1073741823
size: 17 -- capacity: 32 -- max: 1073741823
```

8.4.4 Les itérateurs

✓ Les itérateurs sont des objets spécialement conçus pour faciliter l'accès et les opérations sur les éléments d'un conteneur.

✓ Voici les méthodes courantes implémentées par pratiquement tous les conteneurs pour obtenir des itérateurs :

- `itérateur` **`insert`**`( itérateur , const T& x);`
  `void` **`insert`**`( itérateur , size_t n, const T& x);`
  Insertion d'une valeur **`x`** [resp. de **`n`** copies d'une valeur **`x`**].
- `iterator` **`erase`**`( itérateur );` `void` **`clear`**`();`
  Suppression d'une valeur [resp. de toutes les valeurs].
- `itérateur` **`begin`**`();` `const itérateur` **`begin`**`() const;`
  `itérateur` **`end`**`();` `const itérateur` **`end`**`() const;`
  Itérateur positionné sur le premier [resp. le dernier] élément.
- `itérateur` **`rbegin`**`();` `const itérateur` **`rbegin`**`() const;`
  `itérateur` **`rend`**`();` `const itérateur` **`rend`**`() const;`
  Itérateur inverse positionné sur le premier [resp. le dernier] élément.
- `itérateur void` **`swap`**`(vector<T> &v);`
  Echange des valeurs des vecteurs **`*this`** et **`v`**.

✓ Lorsqu'un itérateur est valide, c'est à dire qu'il est placé sur un élément d'un conteneur, on peut accéder à l'élément avec les opérateurs `*` et `->`.

✓ L'opérateur ++ préfixé (++i) et postfixé (i++) est défini pour les itérateurs, et place l'itérateur sur l'élément suivant.

✓ Les opérateurs d'inégalité != et d'égalité == sont définis, et permettent de vérifier ou non si deux itérateurs sont placés sur le même élément d'un même conteneur.

✓ Voici un exemple classique de parcours d'un conteneur :

```
#include <list>
using namespace std;
void main()
{
        list<int> l;
        for (int i = 1; i < 10; i++) l.push_back(i);
        int sum =0;
        list<int>::iterator it;
        for( it = l.begin() ; it != l.end() ; it++ ) sum+= *it;
        // sum = 1+2+...+9 = 45
}
```

✓ On aurait pu calculer la même somme en parcourant la liste dans le sens inverse :

```
list<int>::reverse_iterator rit;
for( rit = l.rbegin() ; rit != l.rend() ; rit++ ) sum+= *rit;
// sum = 9+8+...+1 = 45
```

✓ Beaucoup de détails relatifs aux itérateurs ne sont pas exposés ici. Néanmoins on reviendra sur les itérateurs dans le paragraphe consacré aux algorithmes (8.4.9).

8.4.5 Les conteneurs séquences concrets: list , vector, deque

✓ Voici un rapide comparatif de ces 3 types :

	`vector`	`list`	`Deque` (files à double entrées)
Accés direct aux éléments (a[78]…)	Oui en temps constant	non	Oui en temps constant
Insertion/suppression au début	Temps linéaire	Temps constant	Temps constant
Insertion/suppression à la fin	Temps constant	Temps constant	Temps constant
Insertion/suppression entre 2 éléments	Temps linéaire	Temps constant	Temps linéaire

✓ On voit que les listes doivent être utilisées si il y a de nombreuses insertions/suppressions partout dans la séquence.

✓ On voit que les vecteurs et les files à doubles entrées doivent être utilisés lorsque l'on à besoin d'accès direct aux éléments par leurs positions dans la séquence (vect[54]…), aussi appelé accès aléatoire.

- ✓ On voit que l'avantage des files à double entrées sur les vecteurs est l'insertion/suppression au début en un temps constant.
- ✓ En revanche on ne voit pas que l'avantage des vecteurs sur les files à double entrées est la présence des méthodes `capacity()` et `reserve()` sur le vecteur. De plus une insertion dans une telle file invalide les itérateurs couramment définis.
- ✓ Précisons que les listes possèdent des méthodes très spécifiques :

```
void splice(iterator pos, list<T>& x);
void splice(iterator pos, list<T>& x, iterator i);
void splice(iterator pos, list<T>& x, iterator premier,iterator dernier);
```

Insère, devant la position indiquée, des éléments extraits, et supprimés, de la liste x : tous les éléments (premier cas), l'élément à la position indiquée (second cas), les éléments de l'intervalle indiqué (troisième cas).

```
void remove(const T& valeur);
template <class PredicatUn> void remove_if(PredicatUn predicat);
```

Supprime les éléments égaux à la valeur indiquée (premier cas) ou les éléments sur lesquels la valeur du prédicat unaire indiqué n'est pas `false`.

```
void unique();
template <class PredicatBin> void unique(PredicatBin pred);
```

Chaque groupe d'éléments consécutifs *`la, la+1, … lb`* vérifiant *`li = li+1`* (premier cas) ou `pred(li, li+1)` (deuxième cas) est remplacé par *la*, le premier des éléments du groupe.

```
void sort();
template <class Compare> void sort(Compare comp);
void merge(list<T,Allocator>& x);
template <class Compare> void merge(list<T,Allocator>& x, Compare comp);
```

Tri de liste (`sort`) et fusion de listes ordonnées (`merge`). Dans le premier cas il est supposé qu'une relation d'ordre est définie sur les éléments des listes. Dans le deuxième cas, la relation est fournie (c'est `Compare`).

```
void reverse();
```

Inversion des éléments d'une liste.

8.4.6 Les conteneurs séquences abstraits: stack, queue, priority_queue

- ✓ On les appelle aussi les adaptateurs.
- ✓ L'idée est simple : on utilise un conteneur séquence concret, et on modifie l'opération d'extraction de façon à ce que :
  - o L'élément extrait d'une pile (`stack`) soit l'élément le plus récemment inséré.
  - o L'élément extrait d'une `queue` soit l'élément le moins récemment inséré.
  - o L'élément extrait d'une file de priorité (`priority_queue`) soit l'élément le plus prioritaire, la priorité étant déterminée par une opération `Compare()`.
- ✓ Voici les contraintes imposées aux conteneurs séquentiels, et les choix courants :

	`stack`	`queue`	`priority_queue`
Méthodes que doit implémenter le conteneur séquence	`push()` `top()` `pop()`	`front()` `back()` `push_back()` `pop_front()`	`front()` `push_back()` `pop_back()`
Conteneurs séquences possible	`vector` `deque` (par défaut)	`deque` (toujours meilleur choix)	`vector` (par défaut) `deque`
Insertion/suppression à la fin	Temps constant	Temps constant	Temps constant
Insertion/suppression entre 2 éléments	Temps linéaire	Temps constant	Temps linéaire

- ✓ Voici un exemple de l'utilisation d'une file de priorité. Les éléments sont entiers mais l'ordre imposé est spécial : les paires sont toujours prioritaires sur les impairs, et l'ordre naturel des entiers reste entre paires ou entre impaires :

```
#include <iostream>
#include <queue>
using namespace std;
struct moindre : public binary_function<int, int, bool> // utilisation d'un foncteur
{
        bool operator()(int a, int b)
        {
        if (a % 2 == 0)
                return b % 2 != 0 ? false : a < b;
        else
                return b % 2 == 0 ? true  : a < b;
        }
};
void main()
{
        int x, i;
        priority_queue<int, vector<int>, moindre> p;
        for (i = 0; i < 16; i++)
        {
                cout << (x = rand() % 100) << ' ';
                p.push(x);
        }
        cout << '\n';
        while ( ! p.empty())
        {
                cout << p.top() << ' ';
                p.pop();
        }
        cout << '\n';
}
```

- ✓ Affichage obtenu :

```
41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91
78 64 62 58 34 24 0 91 81 69 67 61 45 41 27 5
```

- ✓ Attention cependant à ne pas voir une file de priorité comme une méthode de tri. En effet si le conteneur séquence sous jacent à une méthode d'accès aléatoire en temps constant (comme `vector`) alors les opérations `push()` et `pop()` demandent un temps logarithmique.

8.4.7 Les conteneurs associatifs : set, map

- ✓ On a vu que les conteneurs associatifs associent une clé à chacun de leurs éléments, qu'ils sont génériques, paramétrés par le type des clés et le type des éléments.
- ✓ De plus il existe une opération `bool Compare(clé,clé)` totalement définie sur l'ensemble des clés possibles.
- ✓ `Compare()` doit induire un *ordre strict faible* sur l'ensemble des clés possibles c'est à dire :
  - o `Compare()` n'est pas réflexive : $\forall$`x` $\neg$`Compare(x,x)`
  - o `Compare()` est transitive :
    $\forall$ `x,y,z  Compare(x,y)^Compare(y,z)` $\Rightarrow$ `Compare(x,z)`
  - o Soit la relation `Equivalent(x,y)` définie par :

    $\forall$ `x,y Equivalent(x,y)` $\Leftrightarrow$ $\neg$`Compare(x,y)` $\wedge$ $\neg$`Compare(y,x)`
  - o `Equivalent()` est une relation d'équivalence (réflexive, transitive, symétrique).

- ✓ Tout ça pour montrer que 2 clés peuvent être différentes et cependant équivalentes. Bien entendu une clé est aussi équivalente à elle-même. Cette idée est a l'origine de la différence entre `set` et `multiset` et `map` et `multimap`.
- ✓ Dans les conteneurs `set` et `map` il ne peut y avoir 2 éléments associés à deux clés équivalentes, à l'inverse des `multimap` et des `multiset`. Voyons tout ceci en exemple :
- ✓ `set` et `multiset` :

```
#include <iostream>
#include <set>
using namespace std;
void main()
{
	int x, i;
	set<int> s;
	multiset<int> ms;
	cout<<"Les éléments pris au hasard:\n";
	for (i = 0; i < 16; i++)
	{
		cout << (x = rand() % 10) << ' ';
		s.insert(x);
		ms.insert(x);
	}
	cout << "\nLe set:\n";
	set<int>::iterator i0, i1;
	for (i0 = s.begin(), i1 = s.end(); i0 != i1; i0++)
		cout << *i0 << ' ';
	cout << "\nLe multiset:\n";
	multiset<int>::iterator mi0, mi1;
	for (mi0 = ms.begin(), mi1 = ms.end(); mi0 != mi1; mi0++)
		cout << *mi0 << ' ';
	cout << '\n';
}
```

- ✓ Affichage obtenu :

```
Les éléments pris au hasard:
1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1
Le set:
0 1 2 4 5 7 8 9
Le multiset:
0 1 1 1 1 2 4 4 4 5 5 7 7 8 8 9
```

- ✓ `map` :

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
template<class A, class B>
ostream &operator<<(ostream &o, pair<A, B> &p)
    { return o << '<' << p.first << ',' << p.second << '>'; }
void main()
{
    map<string, int> m;
    m["Patrick"]   = 175;
    m["Sebastien"]  = 186;
    m["Olivier"]    = 193;
    m["Patrick"]    = 180;
    m["Olivier"]    = 170;
    cout << m.size() << '\n' << m["Patrick"] << ' '
    << m["Olivier"] << ' ' << m["Sebastien"] << '\n';
    map<string, int>::iterator courant = m.begin();
    map<string, int>::iterator dernier = m.end();
    while (courant != dernier)
        cout << *courant++ << ' ';
    cout << '\n';
}
```

- ✓ Affichage obtenu :

```
3
180 170 186
<Olivier,170> <Patrick,180> <Sebastien,186>
```

- ✓ `multimap` :

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
template<class A, class B>
ostream &operator<<(ostream &o, pair<A, B> &p)
        { return o << '<' << p.first << ',' << p.second << '>'; }
void main()
{
        multimap<string, int> m;
        m.insert(make_pair(string("Patrick" ), 175));
        // on ne peut écrire:   m["Patrick"]  = 175;
        m.insert(make_pair(string("Sebastien"),186));
        m.insert(make_pair(string("Olivier" ), 193));
        m.insert(make_pair(string("Patrick" ), 180));
        m.insert(make_pair(string("Olivier" ), 170));
        cout << m.size() << '\n';
        // on ne peut écrire:   cout << m["Patrick"];
        multimap<string, int>::iterator p = m.lower_bound("Olivier");
        multimap<string, int>::iterator q = m.upper_bound("Olivier");
        while (p != q)  cout << *p++ << ' ';
        cout << '\n';
}
```

- ✓ Affichage obtenu :

```
5
<Olivier,193> <Olivier,170>
```

8.4.8 Les conteneurs de bits : vector<bool>, bitset

- ✓ L'idée sous jacente de conteneurs de bits est d'optimiser la place mémoire lorsque l'on traite des bits. Concrètement pour stocker N bits l'emplacement mémoire ne doit pas dépasser N/8+1 octets.
- ✓ Les vecteurs de bits s'utilisent comme les vecteurs normaux.
- ✓ La principale différence est que la taille des vecteurs de bits peut être précisée dynamiquement. Cependant les `bitset` sont plus adaptés à la manipulation des opérations usuelles sur les booléens.
- ✓ Un `bitset` est un tableau de bits rangés de manière compacte et principalement destiné à tirer profit des opérations logiques bit à bit disponibles sur la machine utilisée.
- ✓ La classe `bitset` est générique de paramètre la taille du tableau de bits.
- ✓ Pour instancier `bitset` il faut inclure le fichier `<bitset>` qui définit cette classe de la manière suivante :

```
template<size_t N> class bitset
{
  public:
     class reference
     {
       friend class bitset;
       public:
        reference& operator=(bool x);              // pour b[i] = x;
        reference& operator=(const reference&);    // pour b[i] = b[j];
        bool operator~() const;                    // bascule d'un bit
        operator bool() const;                     // pour x = b[i];
        reference& flip();                         // pour b[i].flip();
     };
```

```
        // construction
        bitset();
        bitset(unsigned long val);
        explicit bitset(string s, size_t pos, size_t nbr)

        // opérations
        bitset<N>& operator&=(const bitset<N>& rhs);
        bitset<N>& operator|=(const bitset<N>& rhs);
        bitset<N>& operator^=(const bitset<N>& rhs);
        bitset<N>& operator<<=(size_t pos);
        bitset<N>& operator>>=(size_t pos);
        bitset<N> operator<<(size_t pos) const;
        bitset<N> operator>>(size_t pos) const;

        // activation (set) et désactivation (reset)
        bitset<N>& set();
        bitset<N>& set(size_t pos, int val = true);
        bitset<N>& reset();
        bitset<N>& reset(size_t pos);

        // bascule
        bitset<N> operator~() const;
        bitset<N>& flip();
        bitset<N>& flip(size_t pos);

        // accès et conversion
        reference operator[](size_t pos); // accès à un bit
        unsigned long to_ulong() const;
        string to_string() const; // la string retourné est du type "1100110"

        // tests
        bool operator==(const bitset<N>& rhs) const; // égalité entre 2 bitset
        bool operator!=(const bitset<N>& rhs) const; // différence entre 2 bitset
        bool test(size_t pos) const;                 // retourne true si LE bit est 1
        bool any() const;             // retourne true si au moins UN bit est 1
        bool none() const;            // retourne true si tous les bits sont à 0
        size_t size() const ;// retourne la taille du tableau (l'argument générique)
        size_t count() const;// retourne le nombre de bit à 1
};
```

8.4.9 Les algorithmes

- ✓ Plusieurs algorithmes sur les éléments des conteneurs sont fournis par la STL.
- ✓ On peut les diviser en 3 catégories :
  - Les algorithmes de consultation d'une séquence (exemple : recherche d'un élément, nombre d'éléments satisfaisant un prédicat…)
  - Les algorithmes de modification d'une séquence (exemple : copie d'éléments…)
  - Les algorithmes spécialisés dans les séquences ordonnées (exemple : tri, nième élément…)
- ✓ Voici quelques remarques sur les algorithmes de la STL :
  - La plupart sont si simples qu'ils sont implémentés avec une fonction in line, d'où une grande efficacité.
  - Souvent, on représente une séquence par 2 itérateurs : début fin. Début est sur le premier élément, et fin est placé sur un élément fictif qui se trouve après le dernier élément. Ceci est dû au fait que dans une boucle, pour parcourir tous les éléments la condition est du type : tant que l'itérateur est différent de l'élément après le dernier élément.
  - Lorsqu'un algorithme renvoie un itérateur placé sur cet élément fictif, cela signifie que l'opération a échoué. Par exemple, la recherche d'une valeur **x** dans une liste **L** s'écrit :

```
list<int> L;
...
list<int>::iterator r = find(L.begin(), L.end(), x);
if (r != L.end())
//la valeur x apparaît dans la liste à la position r
else
//la valeur x est absente de la liste
```

- Lorsqu'un algorithme a besoin d'un fonction il utilise un foncteur.

✓ Nous ne présentons ici seulement quelques algorithmes représentatifs, pour une description complète voir **[Stroustrup]** :

---

algorithmes de consultation d'une séquence

for_each

```
template<class Iter, class Fonc>
fonc for_each(Iter debut, Iter fin, Fonc f);
```

Action : appel de f sur chaque élément de la séquence [début, fin [.
Retour : f

find

```
template<class IterIn, class T>
Iter find(Iter debut, Iter fin, const T& valeur);
```

Action : recherche le premier élément de la séquence [`debut, fin` [égal à la `valeur` indiquée.
Retour : un itérateur positionné sur l'élément en question, s'il existe ; la valeur `fin` sinon.
Contrainte : la relation égalité doit être définie sur le type `T`.

find_if

```
template<class Iter, class PredUn>
Iter find_if(Iter first, Iter last, PredUn pred);
```

Action : recherche le premier élément *x* de la séquence [`debut, fin` [tel que `pred`(*x*) ≠ `false`.
Retour : un itérateur positionné sur l'élément en question, s'il existe ; la valeur `fin` sinon.

count

```
template<class Iter, class T>
typename iterator_traits<Iter>::difference_type
count(Iter debut, Iter fin, const T& valeur);
```

Action : comptage du nombre d'éléments de la séquence [`debut, fin` [égaux à la `valeur` indiquée.
Retour : le nombre de tels éléments.
Contrainte : la relation égalité doit être définie sur le type `T`.

count_if

```
template<class Iter, class PredUn>
typename iterator_traits<Iter>::difference_type
count_if(Iter debut, Iter fin, PredUn pred);
```

Action : comptage du nombre d'éléments *x* de la séquence [`debut, fin` [pour lesquels `pred(x) ≠ false`.
Retour : le nombre de tels éléments.

equal
```
template<class Iter1, class Iter2>
bool equal(Iter1 debut1, Iter1 fin1, Iter2 debut2);
```
Retour : la réponse à la question : les séquences [**debut1, fin1** [et [**debut2, fin2** [sont-elles égales ?

Les algorithmes non présentés sont nommés:
find_end , find_first_of, adjacent_find, mismatch, search, search_n

---

algorithmes de modification d'une séquence

copy
```
template<class Iter, class Iter>
Iter copy(Iter debut, Iter fin, Iter result);
```
Action : copie, *en avançant*, des éléments de la séquence [`debut, fin` [sur les éléments de la séquence
[`result, result + fin - debut` [. Equivaut à :
pour *i* allant de 0 à `fin - debut - 1`, faire `*(result + i) = *(debut + i)`
Retour : la valeur `result + fin - debut`.
Contrainte : `result` ne doit pas appartenir à l'intervalle [`debut, fin` [.

swap
```
template<class T> void swap(T& a, T& b);
```
Action : échange les éléments `a` et `b`.
Contrainte : le type `T` doit supporter l'affectation

iter_swap
```
template<class Iter1, class Iter2>
void iter_swap(Iter1 a, Iter2 b);
```
Action : échange les éléments `*a` et `*b`.

replace
```
template<class Iter, class T>
void replace(Iter debut, Iter fin,
const T& oldVal, const T& newVal);
```
Action : toutes les occurrences de `oldVal` dans la séquence [`debut, fin` [sont remplacées par `newVal`.
Contrainte : le type `T` doit supporter l'égalité et l'affectation.

replace_if
```
template<class Iter, class PredUn, class T>
void replace_if(Iter debut, Iter fin,
PredUn pred, const T& newVal);
```
Action : tout *x* de la séquence [`debut, fin` [qui vérifie `pred(x) ≠ false` est remplacé par `newVal`.
Contrainte : le type `T` doit supporter l'affectation.

fill
```
template<class Iter, class T>
void fill(Iter debut, Iter fin, const T& valeur);
```
Action : tous les éléments de la séquence [`debut, fin` [sont affectés de la `valeur` indiquée.

Contrainte : le type `T` doit supporter l'affectation

generate

```
template<class Iter, class Generateur>
void generate(Iter debut, Iter fin, Generateur gen);
```

Action : tous les éléments de la séquence [`debut, fin` [sont affectés de la valeur `gen()`.

Contrainte : `Generateur` est un type objet fonction sans argument.

remove

```
template<class Iter, class T>
Iter remove(Iter debut, Iter fin, const T& valeur);
```

Action : suppression des éléments de la séquence [`debut, fin` [qui sont égaux à la `valeur` indiquée.

Retour : la fin de la séquence résultante.

Contrainte : le type `T` supporte l'égalité.

unique

```
template<class Iter>
Iter unique(Iter debut, Iter fin);
```

Action : remplacement de chaque sous-séquence faite d'éléments (consécutifs) égaux par son premier élément.

Retour : la fin de la séquence résultante.

reverse

```
template<class IterBid>
void reverse(IterBid debut, IterBid fin);
```

Action : renversement de la séquence indiquée. Equivaut à :
pour *i* allant de 0 à (`fin - debut`) / 2 - 1 faire `swap(*(debut + i), *(fin - i - 1))`

random_shuffle

```
template<class IterAlea>
void random_shuffle(IterAlea debut, IterAlea fin);
```

Action : arrangement aléatoire des éléments de la séquence [`debut, fin` [, selon une distribution uniforme
(c'est-à-dire que chacune des `(fin - debut)!` permutations possibles a autant de chances d'être employée).

partition

```
template<class IterBid, class PredUn>
IterBid partition(IterBid debut, IterBid fin, PredUn pred);
```

Action : réarrange les éléments de la séquence [`debut, fin` [de telle manière que tous ceux pour lesquels la condition exprimée par `pred` est vraie se trouvent devant tous ceux pour lesquels cette condition est fausse.

Retour : une valeur d'itérateur *r* telle que pour toute valeur `debut` $\leq i < r$ on a `pred(*i) != false` et pour toute
valeur $r \leq j <$ `fin` on a `pred(*j) == false`.

Les algorithmes non présentés sont nommés:
copy_backward, swap_ranges, transform, replace_copy, replace_copy_if, fill_n, generate_n, remove_if, remove_copy, remove_copy_if, unique_copy, reverse_copy, rotate, rotate_copy stable_partition

---

algorithmes spécialisés dans les séquences ordonnées

sort
```
template<class IterAlea>
void sort(IterAlea debut, IterAlea fin);
template<class IterAlea, class Compare>
void sort(IterAlea debut, IterAlea fin, Compare comp);
```
Action : tri des éléments de la séquence [`debut, fin` [

nth_element
```
template<class IterAlea >
void nth_element(IterAlea debut, IterAlea nth, IterAlea fin);
template<class IterAlea, class Compare>
void nth_element(IterAlea debut, IterAlea nth, IterAlea fin,
Compare comp);
```
Action : après l'exécution de cet algorithme, l'élément à la position `nth` est celui qui se trouverait à cet endroit si la séquence [`debut, fin` [était triée. De plus, aucun élément de [`nth, fin` [n'est inférieur à aucun élément
de [`debut, nth` [.

lower_bound
```
template<class Iter, class T>
Iter lower_bound(Iter debut, Iter fin, const T& valeur);
template<class Iter, class T, class Compare>
Iter lower_bound Iter debut, Iter fin, const T& valeur,
Compare comp);
```
Retour : un itérateur pointant la position la plus à gauche, dans la séquence triée [`debut, fin` [, à laquelle on peut placer la `valeur` indiquée sans violer l'ordre.
Contrainte : la séquence [`debut, fin` [doit être triée.

merge
```
template<class Iter1, class Iter2, class Iter>
Iter merge(Iter1 debut1, Iter1 fin1,
Iter2 debut2, Iter2 fin2, Iter result);
template<class Iter1, class Iter2, class Iter, class Compare>
Iter merge(Iter1 debut1, Iter1 fin1,
Iter2 debut2, Iter2 fin2, Iter result, Compare comp);
```
Action : fusion des deux séquences triées [`debut1, fin1` [et [`debut2, fin2` [en une unique séquence triée
[`result, result + (fin1 - debut1) + (fin2 - debut2)` [.
Retour : la valeur `result + (fin1 - debut1) + (fin2 - debut2)`
Contrainte : les deux séquences [`debut1, fin1` [et [`debut2, fin2` [doivent être triées.

includes
```
template<class Iter1, class Iter2>
bool includes(Iter1 debut1, Iter1 fin1,
Iter2 debut2, Iter2 fin2);
```

```
template<class Iter1, class Iter2, class Compare>
bool includes(Iter1 debut1, Iter1 fin1,
Iter2 debut2, Iter2 fin2, Compare comp);
```
Retour : la réponse à la question « les éléments de la séquence [`debut1, fin1` [apparaissent-ils tous dans la séquence [`debut2, fin2` [? »
Contrainte : les deux séquences [`debut1, fin1` [et [`debut2, fin2` [doivent être triées.

`set_union`
```
template<class Iter1, class Iter2, class Iter>
Iter set_union(Iter1 debut1, Iter1 fin1,
Iter2 debut2, Iter2 fin2, Iter result);
template<class Iter1, class Iter2, class Iter, class Compare>
Iter set_union(Iter1 debut1, Iter1 fin1,
Iter2 debut2, Iter2 fin2, Iter result, Compare comp);
```
Action : construit la réunion des séquences triées [`debut1, fin1` [et [`debut2, fin2` [, sous la forme d'une séquence triée rangée à partir de la position `result`.
Retour : la fin de la séquence résultante.
Contrainte : les séquences [`debut1, fin1` [et [`debut2, fin2` [doivent être triées.

`min`
```
template<class T>
const T& min(const T& a, const T& b);
template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
```
Retour : le plus petit des deux éléments (ou, s'ils sont équivalents, le premier).

`min_element`
```
template<class Iter>
Iter min_element(Iter debut, Iter fin);
template<class Iter, class Compare>
Iter min_element(Iter debut, Iter fin, Compare comp);
```
Retour : l'itérateur *i* le plus à gauche tel qu'aucun élément de la séquence ne soit strictement inférieur à `*`*i*.

Les algorithmes non présentés sont nommés:
`stable_sort , partial_sort, partial_sort_copy, upper_bound, equal_range, binary_search, inplace_merge, set_intersection set_difference, .set_symmetric_difference, push_heap, pop_heap, make_heap, sort_heap, max, max_element, lexicographical_compare, next_permutation, prev_permutation`

9 Les mots clés du C++

Les identifiants suivants sont considérés comme des mots-clés du langage C++. Ils ne doivent pas être utilisés pour un autre usage :

asm	**float**	**signed**
auto	**for**	**sizeof**
break	**friend**	**static**
case	**goto**	**struct**
catch	**if**	**switch**
char	**inline**	**template**
class	**int**	**this**
const	**long**	**throw**
continue	**new**	**try**
default	**operator**	**typedef**
delete	**private**	**union**
do	**protected**	**unsigned**
double	**public**	**virtual**
else	**register**	**void**
enum	**return**	**volatile**

10 Bibliographie

[Stroustrup] The C++ programming language 3eme Edition
ADDISON-WESLEY
Dr. Bjarne Stroustroup
ISBN: 0-201-88954-4
Voir aussi http://www.research.att.com/%7Ebs/C++.html , la page officielle de Bjarne Stroustroup sur le C++.

[Casteyde] The mega cours de C++
Christian Casteyde
http://casteyde.christian.free.fr/cpp/cours/

[Design Patterns] Design Patterns,
Elements of Reusable Object-Oriented Software
ADDISON-WESLEY
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
ISBN: 0-201-63361-2

[UML] UML Distilled,
ADDISON-WESLEY
Martin Fowler with Kendall Scott,
ISBN: 0-201-32563-2

[FAQ C++] C++ FAQ Lite version Française
Copyright 91-99, Marshall Cline
http://www.ifrance.com/jlecomte/c++/c++-faq-lite/index-fr.html