Projet HYDREAU

Volet GRANDES RIVIERES

Aperçu du langage C++ 24 mars 1993

Projet Hydreau

Aperçu du langage C++

André Houde
Marc Hughes
Yves Secretan
Michel Leclerc
INRS-EAU

1 Introduction

Ce document contient un résumé des principales fonctionnalités du C++. Il ne tient pas compte du fait que le lecteur connaisse ou non le langage C. Il explique la base du C++ (qui peut être celle du C également) ainsi que les caractéristiques faisant sa puissance.

Le C++ est l'extension orientée objet du langage C. Le seul fait qu'il permette une implantation objet en fait un compilateur beaucoup plus complexe que le C même si la base est semblable. Une compatibilité quasi-complète a été conservée par le C++ pour le C. Un programme C est compilé sans problème avec un compilateur C++ mais l'inverse (un programme C++ avec un compilateur C) n'est pas possible.

Ce document se divise donc en deux grandes parties. Les chapitres 2 à 7 inclusivement expliquent la base de programmation du C++ indépendamment de la conception orientée objet. Le lecteur devrait être en mesure de construire un programme simple en C++ avec toutes les caractéristiques de base acquises dans ces premiers chapitres.

Par la suite, le chapitre 8 couvre l'implantation des classes d'objets ainsi que toutes les notions d'héritage et de programmation orientée objet. Enfin, le chapitre 9 discute de la façon de construire un programme d'envergure avec plusieurs fichiers ainsi que des principes de compilation du C++.

1.1 Généralités

Le C++ fonctionne par appel de fonctions. D'abord, le programme débute dans une fonction principale ayant toujours le nom `main()`. Chaque fonction est délimitée par des accolades: une accolade ouverte (`'{'`) au début et une accolade fermée (`'}'`) à la fin. Les accolades sont aussi utilisées pour délimiter un bloc ainsi un bloc est généralement le code se retrouvant entre accolades suivant un énoncé de contrôle (incluant cet énoncé). Un bloc peut aussi se retrouver sans entête (c'est-à dire sans énoncé de contrôle).

Exemple:

```
main()
{
   // Ceci est le bloc 1
   if(...)
   {
      // Ceci est le bloc 2 qui est dans le bloc 1
      {
          // Ceci est un bloc sans entête
      }
   }
}
```

Tous les énoncés se terminent par un point-virgule (`;`) sauf pour les énoncés de contrôle (`if`, `for`, etc...) qui débutent un bloc. Un point-virgule sans expression est considéré comme l'énoncé nul:

```
       ;       // énoncé nul.
```

Il y a deux façons d'insérer des commentaires. D'abord, la première façon, les caractères '//' indiquent un début de commentaire jusqu'à la fin de la ligne où ils se trouvent.

Exemple:

```
char a; // Ceci est un commentaire.
        // int i; Cet énoncé n'est pas vu.
```

La deuxième façon est utile pour un commentaire sur plusieurs lignes qui débute par '/*' et se termine par '*/'.

Exemple:

```
/*
   char a;      // Ceci n'est pas du code vu par le compilateur
   On est toujours en commentaire.
   Il se termine ici. */
```

Attention à ce type de commentaire. Le C++ ne permet pas de commentaires imbriqués.

Exemple:

```
/* .1..
/* .2.. */
   .3.. */
```

Le compilateur, après avoir détecté un début de commentaire '/*', fermera ce commentaire dès la première occurence des caractères '*/' indépendamment de leur emplacement. Dans cet exemple, les parties 1 et 2 sont en commentaires, mais la partie 3 est considérée comme du code actif. Il y aura donc une erreur à la compilation.

2 Définition des variables

Une variable peut être définie n'importe où dans un programme, mais une seule fois à l'intérieur d'un même bloc. Une **définition** effectue une **allocation** et possiblement une **initialisation**. Une définition est référée comme étant un énoncé de déclaration. Seuls les énoncés de déclaration peuvent se retrouver à l'extérieur d'une fonction. La nuance entre une définition et une déclaration est précisée au dernier chapitre (section 9.3).

Exemple:

```
char a;         // Définition et allocation.
int i = 0;      // Définition, allocation et initialisation.
```

Plusieurs variables peuvent être définies sur la même ligne, lorsqu'elles sont de même type.

Exemple:

```
int i=0,j,k;    // On définit i, j, k et on initialise i.
```

Une variable est représentée par un identificateur comportant un ou plusieurs caractères alphanumériques de base (A-Z majuscule ou minuscule, 0-9 et le caractère souligné '_'). Un identificateur peut débuter par n'importe quel caractère sauf un chiffre. **Le compilateur C++ est sensible aux majuscules** et fait donc une différence entre `a` et `A`.

Le compilateur possède également une liste de mots réservés. Ces mots ne peuvent être utilisés en tant qu'identificateur.

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

2.1 Variables globales

Une variable globale est n'importe quelle variable définie à l'extérieur d'un bloc. Son rayon d'action s'étend de l'endroit où elle est définie jusqu'à la fin du fichier.

L'utilisation de ce type de variable devrait être minimale. En effet, les variables globales sont difficiles à maintenir et augmentent la complexité du code (surtout au niveau de la lisibilité et de la compréhension).

2.2 Variables locales

Une variable est locale lorsqu'elle est définie à l'intérieur d'un bloc. Elle est accessible à partir de l'endroit où elle est définie jusqu'à la fin du bloc.

Exemple:

```
main()
{
   char a;      // Variable locale au main()

   a = 'a';

   if(a == 'a')
   {
      a = 'b';

      int i =0;  // Variable locale au bloc if

   }
   // ici, la variable i n'est plus définie.
}
```

2.3 Accès aux variables globales de même nom

On peut accéder à une variable globale de même nom qu'une variable locale avec l'opérateur '::'. (Attention: mauvaise programmation!).

Exemple:

```
int i = 0;              // Définition et initialisation de la variable globale i.

main()
{
   int i = 5;           // Définition et initialisation de la variable locale i.

   i = i + 1;           // On incrémente la variable locale i (=6)
   ::i = ::i + 1;       // On incrémente la variable globale i (=1)

   // On opère avec les deux.
   ::i = i*2;           // Variable globale i = 12
                        // Variable locale i toujours égale à 6

   if(i > 0)
   {
      int i=20;
      i = i + 1;        // On incrémente la variable locale i (=21)
      ::i = ::i + 1;    // On incrémente la variable globale i (=13)
    }
}
```

3 Types

Le C++ fournit plusieurs types de données de base. Ces types et opérations de base font du C++ un langage simple qui garde toute sa puissance avec les possibilités de définition de nouveaux types (i.e. classes).

3.1 Types de base

La table suivante contient un résumé de tous les types disponibles, leur taille et l'intervalle qu'ils couvrent. Mis à part le type `char`, la taille de tous les types dépend de la machine utilisée. La taille donnée ici correspond à un micro-ordinateur sous OS/2. Par défaut, un type est signé (`signed`) et le modificateur `unsigned` (ex: `unsigned int`) indique un type non-signé. Ces modificateurs (`signed` et `unsigned`) ne s'appliquent qu'aux types `char` et `int`.

Type	Taille (bits)	Intervalle signé	Intervalle non-signé
`char`	8	-127 à 128	0 à 255
`int`	32	-2147483648 à 2147483647 (2G)	0 à 4294967295 (4G)
`short int`	16	-32768 à 32767 (32K)	0 à 65535 (64K)
`long int`	32	-2147483648 à 2147483647 (2G)	0 à 4294967295 (4G)
`float`	32	3.4E-38 à 3.4E+38	---
`double`	64	1.7E-308 à 1.7E+308	---
`long double`	80		---

`G: GIGA (1 000 000 000 de bytes)`

3.1.1 Caractère (`char`)

Une variable de type `char` représente un seul octet (8 bits). Lorsque signée (par défaut), elle peut prendre des valeurs entre -127 et 128 inclusivement et entre 0 et 255 lorsque non-signée.

Exemple:

```
char a;                 // -127 <= a <= 128
unsigned char b;        // 0 <= b <= 255
```

De façon générale, le type `char` est utilisé pour conserver un caractère. Par contre, ce caractère est stocké de façon numérique. Une variable de ce type peut donc être utilisée comme un entier de 8 bits.

Exemple:

```
main()
{
   char a = 'Z';       // La valeur ASCII de Z est 90. Donc, a = 90.
   char b = 10;

   int i = a + b;      // Puisque a = 90 et b = 10, alors i = 100.
}
```

3.1.2 Entier (`int`)

De façon générale, une variable de type `int` représente un mot de mémoire (2 mots sous OS/2). Un mot, dépendamment des machines, représente 2 ou 4 octets (un micro-ordinateur sous OS/2 possède des mots de 2 octets).

Une variable entière (`int`) peut être précédée par des modificateurs. Le modificateur `long` donne une taille de 2 mots et le modificateur `short` donne une taille de 1 mot (toujours dépendamment des machines).

Par défaut, une variable est de type `int`. C'est-à-dire que l'on peut définir une variable entière en ne spécifiant que les modificateurs (`unsigned`, `short` ou `long`).

Exemple:

```
unsigned i;           // Equivalent de unsigned int i;       0 <= i <= 4G
long j;               // Equivalent de long int j;        -2G <= j <= 2G
short k;              // Equivalent de short int k;    -32768 <= k <= 32767
unsigned long a;      // Equivalent de unsigned long int a; 0 <= a <= 4G
```

3.1.3 Point flottant (`float`, `double`)

Pour les variables à point flottant, on utilise les types `float`, `double` ou `long double` pour une précision simple, double ou étendue.

Typiquement, un `float` est représenté par un mot, un `double` par 2 mots et un `long double` par 3 ou 4 mots. Bien sûr, ceci varie sur différentes machines. Sur un micro-ordinateur sous OS/2, on a 2 mots (32 bits) pour un `float`, et 4 mots (64 bits) pour un `double` et 5 mots (80 bits) pour un `long double`.

Exemple:

```
float f;                    // 3.4E-38 <=f<= 3.4E+38 et -3.4E+38 <=f<= -3.4E-38
double d = 3.141592;        // 1.7E-308<=f<=1.7E+308 et -1.7E+308<=f<=-1.7E-308
```

3.2 Pointeurs

Une variable de type pointeur contient l'adresse d'une variable ou symbole contenant une donnée. Un pointeur possède aussi un type de donnée pour indiquer le contenu à l'adresse pointée. On définit un pointeur avec l'opérateur '*'.

Exemple:

```
int  *pi;       // On définit le pointeur de type int.
char *pc;       // On définit le pointeur de type char.
```

Attention! L'opérateur '*' est un opérateur de droite et s'applique à la variable et non au type. Il faut donc faire attention à une écriture comme celle-ci:

```
char* pc, pc2;
```

Dans cet exemple, pc2 est un caractère et pc est un pointeur à un caractère alors que l'on aurait tendance à lire que les deux variables sont des pointeurs.

Il est donc recommandé d'écrire l'association avec l'opérateur près de la variable.

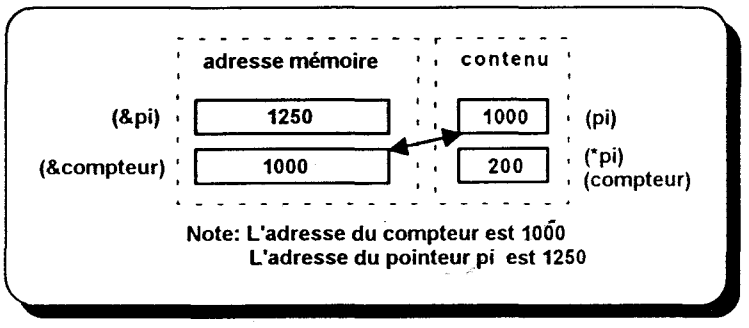
```
char *pc, *pc2;
```

3.2.1 Opérateurs de pointeur

Les opérateurs de pointeur sont '*' et '&' qui sont des opérateurs de droite. L'opérateur unitaire '&' retourne l'adresse mémoire de l'opérande.

Exemple:

```
int *pi;
int compteur;

compteur = 200;         // initialisation de compteur à 200.
pi = &compteur;         // pi pointe à la variable compteur.
```



On doit préciser le fait qu'une variable de type pointeur occupe un espace mémoire et possède une adresse au même titre qu'une variable ordinaire. C'est le contenu de la variable de type pointeur qui est l'adresse à laquelle on pointe. Dans l'exemple précédent, la variable `pi` de type

pointeur contient le nombre 1000 qui est l'adresse où l'on pointe, c'est à dire l'adresse de la variable compteur.

L'opérateur unitaire '*' est le complément de '&'. L'opérateur '*' appliqué à une variable de type pointeur retourne la valeur contenue à l'adresse pointée.

Exemple:

```
int *pi;
int compteur;
int valeur;

compteur = 200;        // initialisation de compteur à 200.
pi = &compteur;        // pi pointe à la variable compteur.

valeur = *pi;          // valeur contient maintenant la valeur 200.
```

Il doit absolument y avoir une correspondance exacte entre le type de pointeur et le type de donnée. Le C++ est un langage fortement typé et le compilateur vérifie la correspondance entre les types de chaque variable lors d'affectations ou d'opérations.

Un pointeur peut aussi être affecté à un autre pointeur

Exemple:

```
int valeur;
int *pi1, *pi2;

pi1 = &valeur;
pi2 = pi1;     // pi1 et pi2 contiennent la même adresse (variable valeur)
```

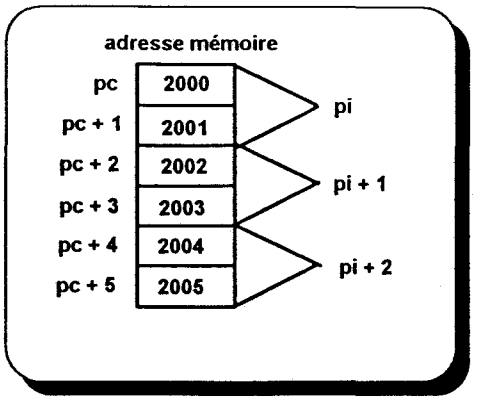
3.2.2 Opérations mathématiques

Il y a deux opérations mathématiques possibles sur un pointeur, l'addition et la soustraction. En additionnant une valeur à un pointeur, on incrémente l'adresse contenue dans ce pointeur. Ce dernier pointera donc à une autre valeur. Il faut prendre note que chacune des adresses des pointeurs est relative à son type. C'est-à-dire qu'en additionnant une valeur au pointeur, on ajoute en réalité cette valeur multipliée par la taille du type associé.

Exemple:

```
short *pi = 2000;
char  *pc = 2000;

pi = pi + 1;   // pi sera à l'adresse 2002;
pc = pc - 1;   // pc sera à l'adresse 2001;

pi = pi - 1;   // pi sera à l'adresse 2000;
```

Note: Dans cet exemple, on affecte des adresses fictives directement aux pointeurs. Cette pratique n'est pas applicable dans un contexte de programmation et est effectuée pour le bien de l'exemple uniquement.



3.2.3 Vecteurs

Tous les types sont éligibles à être des vecteurs. On définit une variable vecteur avec son type et sa taille.

Exemple:

```
int i[20];      // Vecteur de 20 entiers de type int.
```

Les indices de ce vecteur vont de 0 à 19. Chaque élément est accessible de la même façon qu'il est défini.

Exemple:

```
i[0] = 6;
i[3] = i[0] + 7;
```

Lors de la définition, la taille du vecteur doit absolument être une constante. Une taille variable génère une erreur.

Exemple:

```
const int taille = 20;
int i = 7;

int Vecteur[taille];          // ok
char Chaine[i];               // Pas bon
```

Les vecteurs à plusieurs dimensions se définissent de la même façon.

Exemple:

```
int i[5][3];    // Matrice de 5 rangées et 3 colonnes
```

On accède aux éléments de la même façon. Le tout s'applique pour n'importe quelle dimension supplémentaire.

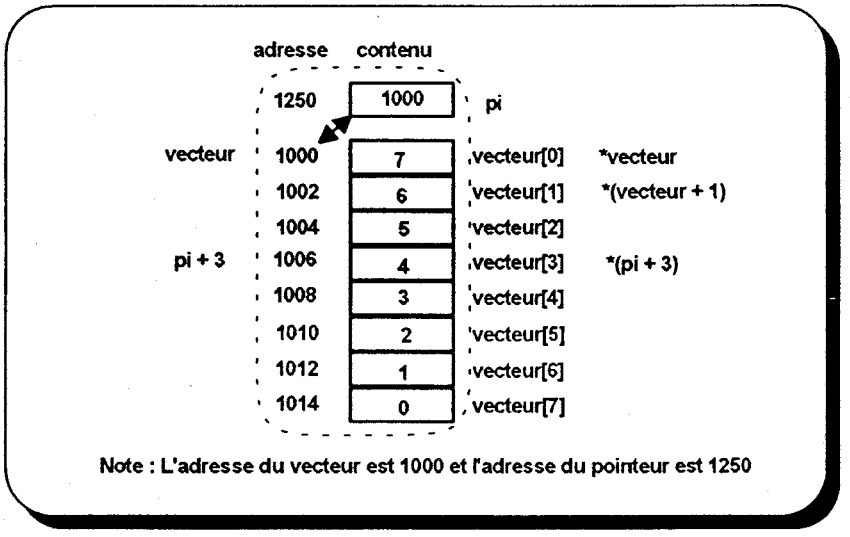
Lors de la définition d'un vecteur, tous ses éléments sont consécutifs en mémoire. Un vecteur de 10 caractères occupera un bloc de 10 octets en mémoire.

Il y a une relation directe entre les pointeurs et les vecteurs. En effet, lors de la définition d'un vecteur, on spécifie sa taille ainsi que son type. Un bloc de mémoire est alors alloué. L'adresse de ce bloc est accessible directement par le nom du vecteur (sans crochets). Ce nom peut aussi être utilisé au même titre qu'un pointeur.

Exemple:

```
short vecteur[8] = { 7, 6, 5, 4, 3, 2, 1, 0 }
short pi[];              // Équivalent de int *pi;
short valeur1, valeur2;

pi = vecteur;          // pi pointe à l'adresse du premier élément du vecteur.

valeur1 = vecteur[3];
valeur2 = *(pi + 3);
```

Dans cet exemple, `valeur1` et `valeur2` sont égales à 4.



3.3 Constantes

Une constante est n'importe quelle valeur ou variable qui sera inchangée à l'intérieur du programme en entier. Une constante peut être tout simplement une valeur, un caractère, une chaîne de caractères ou un identificateur d'un certain type.

3.3.1 Constantes littérales

À l'intérieur d'un programme, une valeur comme 12 qui apparaît dans un programme est perçue comme étant une constante littérale.

3.3.1.1 Caractères

La façon d'affecter un caractère explicite est de le mettre entre apostrophes.

Exemple:

```
a = 'z';        // La valeur du caractère z est placée dans a.
```

Les caractères explicites (entre apostrophes) sont donc convertis dans leur valeur correspondante à partir de la table de caractères de la machine (table ASCII pour les micro-ordinateurs) et peuvent être utilisés directement dans une opération.

Exemple:

```
char b = 10;
int i;
i = b + '2' + 5;        // b=10, le caractère 2 à un code ASCII de 50, donc i = 65.
...
```

Plusieurs caractères de contrôle doivent être considérés et sont précisés par une séquence d'échappement ("escape sequence"). Une séquence d'échappement est toujours précédée du caractère '\' (barre inversée ou "backslash"). Une séquence se retrouve toujours entre apostrophes pour spécifier un caractère ou entre guillemets lorsqu'elle est inclue à l'intérieur d'une chaîne de caractères. Ces séquences sont interprétées à la compilation pour être converties en un seul code correspondant au code de la table de caractères (ex: table ASCII).

```
\n      nouvelle ligne ("new line")
\t      tabulation horizontale ("tab")
\v      tabulation verticale
\b      retour arrière ("backspace")
\r      retour de chariot ("carriage return")
\f      nouvelle page ("form feed")
\a      alerte sonore ("beep")
\\      barre inversée ("backslash")
\?      point d'interrogation
\"      guillemet
\'      apostrophe
```

Une séquence d'échappement peut être généralisée sous la forme:

```
\ooo
```

où ooo est un nombre **sous forme octale** compris dans la table de caractères de la machine (ex: table ASCII).

Exemple: (pour la table ASCII)

```
\0      caractère nul
\062    caractère '2'
```

3.3.1.2 Entiers

Une constante littérale entière peut prendre plusieurs formes soit décimale, octale ou hexadécimale.

Exemple:

```
12      // Valeur décimale
012     // Valeur octale
0x12    // Valeur hexadécimale
```

Tout nombre est considéré décimal par défaut. Tout nombre précédé de 0 (zéro) prend la forme octale et tout nombre précédé de 0x ou 0X prend la forme hexadécimale.

Le type adopté pour chaque entier est `int`. On peut spécifier une valeur comme étant `unsigned` avec le suffixe `U` ou `u` ou `long` avec le suffixe `L` ou `l`. Il est fortement recommandé d'utiliser des majuscules pour ces suffixes pour éviter de confondre le `L` minuscule (`l`) avec le chiffre 1.

Exemple:

```
128U    // Constante considérée comme unsigned int
345L    // Constante considérée comme long int
529lu   // A ne pas faire! Ici, on peut confondre la lettre l pour le chiffre 1.
        // Cet énoncé est quand même bon (long unsigned int).
529UL   // Considérée comme unsigned long int.
```

3.3.1.3 Points flottants

Un nombre en point flottant tel `3.141592` sera de type `double`. Ici aussi, il est possible de modifier le type de la constante avec un suffixe. Le suffixe `F` ou `f` précise une constante de type `float` et le suffixe `L` ou `l` précise une constante de type `long double`. Pour les mêmes raisons que pour les entiers, il est fortement recommandé d'utiliser des majuscules pour ces suffixes.

Exemple:

```
1.2F    // Considéré comme float.
0.99L   // Considéré comme long double.
0.0     // Valeur 0 prenant le type double.
```

La notation exponentielle est indiquée par la lettre `E` ou `e`.

Exemple:

```
1.2E-4  // Équivalent de 1.2 X 10^-4
```

3.3.1.4 Chaînes de caractères

Une chaîne de caractères est représentée entre guillemets (contrairement à un seul caractère qui est représenté entre apostrophes). Cette chaîne sera considérée comme un vecteur de caractères, donc de type `char*` ou `char[]`. Toutes les séquences d'échappement peuvent être placées à

l'intérieur d'une chaîne. Rappelons que ces séquences sont interprétées à la compilation et sont remplacées par le code ou caractère qu'elles représentent.

Exemple:

```
"Bonjour!\n"   Représente la chaîne Bonjour! suivi d'une nouvelle ligne.
```

On peut aussi placer une chaîne de caractères sur plusieurs lignes en plaçant le caractère '\' (barre inversée ou "backslash") à la fin de chaque ligne pour indiquer la continuation de la chaîne sur la ligne suivante.

Exemple:

```
"Cette chaîne se retr\
ouve sur plusieurs li\
gnes. \n\n"
```

Puisqu'une chaîne est considérée comme un vecteur de caractères, le caractère nul ('\0') indique la fin d'une chaîne. Une chaîne de caractères prend donc comme espace mémoire chaque caractère de cette chaîne **plus** un caractère de fin de chaîne.

Exemple:

```
"Allo" On a 5 caractères, 4 pour Allo plus un caractère nul.
```

Une chaîne de caractères peut être affectée à une variable de même type **à l'initialisation seulement** dans la cas d'un vecteur. Pour opérer sur une chaîne de caractères, on doit utiliser les fonctions disponibles à cette fin (ex: `strcpy()`). Un pointeur peut par contre être directement affecté.

Exemple:

```
main()
{
   char s1[10] = "Allo!\n";          // C'est bon, c'est une initialisation
   char *s2 = "Bonjour",             // C'est bon aussi
        *s3, s4[20];

   s3 = "C'est bon, c'est un pointeur";      // Ok
   s4 = "Pas bon, c'est un vecteur";         // Erreur, la chaîne n'est pas copiée dans le
                                             // vecteur, le pointeur de tête est modifié.
}
```

3.3.2 Modificateur `const`

Le modificateur `const` est utilisé pour indiquer qu'un identificateur est une constante symbolique. Ce mot-clé sert à protéger les données de manipulations maladroites qui pourraient modifier la valeur d'une constante à l'intérieur du programme. Le modificateur `const` permet au compilateur de savoir quels identificateurs ne peuvent pas être modifiés. Ainsi, on est averti à la compilation si on tente de changer une constante. Une constante doit toujours être initialisée puisqu'il est impossible de l'affecter ailleurs.

Exemple:

```
int i = 0;              // Variable i
const int j = 6;        // Constante j initialisée à 6.  Partout dans le programme,
                        // j sera toujours égale à 6.
const int k;            // Définition inutile puisque k n'est pas initialisée (erreur).
```

Note aux programmeurs habitués du C: Le modificateur `const` est fortement suggéré pour remplacer les `#define`. Ceci permet surtout au compilateur de faire une vérification de type sur les constantes et ainsi s'assurer du bon type. On évite également un certain traitement du préprocesseur.

Il est impossible d'affecter l'adresse d'une constante symbolique à un pointeur. En effet, si cette opération était permise, on pourrait modifier la constante à l'aide du pointeur.

Exemple:

```
const int i = 0;              // Constante symbolique i.
int *pi = &i;                 // Erreur: pi est un pointeur à une variable.
```

Par contre, il est possible de définir un pointeur à une constante symbolique.

Exemple:

```
const int j = 0;              // Constante symbolique j.
const int *pj = &j;           // ok : pj pointe à une constante.
```

Attention, car dans cet exemple, on définit un pointeur à une constante et non un pointeur constant. Également, un pointeur à une constante peut adresser une variable, mais ne pourra jamais la modifier.

Exemple:

```
int i = 0;
const int *pi = &i;
...
*pi = 10;       // Erreur, on a spécifié que pi pointait à une constante
```

On peut définir des pointeurs constants. Ces pointeurs s'adresseront à une variable spécifique et ne pourront jamais être modifiés. Par contre, la valeur pointée pourra l'être.

Exemple:

```
int i = 0;              // Variable i;
int *const pi = &i;     // pi pointe à i et ne pourra jamais pointer ailleurs.
*pi = 10;               // La valeur de i peut être modifiée à l'aide de pi(i=10).
```

On peut finalement avoir des pointeurs constants à des constantes.

Exemple:

```
const int i = 0;
const int *const pi = &i;
```

3.4 Alias (typedef)

On peut donner un alias à un type de données existant (n'importe quel).
Syntaxe: `typedef` <type existant> <synonyme>

Exemple:

```
typedef unsigned long int Entier;     // On définit le synonyme Entier
...
main()
{
   Entier i = 0;   // La variable i est une variable de type unsigned long int
   ...
}
```

Cette fonctionnalité est utile surtout pour des questions de lisibilité et de compréhension du code. Également, on peut s'en servir pour augmenter la portabilité du code d'une machine à une autre (puisque la taille des types varie selon les machines).

On doit faire attention au fait que le `typedef` n'effectue pas un simple remplacement dans le code. Il interprète le type d'une certaine façon pour appliquer les modificateurs de la manière voulue.

Exemple:

```
typedef char* chaine;
const chaine allo;
```

Dans cet exemple, on aurait tendance à croire qu'on obtient un type `const char*` pour la variable `allo`. En réalité, puisque `chaine` est un nouveau type, c'est ce type qui doit être constant. Puisque c'est un type pointeur, c'est donc le pointeur qui est constant et le type équivalent dans cet exemple est `char *const`.

3.5 Enumérations

Un type de variable énuméré est un type contenant une liste symbolique de constantes entières (type `int`). Le premier élément d'une énumération prend la valeur 0 et les valeurs suivantes sont incrémentées de 1 suivant leur ordre.

Exemple:

```
enum Booleen { Faux, Vrai };  // Faux = 0, Vrai = 1
```

Le nom de l'énumération devient un type à part entière qui ne peut prendre comme valeur que les noms des éléments de sa liste.

On peut initialiser un élément dans la liste à une valeur entière spécifique.

Exemple:

```
enum Erreur {Run = -1, Memoire, Aucune};  // Run = -1, Memoire = 0, Aucune = 1
```

Une variable de ce type ne peut pas prendre une valeur entière intermédiaire.

Exemple: (Avec les énumérations précédentes)

```
main()
{
   int i=0;
   Booleen Var;
   Erreur Err = Aucune;        // ok

   Var = i;                    // Pas bon (avertissement seulement)
   Var = Err;                  // Pas bon (avertissement seulement)
   Var = Vrai;                 // ok
   i = Var;                    // ok, i = 1
}
```

4 Expressions et énoncés

Une expression est composée d'une ou plusieurs opérations. Les items utilisés dans une opération sont appelés opérandes et les opérations sont représentées par des opérateurs.

Les opérateurs s'appliquant à un seul opérande sont des opérateurs unitaires et les opérateurs agissant sur deux opérandes sont des opérateurs binaires.

Exemple:

```
char *pc;    // Opérateur unitaire '*'  (déréférence)
k = i * j;   // Opérateur binaire '*' (multiplication)
```

Dans cet exemple, on remarque que le même opérateur est utilisé dans deux contextes différents faisant de lui un opérateur unitaire dans le premier cas (déréférence) et un opérateur binaire dans le deuxième cas (multiplication).

4.1 Opérateurs

4.1.1 Arithmétique

Opérateur	Fonction	Syntaxe
*	multiplication	expr1 * expr2
/	division	expr1 / expr2
%	modulo (reste)	expr1 % expr2
+	addition	expr1 + expr2
-	soustraction	expr1 - expr2

Une ***division*** entre entiers résulte en un entier tronqué.

Exemple:

```
21 / 6 ;        // Le résultat est 3.
```

Un ***modulo*** calcule le reste d'une division entre entiers. Un modulo ne s'applique qu'à des entiers.

Exemple:

```
int i = 0;
float f = 2.0;

8 % 2;          // le résultat est 0. La division donne 4 reste 0.
10 % 4;         // le résultat est 2. La division donne 2 reste 2.
i % 2;          // ok
i % f;          // erreur.
3.0 % 2;        // erreur.
```

Attention: Une *division par zéro* génère une erreur à l'exécution du programme seulement. On doit également être vigilant au type de résultat obtenu car un débordement peut survenir.
Un *débordement* survient lorsque l'on tente d'affecter une variable avec une valeur qui dépasse les bornes de son type.

Exemple:

```
char c;
int i = 70;

c = i * 2;      // on tente d'affecter c avec 140 alors que ses bornes
                // sont entre -127 et 128.
```

4.1.2 Affectation

On affecte l'opérande de gauche avec le résultat de l'expression de droite à l'aide de l'opérateur '='. Le résultat de l'expression sera de même type de base que la variable que l'on tente d'affecter.

On peut concaténer plusieurs affectations l'une à la suite de l'autre.

Exemple:

```
int i,j;
i = j = 0;      // ok, 0 est affecté à i et j.
```

Par contre, on doit faire attention à la façon de concaténer si on le fait dans une initialisation.

Exemple:

```
int i = j = 0; // Pas bon, j n'est pas défini.
...
int j;
int i = j = 0; // ok, j existe.
```

Il existe aussi une notation compacte pour opérer et affecter dans le même opérateur. Ceci prend la forme générale:

```
x op= y;
```

où op est un de ces opérateurs binaires: +,-,*,/,%,<<,>>,&,|,^. Ceci est l'équivalent de:

```
x = x op y;
```

Exemple:

```
int i=0,j=0;
...
i += 10;        // Équivalent de i = i + 10;
j *= i;         // Équivalent de j = j * i;
```

Les opérateurs valides pour cette notation sont donc:

```
+=,-=,*=,/=,%=,<<=,>>=,&=,|=,^=
```

4.1.3 Incrémentation et décrémentation

Les opérateurs '++' et '--' offrent une notation compacte très pratique pour incrémenter ou décrémenter de 1 la valeur d'une variable. Chaque opérateur agit directement sur l'opérande et de ce fait devient un opérateur d'affectation également. L'opérateur peut être placé sous forme de préfixe ou de suffixe et aura deux effets différents.

Exemple:

```
int i = 0;
i++;            // Opérateur en suffixe.
--i;            // Opérateur en préfixe.
```

La différence entre les deux formes se fera sentir lorsque ces opérateurs sont intégrés à l'intérieur d'une autre expression.

Exemple:

```
int i = 0;
char c[10];

c[i++] = 'z';   // La variable i sera incrémentée après avoir affecté c.
                // Donc, c[0] = 'z' et i = 1.
```

On peut convertir cet exemple en deux lignes équivalentes:

```
c[i] = 'z';
i = i + 1;
```

Si maintenant pour le même exemple, on place l'opérateur d'incrémentation en préfixe:

```
c[++i] = 'z';   // La variable i sera incrémentée avant d'affecter c.
                // Donc, c[1] = 'z' et i = 1.
```

De la même façon, on peut convertir cette expression en deux autres équivalentes:

```
i = i + 1;
c[i] = 'z';
```

Note: La particularité de suffixe et de préfixe est implantée de la façon décrite dans la plupart des compilateurs. Par contre, certains compilateurs peuvent interpréter certains énoncés de différentes façons.

Exemple:

```
int i=0, v[10];
v[i] = i++;             // v[0] = 0 ou v[1] = 0 selon le compilateur.
```

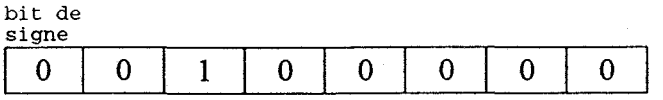
En effet, dans cet exemple, i sera incrémenté mais l'indexation dans le vecteur peut se faire avant ou après cette incrémentation.

4.1.4 Bits

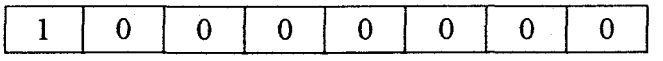
Les opérateurs sur les bits permettent de manipuler directement chaque bit d'une variable. Ces opérateurs ne s'appliquent qu'à des variables entières signées ou non-signées. Ils utilisent l'opérande comme une liste de bits où chaque bit prend la valeur 0 ou 1. Même si l'on est autorisé à manipuler des entiers signés, il est fortement recommandé de manipuler des entiers non-signés pour éviter de modifier par inadvertance le bit de signe. Une telle opération résulterait en une donnée erronée.

Exemple:

```
char c;
```




```
c << 2;
```



Dans cet exemple, puisque c est signé, lorsque l'on déroule les bits de 2 places, au lieu de multiplier par 4 et donner un résultat de 128, on obtient -128.

Opérateur	Fonction	Syntaxe
~	Négation	~expr
<< >>	déroulement à gauche déroulement à droite	expr1 << expr2 expr1 >> expr2
&	ET	expr1 & expr2
^	OU exclusif (XOR)	expr1 ^ expr2
|	OU	expr1 | expr2

L'***opérateur de négation*** ("~") inverse les bits de l'opérande. Chaque bit avec la valeur 0 devient 1 et chaque bit ayant 1 devient 0.

Exemple:

```
unsigned char b = 10;
```

```
b = ~b; // b = 245.
```

1	1	1	1	0	1	0	1

Les ***opérateurs de déroulement*** ("<<" vers la gauche et ">>" vers la droite) déroulent les bits d'un certain nombre de positions vers la gauche ou vers la droite. Les bits qui débordent d'un côté ou de l'autre sont ignorés. C'est toujours la valeur 0 qui est utilisée pour faire le remplissage dans le cas des entiers non-signés. Par contre, pour le déroulement vers la droite avec des entiers signés, le remplissage pourrait se faire avec des 1 si le bit de signe est actif (dépendamment des compilateurs).

Exemple:

```
unsigned char b = 72;
```

0	1	0	0	1	0	0	0

```
b >> 2; // b = 18.
```

0	0	0	1	0	0	1	0

```
b << 4;         // un bit déborde à gauche et est jeté. b = 32.
```

0	0	1	0	0	0	0	0

```
b >> 5; // b = 1.
```

0	0	0	0	0	0	0	1

Note: Ces opérateurs sont très utiles lorsque l'on veut opérer sur une base de 2 ("<<" pour la multiplication et ">>" pour la division). En effet, dans l'exemple précédent, l'opération `b >> 5` est l'équivalent d'une division par 32 (2^5). Le traitement est beaucoup plus efficace.

L'***opérateur ET*** ("&") agit sur deux opérandes. Pour chaque position, le résultat sera 1 si le bit de cette position est 1 pour chaque opérande. Autrement, le résultat sera 0. Attention pour ne pas confondre cet opérateur avec le ET logique ("&&").

Exemple:

```
unsigned char res;
unsigned char c1 = 0x65;
```

0	1	1	0	0	1	0	1

```
unsigned char c2 = 0xAF;
```

1	0	1	0	1	1	1	1

```
res = c1 & c2;
```

0	0	1	0	0	1	0	1

L'***opérateur OU*** ("|") agit également sur deux opérandes. Pour chaque position, le résultat sera 1 si une des deux opérandes ou les deux contiennent un 1 à cette position. Autrement, le résultat sera 0.

Exemple: (avec les définitions précédentes)

```
res = c1 | c2;
```

1	1	1	0	1	1	1	1

L'*opérateur OU exclusif* ("^") agit sur deux opérandes. Pour chaque position, le résultat sera 1 si une des deux opérandes mais pas les deux contient un 1 à cette position. Autrement le résultat sera 0.

Exemple: (avec les définitions précédentes)

```
res = c1^c2;
```

1	1	0	0	1	0	1	0

Ce type d'opérateur est très utile pour conserver des informations booléenes (oui/non, vrai/faux). On minimise l'espace utilisé en plaçant plusieurs informations dans une même variable où chaque bit de cette variable représente une information différente. On utilise donc ces opérateurs pour manipuler ce genre de donnée.

4.1.5 Taille (`sizeof`)

L'opérateur `sizeof` retourne la taille en octets d'une expression ou d'un type. Il peut prendre deux formes:

```
sizeof( type )
sizeof expr
```

Exemple:

```
int i;
i = sizeof i * sizeof(char) * sizeof(char*) * sizeof(char&) * sizeof(char[6]);
        // i = 4 * 1 * 4 * 4 * 6; (sur un micro-ordinateur sous OS/2).
```

4.1.6 Logique et relationnel

Les opérateurs logiques et relationnels sont utilisés pour évaluer une expression à vrai ou à faux. Une expression évaluée à vrai donne une valeur de 1 et une expression fausse donne 0.

Opérateur	Fonction	Syntaxe
!	Négation logique	!expr
< <= > >=	plus petit que plus petit ou égal que plus grand que plus grand ou égal que	expr1 < expr2 expr1 <= expr2 expr1 > expr2 expr1 >= expr2
== !=	égalité inégalité	expr1 == expr2 expr1 != expr2
&&	ET logique	expr1 && expr2
||	OU logique	expr1 || expr2

L'opérateur ET logique ("&&") résulte à vrai lorsque les deux opérandes résultent à vrai également. Autrement, il résulte à faux.

L'opérateur OU logique ("||") résulte à vrai lorsque l'un des deux opérandes résulte à vrai également. Si les deux opérandes résultent à faux, l'opérateur résulte à faux aussi.

Il est garanti que l'évaluation des opérandes se fait de gauche à droite. Pour ces deux opérateurs, il est également garanti que l'évaluation de l'expression s'arrête aussitôt que l'on en connaît le résultat.

```
expr1 && expr2
expr1 || expr2
```

Pour l'opérateur ET logique, si le premier opérande (`expr1`) est faux, il est garanti que le deuxième opérande (`expr2`) ne sera pas évalué et que le résultat sera faux.
Pour l'opérateur OU logique, si le premier opérande est vrai, il est garanti que le deuxième opérande ne sera pas évalué et que le résultat sera vrai.

Les opérateurs relationnels (`<, <=, >, >=`) et d'égalité (`==, !=`) effectuent une comparaison entre leurs opérandes et résultent à vrai (1) ou faux (0). Il faut être vigilant pour différencier l'opérateur d'égalité ("==") et l'opérateur d'affectation ("="). En effet, il est facile de se tromper en plaçant une condition et d'oublier un caractère "=" et ainsi affecter une variable à une nouvelle valeur alors que l'on voulait la tester.

L'opérateur de négation logique ("!") résulte à vrai si l'opérande est évalué à 0. Autrement, il résulte à faux. Cet opérateur est utile pour des questions de lisibilité et de compréhension du code.

Exemple:

```
int present = 0;
...
   if(!present)
   ...
```

Dans cet exemple, on lit *si pas present.* L'équivalent est:

```
   if(present == 0)
```

Écrit de cette façon, cette condition est moins facilement compréhensible même si le résultat est inchangé. Par contre, on doit faire attention pour ne pas abuser de cet opérateur.

Exemple:

```
...
   if(!strcmp(chaine1,chaine2))
   ...
```

La fonction `strcmp` est une routine de la librairie C. Elle compare deux chaînes de caractères et retourne 0 si elles sont égales. Dans cet exemple, on aurait tendance à comprendre que la condition est vraie lorsque les deux chaînes sont différentes, ce qui n'est pas le cas. Il est donc préférable d'écrire:

```
   if(strcmp(chaine1,chaine2) == 0)
```

De cette façon, on évite un brin de confusion.

4.1.7 Si arithmétique

Le si arithmétique prend la forme:

```
expr1 ? expr2 : expr3
```

La première expression (`expr1`) est toujours évaluée. À partir de son résultat, on choisit l'une des deux expressions suivantes (`expr2` ou `expr3`). On évalue la deuxième expression (`expr2`) si ce résultat est autre que zéro (condition vraie). Autrement, avec un résultat de zéro de la première expression (condition fausse), on évalue la troisième expression (`expr3`).

Exemple:

```
int j = 0;
int i = 2;
...
int z = (i > j) ? i : j;  // z prend la valeur maximale entre i et j.
```

Dans cet exemple, la variable z se verra attribuer la valeur de i.

4.1.8 Virgule

L'opérateur virgule (,) est utilisé pour séparer plusieurs expressions pour en faire un seul énoncé et l'utiliser comme expression à part entière. Les expressions sont évaluées de gauche à droite et le résultat d'une expression avec cet opérateur est le résultat de l'expression la plus à droite (la dernière).

Exemple:

```
int i,j;
int z = (i=0,j=6,7);   // On affecte i, ensuite j, et 7 est affecté à z.
```

4.2 Associativité et priorité

Une certaine priorité est accordée aux opérateurs (voir table). La priorité est l'ordre dans laquelle les opérations seront effectuées à l'intérieur d'une expression. On accorde aussi à un opérateur une certaine associativité pour résoudre le dilemme à savoir quel opérateur a priorité dans le cas d'opérateurs de même priorité. L'associativité peut être de gauche à droite ou de droite à gauche. C'est-à-dire, dans le cas de gauche à droite, les opérateurs de même niveau de priorité seront effectués à partir de la gauche de l'expression (du début). Dans le cas de droite à gauche, les opérateurs seront traités à partir de la droite de l'expression (de la fin). La table de la page suivante indique les niveaux de priorité des opérateurs ainsi que leur associativité. Les opérateurs de plus grande priorité se trouvent au haut de la table. Chaque niveau est accompagné de la lettre D ou G pour indiquer une associativité de droite à gauche ou de gauche à droite respectivement.

On doit accorder assez d'importance à cet aspect car il est la source de plusieurs erreurs logiques à l'intérieur d'un programme.

Exemple:

```
char c,chaine[20];
...
if(c = chaine[0] == '\0')
..
```

Dans cet exemple, on voudrait affecter `chaine[0]` à `c` et tester si ce caractère est le caractère nul. Par contre, l'opérateur d'égalité a un niveau de priorité plus élévé que celui d'affectation. Il y aura donc d'abord un test de comparaison et `c` se verra affecter le résultat de ce test soit 0 s'il est faux ou 1 s'il est vrai.

Pour inhiber l'effet de la priorité, on utilise les parenthèses qui divisent une expression en sous-expressions. L'exemple précédent devient donc:

```
if( (c = chaine[0]) == '\0')
```

Les sous-expressions de plus bas niveau sont d'abord évaluées. Donc, dans le même exemple, `c` est d'abord affecté et le résultat de cette sous-expression (la valeur de c) sera comparé au caractère nul.

17D 17G	:: ::	"scope" global "scope" classe
16G 16G 16G 16G	->,. [] () ()	sélection de membres index de vecteur appel de fonction construction de type
15D 15D 15D 15D 15D 15D 15D 15D	sizeof ++,-- ~ ! +,- *,& () new, delete	taille (en octets) incrémentation, décrémentation Négation Négation logique plus, moins unitaire déréférence, adresse-de conversion de type mémoire dynamique
14G	->*,.*	sélection de pointeurs aux membres
13G	*,/,%	opérateurs multiplicatifs
12G	+,-	opérateurs arithmétiques
11G	<<,>>	déroulement binaire
10G	<,<=,>,>=	opérateurs relationnels
9G	==,!=	égalité, inégalité
8G	&	ET binaire
7G	^	OU EXCLUSIF binaire (XOR)
6G	|	OU binaire
5G	&&	ET logique
4G	||	OU logique
3G	?:	opérateur SI
2D 2D 2D	=,*=,/= %=,+=,-=,<<= >>=,&=,|=,^=	opérateurs d'affectations
1G	,	opérateur virgule

Priorité plus élevée au haut de la table.
D: associativité de droite à gauche.
G: associativité de gauche à droite.

4.3 Conversions de types

Toutes les opérations effectuées sur des données doivent se faire sur la base d'un même type. Les conversions de type se font la plupart du temps de façon implicite (interne au compilateur). Les conversions explicites, par ailleurs, doivent être possibles dans le cas où le compilateur n'a pas de conversion existante pour passer d'un certain type à un autre.

De façon générale, une conversion de type s'applique pour promouvoir un type vers un deuxième type plus précis ou plus large (en octets), c'est-à-dire sans perte d'information. Par exemple, d'un type `int` vers `double`, `short` vers `long`, etc.

Les conversions dangereuses sont celles qui passent d'un type de données large vers un plus étroit. Essentiellement, cette pratique n'est pas appliquée. Par contre, il faut être très vigilant lorsque l'on passe d'un type signé à un type non-signé (pour les entiers) ou vice-versa. Le bit le plus significatif, s'il est 1, change complètement de nature s'il est appliqué à un type signé (bit de signe) ou non-signé.

Le compilateur peut effectuer certaines conversions implicites de façon bizarroïde lorsqu'une équation le moindrement complexe utilise différents types. On peut facilement perdre de la précision dans le résultat. Dans ce cas, il est recommandé d'utiliser les conversions explicites.

4.3.1 Conversions implicites

Les conversions implicites sont utilisées pour permettre l'éxécution d'une opération puisque cette dernière doit se faire sur la base d'un même type.

Exemple:

```
int i = 2;
i = i + 3.141592;      // i = 5.
```

Dans cet exemple, le nombre `3.141592` est de type `double`. Puisqu'il y a une opération avec un entier de type `int`, ce dernier se verra accorder une promotion au type `double` et sa valeur sera 2.0. Par la suite, pour effectuer l'opération d'affectation, une deuxième conversion se fera pour passer du type `double` au type de la variable où l'on veut mettre le résultat soit `int`. Le résultat sera donc tronqué à 5.

On doit faire attention à la notation. L'exemple ci-haut peut être réécrit sous la forme:

```
i += 3.141592;
```

Le résultat sera exactement le même **ainsi que le cheminement**. Les deux formes d'écriture sont équivalentes sur tous les points de vue alors que l'on aurait tendance à croire que le nombre est d'abord tronqué pour ensuite être additionné et affecté.

4.3.2 Conversions explicites

Les conversions explicites se font sous la forme:

```
type ( expr )
( type ) expr
```

Ces deux notations représentent une conversion explicitement demandée à l'intérieur du programme.

Exemple:

```
i = i + int(3.141592);
```

Dans cet exemple, il y aura une seule conversion appliquée. Le nombre `3.141592` est explicitement réduit à `3` et sera directement additionné à i sans aucun besoin pour d'autres conversions.

On doit faire attention aux conversions explicites s'appliquant à des types pointeurs. En effet, dépendamment de la syntaxe adoptée, le compilateur interprète différemment les lignes suivantes:

```
(char*)ptr;     // ok: conversion au type char*
char*(ptr);     // erreur
```

L'erreur se dégageant de cet exemple est tout simplement due à la priorité des opérateurs. L'opérateur '*' peut être associé aux parenthèses et le compilateur peut interpréter quelque chose de bizarre. Pour utiliser la deuxième forme de syntaxe, on doit passer par un alias (`typedef`) pour le type pointeur:

```
typedef char* charp;
...
charp(ptr);     //ok: conversion explicite au type charp.
```

La section suivante sur le type void explique un peu plus l'utilité des conversions explicites.

4.3.3 Type `void`

Le type de pointeur `void*` est un type générique utilisé pour adresser tous les types non-constants. Ce type permet d'adresser certaines données dont on ne connaît pas nécessairement le type à un certain point du programme.

Exemple:

```
int i = 0;
double d = 2.0;
void* pv;

pv = &i;
d = *pv;                // Pas bon.
d = (int*)pv;           // ok.
```

Dans cet exemple, le pointeur `void*` contient l'adresse de `i` mais ne sait pas de quel type est cette donnée. Il est donc impossible de déréférencer directement un pointeur de ce type. Une conversion explicite est donc obligatoire et la vérification de type ne se fait plus par le

compilateur. On doit s'assurer de la validité du type de la donnée à ce point dans le programme. Une mauvaise manipulation à ce niveau provoque des erreurs à l'éxécution du programme.

4.4 Énoncés de contrôle

Tout langage de base fournit plusieurs énoncés pour implanter le traitement des conditions et des boucles. Le C++ ne fait pas l'exception et donne quelques énoncés de base pour effectuer le traitement séquentiel du programme.

4.4.1 Énoncé `if-else`

L'énoncé `if` teste une condition particulière. Lorsque cette condition est vraie, on éxécute une certaine partie de code. Autrement, elle est ignorée. La syntaxe est la suivante:

```
if( expr )
   bloc;
```

L'expression doit se retrouver entre parenthèses. Si cette dernière donne un résultat égal à zéro, la condition est considérée comme fausse et les énoncés ne sont pas éxécutés. Toute autre valeur que zéro évaluée par l'expression donne une condition vraie et les énoncés associés sont exécutés.

Exemple:

```
int v[10];
...
if(v[0] == 1)
{
...
}
```

Lorsque plusieurs énoncés doivent être intégrés sous la même condition, un bloc doit être débuté après l'expression de cette condition comme dans l'exemple précédent. Si aucun bloque n'est ouvert seul l'énoncé suivant la condition est associé à cette dernière.

Exemple:

```
int v[10];
...
if(v[0] == 1)
   v[1] = 2;    // Cet énoncé est associé à la condition.
   v[2] = 3;    // Pas celui-ci.
```

Dans cet exemple, on remarque que le second énoncé est exécuté de façon inconditionnelle. Son indentation par contre porte à croire qu'il est associé à la condition. Ce type d'erreur est fréquent et il est préférable de mettre un bloc ou de placer l'énoncé associé sur la même ligne que la condition:

```
if(v[0] == 1) v[1] = 2;
v[2] = 3;
```

Encore ici, on souligne l'importance de différencier l'opérateur d'égalité ("==") et l'opérateur d'affectation ("="). En effet, une seule erreur typographique change complètement la nature de la condition:

```
if(v[0] = 1) v[1] = 2;
```

Ici, en changeant l'égalité pour une affectation, `v[0]` se voit affecter la valeur 1 et puisque 1 est autre que 0, la condition est toujours vraie.

Un énoncé de type **Si-Alors** a également un énoncé de type **Sinon**. L'énoncé `if` peut donc se voir associer l'énoncé `else`. La syntaxe prend cette forme:

```
if( expr )
   bloc;
else
   bloc;
```

Tout comme l'énoncé `if`, on peut associer un bloc ou un seul énoncé au `else`. Les suggestions d'écriture pour le `if` s'appliquent autant pour le `else`.

On doit également être vigilant à une indentation comme celle-ci:

```
if(...)
   if(...)
      if(...)
         ...
else
   ...
```

Dans ce cas-ci, l'énoncé `else` est associé au dernier énoncé `if` et non au premier comme le laisse croire une écriture comme celle-ci. En effet, l'énoncé `else` s'applique au dernier énoncé `if` rencontré à moins que cet effet ne soit inhibé par l'ajout d'accolades.

Exemple:

```
if(...)
   if(...)
      if(...)  // else associé ici
         ...
      else
         ...

if(...) // else associé ici
{
   if(...)
      if(...)
         ...
}
else
   ...
```

4.4.2 Énoncé `switch`

L'énoncé `switch` est surtout utilisé pour remplacer plusieurs niveaux de `if-else-if` imbriqués. Le résultat de l'expression du `switch` est comparé avec une liste de valeurs **constantes**.

```
switch( expr )
{
case V:
   ...
   break;
...
default:
   ...
}
```

Dans cette syntaxe, on peut retrouver plusieurs `case` et `v` doit être une constante (littérale, `const`, ou énumérée). L'énoncé `break` est utilisé pour quitter le bloc `switch`. L'énoncé `default` permet de traiter le cas où le résultat de l'expression ne correspond à aucune valeur énumérée dans la liste de `case`.

Exemple:

```
char c = 'a';
...
switch(c)
{
case 'a':       // c == 'a'
   ...
   break;
case 'z':       // c == 'z'
   ...
   break;
default:        // c est aucun cas de la liste (ni 'a' ni 'z').
   ...
   break;
}
```

On doit être vigilant au fait que lorsqu'un cas est rencontré et que son code est exécuté, l'énoncé `switch` ne s'arrête pas automatiquement lorsqu'il rencontre le prochain cas. Il sort de son bloc à un énoncé `break` ou à la fin du bloc seulement.

Exemple:

```
switch(c)
{
case 'a':       // si c == 'a'
   ...          // le code ici est exécuté
case 'z':       // il n'y a pas eu de break, donc on continue
   ...          // le code ici est aussi exécuté
   break;
}
```

Cette particularité peut être la source de quelques erreurs mais peut aussi être utilisée pour traiter certains cas.

Exemple:

```
switch(c)
{
case 'a':       // Si c est un a minuscule ou majuscule,
case 'A':       // on effectue le même traitement.
   ...
   break;
case 'z':
case 'Z':
   ...
   break;
}
```

4.4.3 Énoncé `while`

L'énoncé de boucle `while` possède la syntaxe suivante:

```
while( expr )
   bloc;
```

Tous les énoncés du bloc seront exécutés tant que l'expression (`expr`) résulte en une valeur non-nulle (autre que zéro). Dès que l'évaluation de `(expr)` du `while` est zéro, la boucle se termine et le traitement continue après le bloc de la boucle.

Exemple:

```
int i=0;
while(i < 10)
{
   ...
   i++;
}
// ici, i = 10.
```

4.4.4 Énoncé `for`

L'énoncé itératif `for` est conçu pour faciliter la manipulation d'une boucle itérative.

```
for(expr1; expr2; expr3)
   bloc;
```

La première expression (`expr1`) est exécutée au début de la boucle seulement (une seule fois peu importe le nombre d'itérations). Cette expression peut être déclarative.

Exemple:

```
for(int i=0,j; ...; ...)    // expression déclarative.
for(i += 10; ...; ...)
```

La deuxième expression (`expr2`) est la condition de boucle. La boucle continue tant et aussi longtemps que cette expression donne un résultat non-nul.

Exemple:

```
for(...; i < 20; ...)
for(...; 1; ...)            // boucle infinie.
```

La troisième expression (`expr3`) est exécutée à la toute fin de chaque itération de la boucle. Il est garanti que cette expression ne sera pas exécutée si la condition de boucle (`expr2`) est fausse dès la première itération.

Exemple:

```
for(int i=0; i < 20; i++)
{
   ...  // Cette boucle est exécutée 20 fois.
}
```

Les expressions `expr1` et `expr3` peuvent être des énoncés nuls.

```
for(; i < 20;)
```

On peut considérer l'énoncé `for` comme l'équivalent de cette syntaxe avec l'énoncé `while`:

```
expr1;
while(expr2)
{
   bloc;
   expr3;
}
```

Toute définition de variable faite à l'intérieur de la première expression (`expr1`) n'est pas locale au bloc `for`. Elle sera locale au bloc du niveau de l'énoncé `for`. Ceci se voit par l'équivalence avec l'énoncé `while` et s'explique par le fait que la première expression (`expr1`) est toujours évaluée en débutant un énoncé `for`.

4.4.5 Énoncé `do-while`

L'énoncé de boucle `do-while` est utilisé dans le cas où l'on veut qu'une première itération de boucle soit exécutée de façon inconditionnelle.

```
do
{
   énoncés;
}while( expr );
```

La boucle se termine lorsque l'expression de condition (`expr`) évalue un résultat de zéro.

4.4.6 Énoncé `break`

L'énoncé `break` s'applique aux énoncés `switch`, `while`, `for` et `do`. Il termine l'énoncé le plus local et continue l'exécution immédiatement suivant le bloc de l'énoncé concerné. Dans le cas des boucles (le cas du `switch` ayant déjà été couvert), cet énoncé est utile lorsque l'on veut terminer leur exécution après avoir testé une certaine condition à l'intérieur de la boucle.

Exemple:

```
while(1)        // boucle infinie
{
   ...
   if(ch == 'a') break;
   ... // Si ch == 'a', cette partie n'est pas exécutée
}
// On tombe directement ici après le break.
```

Dans cet exemple, on remarque la grande utilité de cet énoncé pour des boucles dont on ne veut pas tester la condition au début. On peut faire une boucle infinie et placer l'énoncé `break` associé avec la condition de sortie de boucle.

4.4.7 Énoncé `continue`

L'énoncé `continue` termine l'itération courante d'une boucle `for`, `while` ou `do`. L'exécution résume au début de la boucle dans le cas du `while` et du `do` et dans le cas du `for`, la troisième expression est exécutée avant de continuer la prochaine itération.

Exemple:

```
while(1)        // Après un continue, on retombe ici
{
   ...
   if(ch == 'a') continue;
   ...  // Cette partie n'est pas exécutée pour cette itération si ch == 'a'
}

for(int i=0; i < 20; i++)      // On retombe ici après un continue et
{                              // après avoir exécuté i++
   ...
   if(...) continue;
   ...  // Après un continue, cette partie n'est pas exécutée
}
```

4.4.8 Énoncé `goto`

L'énoncé `goto` déplace l'exécution du programme de façon inconditionnelle à un endroit identifié par une étiquette. La syntaxe prend la forme suivante:

```
goto etiquette;
```

L'étiquette peut se trouver n'importe où dans un même programme et prend la forme suivante:

```
etiquette:
```

Cet énoncé de saut inconditionnel ne doit pas sauter par dessus une définition de variable avec une initialisation implicite ou une initialisation explicite.

Exemple:

```
main()
{
   int i = 0;
   ...
   if(i == 0) goto fin;
   int j;              // ok, variable pas initialisée
   int k = 10;         // Pas bon, initialisation.
   fin:
}
```

L'utilisation de cet énoncé constitue une mauvaise programmation et provoque un méli-mélo difficile à déchiffrer (mieux connu sous le nom de code spaghetti). Son usage devrait donc en être très restreint et se limiter à alléger le programme lorsque l'on veut sortir de plusieurs niveaux de boucle d'un seul coup où les énoncés `break` alourdissent cette tâche.

5 Mémoire dynamique

Contrairement à la mémoire statique qui est allouée à la compilation, la mémoire dynamique est allouée à l'exécution. On gére les variables dynamiques à l'aide des opérateurs `new` et `delete`.

L'opérateur `new` retourne un pointeur à l'élément nouvellement alloué (donc son adresse) du type de cet élément.

Exemple:

```
int *pi;
pi = new int;  // On alloue la taille d'un int.
```

On peut aussi allouer dynamiquement des vecteurs à l'aide de crochets (`[]`). La taille du vecteur doit être spécifiée à l'aide d'une expression quelconque.

Exemple:

```
int i=1;
char *s = new char[i*10];
```

Puisque l'allocation est dynamique, on doit aussi voir à la désallocation de cette même mémoire. L'opérateur `delete` permet de désallouer la mémoire des variables désuètes. Il est très important de désallouer la mémoire dès que l'on a terminé de l'utiliser. Les oublis monopolisent beaucoup d'espace mémoire et font que l'on épuise cette ressource beaucoup plus rapidement.

Exemple:

```
delete pi;      // La mémoire associée à pi est désallouée
```

On doit faire attention pour ne plus utiliser une variable dont on vient de désallouer la mémoire. En effet, ceci cause des erreurs monumentales à l'exécution puisque l'on adresse une variable qui n'existe plus.

Exemple:

```
delete pi;
*pi = 2*3;      // Non: très dangereux, erreur à l'exécution.
```

Pour effacer un vecteur, on utilise le même opérateur mais avec des crochets ([]) pour spécifier que l'on efface le vecteur au complet. Sinon, le premier élément seulement sera effacé.

Exemple:

```
delete[] s;     // Le vecteur complet est désalloué
delete s;       // Seul le premier caractère est effacé.
```

Cet opérateur ne doit pas s'appliquer sur tous les pointeurs. Les variables allouées statiquement à la compilation ou initialisées sont désallouées par le compilateur et l'opérateur `delete` ne s'applique qu'aux variables allouées par l'opérateur `new`.

Exemple;

```
int i;
int *pi = &i;
char *s = "Allo\n";
long *pi2 = new long;
...
delete pi2;     // ok
delete s;       // dangereux
delete pi;      // dangereux
```

Dans cet exemple, les erreurs ne sont pas générées à la compilation mais surviendront de façon imprévisible à l'exécution.

L'opérateur `delete` ne peut pas s'appliquer sur des pointeurs à des constantes (modificateur `const`).

Exemple:

```
const int *pi = new int;
...
delete pi;      // erreur
```

Cet opérateur peut s'appliquer à un pointeur nul sans danger.

Exemple:

```
int *pi = 0;
...
delete pi;
```

L'usage du `new` doit se faire également avec vigilance. En effet, la mémoire dynamique n'est pas infinie et le programme peut à un certain point avoir rempli la mémoire et ne plus pouvoir en allouer (surtout si on alloue constamment sans jamais désallouer). À ce moment, l'opérateur `new` retourne la valeur 0. Le programme doit donc prévoir ce cas et continuer seulement si l'allocation de mémoire a réussi. Autrement, la première opération effectuée avec cette variable qui pointe à 0 causera une erreur à l'exécution (ça plante).

Exemple:

```
int *pi;
pi = new int;          // S'il n'y a plus de mémoire, pi = 0.
*pi = 18;              // Il y a une erreur car *pi n'existe pas.
...
pi = new int;
if(pi == 0)            // traiter l'erreur;
   ...
```

On voit évidemment qu'un test pour chaque énoncé `new` allourdit un programme de façon monstrueuse. La variable globale **`_new_handler`** est un pointeur à une fonction (les détails des pointeurs aux fonctions sont donnés à la section 6.7) qui est appelé dès que n'importe quel énoncé `new` retourne la valeur 0 au moment où la mémoire est épuisée. Par défaut, cette variable ne pointe à rien et aucune fonction n'est appelée si un énoncé `new` retourne 0. On peut donc définir sa propre fonction qui sera appelée dès qu'un énoncé `new` ne parvient pas à allouer de mémoire. Cette fonction a un type de retour `void` et ne prend pas d'arguments. On affecte la variable `_new_handler` avec l'opérateur '=' tout simplement ou par la fonction `set_new_handler()`

.

Exemple:

```
void FonctionPlusDeMemoire();
_new_handler = FonctionPlusDeMemoire;
set_new_handler(FonctionPlusDeMemoire);      // Enonce equivalent au precedent
```

6 Fonctions

Une fonction peut être perçue comme une opération construite sur mesure. Elle est représentée par un nom et ses opérandes sont donnés dans une liste d'arguments compris entre parenthèses (" () ") et séparés par des virgules. Une fonction peut donner un résultat mieux connu sous le nom de *type de retour*. Si elle ne retourne aucun résultat, elle est dite être de type `void`. Le corps d'une fonction (ses énoncés) se retrouve à l'intérieur d'un bloc. On y réfère comme étant sa définition.

Exemple:

```
int calculBidon(int x, int y) // Définition de calculBidon
{
   // Corps de la fonction
   return x * y;                    // Retourne le résultat de type int.
}
```

À la définition d'une fonction, on doit spécifier le type de chaque argument ainsi que le type de retour. L'utilisation de la fonction est reconnu comme étant un *appel* de fonction. À chaque appel de fonction, le compilateur vérifie la correspondance entre les types des arguments. Le nombre d'arguments placés à l'appel doit correspondre exactement au nombre actuel dans la définition. On rappelle que le C++ est un langage fortement typé et que tous les types d'arguments sont vérifiés à l'appel.

Exemple:

```
int i=9,j;
float f;
char *s;
...
j = calculBidon(i,10);        // 10 est un entier par défaut.
j = calculBidon(f,s);         // Erreur, mauvais types.
j = calculBidon(i);           // Erreur, manque un argument
...
```

Avant d'utiliser une fonction, cette dernière doit absolument être définie ou déclarée pour informer le compilateur de son existence. Pour déclarer une fonction, on doit spécifier son type de retour, son nom, et le type de ses arguments. Une déclaration de ce genre est appelée un *prototype de fonction.*

Exemple:

```
int calculBidon(int, int);    // Prototype de calculBidon
```

On remarque que les noms des arguments ne sont pas nécessaires, seulement leur type

6.1 Arguments

Une fonction ne possède pas obligatoirement une liste d'arguments. On peut ne pas spécifier d'arguments en laissant les parenthèses vides ou en spécifiant le type `void`.

Exemple:

```
int calculBidon();              // Fonction sans arguments.
int calculBidon(void);          // Équivalent de la déclaration précédente.
```

Un argument se passe de différentes façons. On a d'abord le **passage par valeur**. Lors de l'appel de la fonction une copie de la valeur est passée à cette fonction. Un argument passé par valeur est spécifié par son type seulement.

Exemple:

```
int calculBidon(int,int);       // Passage par valeur.
int i = 10, j = 9;
...
j = calculBidon(i,j);
// Ici, i est toujours égal à 10.
...

int calculBidon(int x, int y)
{
   // Passage par valeur, on reçoit des copies des arguments dans x et y.
   int res = x * y;
   x = 5;               // Passage par valeur, aucun changement à i qui a été passé.
   return res;
}
```

Le **passage par pointeur** se fait de la même façon, sauf qu'un argument est spécifié avec l'opérateur de déréférence pour indiquer un type pointeur. Également dans ce cas, une copie du pointeur (une copie de l'adresse) est passée à la fonction. Ce type de passage d'arguments est utilisé pour être capable de modifier la valeur pointée à l'intérieur de la fonction.

Exemple:

```
int calculBidon(int*,int*);     // Passage par pointeur.
int i = 10, j = 9;
int *pi = &i, *pj = &j
...
j = calculBidon(pi,pj);
// Ici, pi pointe toujours à i mais i vaut maintenant 5.
...

int calculBidon(int *x, int *y)
{
   // Passage par pointeur, on reçoit des copies des pointeurs dans x et y.
   int res = *x * *y;
   *x = 5;              // Passage par pointeur, changement à la valeur de pi
                        // qui a été passé.
   x = y;               // On change le pointeur x, mais aucun changement à pi.
   return res;
}
```

Le **passage par référence**, contrairement aux deux autres méthodes, ne fait pas de copies lors du passage des arguments. La fonction reçoit l'adresse de la variable et utilise donc la variable en elle-même et non une copie.

Exemple:

```
int calculBidon(int&,int&);   // Passage par référence.
int i = 10, j = 9;
...
j = calculBidon(i,j);
// Ici, i est maintenat égal à 5.
...

int calculBidon(int &x, int &y)
{
   // Passage par référence, on reçoit les variables i et j dans x et y.
   int res = x * y;
   x = 5;              // Passage par référence, changement à i qui a été passé.
   return res;
}
```

Un argument de type pointeur peut aussi être passé par référence. Ceci est utile lorsque l'on veut modifier le pointeur lui-même dans la fonction.

Exemple:

```
int calculBidon(int*&,int*&); // Passage de pointeurs par référence.
int i = 10, j = 9;
int *pi = &i, *pj = &j
...
j = calculBidon(pi,pj);
// Ici, pi pointe maintenant à j.
...

int calculBidon(int *&x, int *&y)
{
   // Passage de pointeurs par référence, on reçoit les pointeurs dans x et y.
   int res = *x * *y;
   *x = 5;             // Passage de pointeur, changement à la valeur de pi
                       // qui a été passé.
   x = y;              // On change le pointeur x, changement à pi puisque
                       // passé par référence.
   return res;
}
```

Un argument peut être spécifié avec un modificateur `const` pour empêcher la fonction de modifier cet argument.

Exemple:

```
int calculeBidon(const int &x, const int &y)
{
   ...
   x = 5;      // Erreur, argument de type const.
   ...
}
```

Un vecteur ne peut jamais être passé par valeur. Il doit être passé par le pointeur à son premier élément (élément 0).

Exemple:

```
// Déclarations équivalentes.
void calculBidon(int*);
void calculBidon(int[]);
void calculBidon(int[10]);
```

Toutes les manipulations sur le vecteur à l'intérieur de la fonction sont faites sur l'original. Également, en C++, il n'y a aucune vérification de taille de vecteurs. On doit donc s'assurer de ne pas dépasser les bornes.

Les vecteurs multidimensionnels doivent spécifier toutes les dimensions suivant la première. Le compilateur doit connaître ces dimensions pour être en mesure d'effectuer une opération du style:

```
void calculBidon(int matrice[][10])
{
   matrice++;  // Passe à la rangée suivante.
}
```

En effet, le compilateur doit savoir de combien augmenter l'adresse pour passer à la rangée suivante.

6.1.1 Arguments par défaut.

Les arguments par défaut sont utilisés pour éviter de toujours spécifier un argument qui prend la plupart du temps la même valeur. Un argument prenant une valeur par défaut doit se retrouver à droite de la liste. On peut en spécifier plusieurs en autant qu'ils soient tous à droite et qu'ils ne soient pas entrecoupés par un autre argument sans valeur par défaut. Les valeurs par défaut sont spécifiées soit dans la définition ou dans le prototype (mais pas dans les deux). Un argument ne doit pas être affecté à une valeur par défaut plus d'une fois malgré qu'un prototype peut être déclaré plusieurs fois en spécifiant un différent argument par défaut à chaque fois.

Exemple:

```
// Erreur: la valeur par défaut n'est pas à droite
void calculBidon(int x, int y=0, int z);

void calculBidon(int x, int y, int z=0);     // ok

// Erreur: z a déjà sa valeur par défaut
void calculBidon(int x, int y=0, int z=0);

void calculBidon(int x, int y=0, int z);     // ok
void calculBidon(int x=0, int y, int z);     // ok

// Ces trois déclarations équivalent la déclaration suivante
void claculBidon(int x=0, int y=0, int z=0); // ok
```

À l'appel de la fonction, on spécifie seulement les arguments auxquels on doit donner une valeur (sans valeur par défaut) et ceux dont on veut changer la valeur par défaut. On doit absolument conserver un certain ordre dans les arguments, c'est-à-dire que, par exemple, il est possible de spécifier le dernier argument si et seulement si tous les arguments précédents ont aussi été spécifiés.

Exemple:

```
// Équivalent de calculBidon(0,0,0)
calculBidon();

// Équivalent de calculBidon(2,0,0)
calculBidon(2);

// Équivalent de calculBidon(2,5,0)
calculBidon(2,5);

// Appel de la fonction
calculBidon(2,5,92);

// Erreur: le deuxième argument n'est pas spécifié même s'il a une valeur
// par défaut.
calculBidon(2,,92);
```

6.1.2 Liste variable (...)

Dans certains cas, on ne peut pas savoir d'avance le nombre d'arguments qui sera donné à une fonction. Dans cette situation, on peut utiliser les trois petits points(...). Cette spécification indique que la fonction prend un nombre indéterminé d'arguments de type inconnu. On peut l'écrire suivant une liste d'arguments obligatoires ou seul:

Exemple:

```
void calculBidon(int,...);
void calculBidon(...);
```

Attention pour ne pas confondre cette notation avec celle indiquant aucun argument:

```
void calculBidon();          // Aucun argument
void calculBidon(...);       // Nombre indéterminé d'arguments
```

Ce type d'arguments doit être utilisé avec les fonctions `va_arg`, `va_end` et `va_start`. Consultez la documentation du compilateur pour les détails d'implémentation de ce type d'arguments.

6.2 Retour d'une valeur

Tous les types sont éligibles à devenir un type de retour d'une fonction. Par défaut, le type est `int`. On peut retourner aussi un type dérivé comme `int&` ou `char*` tout comme le type `void`. Un vecteur ne peut pas être un type de retour tout comme une fonction. On peut le faire par contre à l'aide d'un pointeur.

Exemple:

```
int calculBidon();
calculBidon();          // énoncé équivalent au précédent
int &fou();
char *chaine();
int[9] allo();          // Erreur: un vecteur ne peut pas être retourné.
void rien();            // ne retourne rien.
```

À l'intérieur de la fonction, on doit spécifier la valeur retournée. Ceci se fait avec la syntaxe:

```
return expression;      // expression donne un résultat du type de retour.
return;                 // pour le type void
```

Dans le cas où l'on retourne une valeur, on doit spécifier un énoncé `return`. Cet énoncé n'est pas nécessaire pour un type de retour `void`.

Exemple:

```
int calculBidon()
{
   int i=0;
   ...
   if(i == 0) return;          // Erreur: aucune expression associée.
   if(i == 0) return i;        // ok
   // Erreur: si i != 0, on ne retourne pas de valeur.
}
void rien()
{
   ...
   if(...) return;     // ok: type void
   // ok: pas besoin de retour explicite, type void.
}
```

Si on doit retourner plusieurs valeurs différentes pour la même fonction, il est préférable d'en faire une fonction de type `void` et que les résultats soient adressés dans les arguments de cette même fonction.

On peut aussi retourner une référence à une variable avec l'opérateur `&`. L'adresse d'une variable est retournée.

Exemple:

```
int x;
int &nombre()
{
   x = 5;
   return x;
}
main()
{
   int i = nombre();   // i = x = 5.
   i++;
   nombre() = i;       // x = i = 6.
}
```

On doit être vigilant avec ce type de retour car on doit s'assurer que la variable existe toujours lorsque l'on sort de la fonction. Un retour d'une variable locale est très dangereux.

Exemple:

```
int &nombre()
{
   int x = 5;
   return x;   // DANGEREUX: x n'existe plus lorsque l'on retourne.
}
```

6.3 Récursion

Une fonction s'appelant elle-même de façon directe ou indirecte est dite récursive. La récursion est dite de premier niveau lorsqu'une fonction s'appelle elle-même et de x niveaux lorsque x

fonctions s'appelent entre elles avant de fermer la récursivité. On doit toujours avoir une condition d'arrêt à l'intérieur d'une fonction de ce type. Autrement, la récursivité serait infinie.

Exemple:

```
int factorielle(int x)
{
   if(x > 1) x *= factorielle(x-1);
   return x;
}
```

Une fonction récursive est l'équivalent d'une boucle avec une pile. Même si la récursion est plus lente qu'une méthode avec une boucle, elle est plus lisible et plus compacte.

6.4 Macros (`inline`)

Les macros peuvent être considérées comme étant une fonction très courte effectuant un traitement simple. Souvent, implanter ce genre de fonction comme fonction à part entière alourdit l'exécution de façon inutile (appel de la fonction, empiler les arguments, etc..). Le mot-clé `inline` placé à l'avant de la définition d'une fonction donne indique au compilateur qu'il peut s'agit d'une macro. L'appel d'une macro est donc remplacé par son code à la compilation.

Exemple:

```
inline int carre(s) { return s*s; }
```

Notons que le mot-clé `inline` ne donne qu'un indice et ne fait pas de la fonction une macro de façon inconditionnelle. Si la fonction est trop complexe ou même récursive, le compilateur peut décider d'en faire une fonction à part entière.

De façon générale, les macros n'ont pas de prototypes puisque leur définition est courte et placée en entête de programme.

Note aux habitués du C: Cette façon de faire devrait remplacer inconditionnellement les `#define` pour les macros. En effet, puisqu'une fonction `inline` est considérée comme une fonction, on évite les erreurs du genre:

```
#define carre(s) ((s)*(s))
...
carre(x++);     // est remplace par ((x++)*(x++)), woops
```

Le mot-clé `inline` évite ce genre de problème et la macro est traitée comme une fonction à part entière.

6.5 Surcharge

En C++, il est possible de définir plusieurs fonctions de même nom avec des arguments différents. L'utilité de cette caractéristisque se démontre lorsque la même fonction s'applique pour différents types d'arguments. À la définition et au prototypage, c'est la liste d'arguments qui différencie une fonction d'une autre de même nom (le type de retour n'est pas examiné).

Exemple:

```
int max(int,int);              // Première définition
long max(int,int);             // Erreur: les arguments existent déjà
double max(double,double);     // ok, deuxième définition
void max();
int max(int,int,int,int);
...
```

Bien sûr, chaque surcharge doit fournir sa propre définition de fonction. Un appel à ce type de fonction doit se faire sans ambiguité. Autrement, le compilateur génère une erreur. On peut souvent faire appel aux conversions explicites pour remédier à ce problème.

Exemple:

```
int i=0;
double d=2.0;
...
max(i,10);                     // ok: appel de max(int,int)
max(i,10.0);                   // Erreur: ambigu car 10.0 est par défaut un double
max(double(i),10.0);           // ok:conversion explicite,appel de max(double,double)
max(i,d);                      // Erreur: ambigu car i est int et d est double
max(double(i),d);              // ok: appel de max(double,double);
max(i,int(d));                 // ok: appel de max(int,int)
```

6.6 Fonction générique (`template`)

Une fonction générique est utilisée lorsque les mêmes fonctionnalités d'une fonction (le code d'une fonction) s'applique exactement de la même façon à des types différents. Une fonction générique possède une liste d'arguments et un type de retour ayant un certain type non prédéfini.

Exemple:

```
template <class Type> Type max(Type x, Type y) { ... }
```

Cet exemple illustre bien la syntaxe à utiliser. D'abord le mot-clé `template` doit débuter la définition ou le prototype. Il est suivi de la liste de paramètres formels indiquant les différents identificateurs qui seront utilisés comme type à l'intérieur de cette fonction. Elle est obligatoire, ne peut pas être vide et est comprise entre les symboles '<' et '>'. Chaque identificateur doit être unique et précédé du mot-clé `class`. Ces identificateurs doivent se retrouver en tant que type pour les arguments de la fonction. On ne peut pas spécifier un type de retour avec un certain identificateur qui ne se retrouve pas dans la liste d'arguments.

Exemple:

```
// Erreur: le type A n'est pas dans la liste d'arguments.
template <class A, class B> A min(B x, B y);

// Erreur: mot-clé class manquant avant B.
template <class A, B> A min(A x, B y);

// ok.
template <class A, class B> A min(A x, B y);
```

À l'appel d'une fonction générique, les types paramétrés (A et B dans cet exemple) prendront les types des arguments passés.

Exemple:

```
int i;
double d;
min(i,d);       // La fonction prend la forme: int min(int,double);
min(d,i);       // La fonction prend la forme: double min(double,int);
```

À l'intérieur de la définition de la fonction, chaque référence à un type formel est changé pour prendre le type indiqué par l'argument à l'appel.

Exemple:

```
template <class A, class B> A min(A x, B x)
{
   A var1;              // Si un appel à cette fonction est effectuée avec un
   ...                  // premier argument de type int, var1 sera de type int.
   return var1;
}
```

Une fonction générique agit de la même façon qu'une autre fonction. Elle peut être une macro (inline) ou tout autre forme de fonction en autant que le mot-clé modificateur soit placé après la liste de paramètres formels.

Exemple:

```
// Erreur: mot-clé mal placé
inline template <class A> A min(A x, A y) {...}

// ok
template <class A> inline A min(A x, A y) {...}
```

Une fonction générique peut aussi être surchargée. Il suffit que les types d'arguments ne laissent aucune ambiguïté de la même façon que les fonctions normales (voir section précédente).

Exemple:

```
template <class A> A min(A, A);
template <class A> A min(A, int);          // ok
template <class B> B min(B, B);            // Erreur: ambigu avec min(A,A).
template <class B> B min(B, double);       // ok
```

On peut aussi surcharger une fonction générique en la spécialisant. Une **spécialisation** se fait comme une deuxième définition de la fonction d'une façon normale.

Exemple:

```
template <class A> A min(A, A);
int min(int,int);
```

Lors de l'appel de la fonction, la fonction spécialisée a priorité sur la fonction générique.

Lors de la définition d'une fonction générique, le compilateur ne l'interprète pas immédiatement. En effet, il doit attendre un appel explicite avant d'être en mesure de compiler la fonction avec les bons types. C'est pourquoi, lors d'un appel à une fonction générique, cette même fonction doit être définie à l'intérieur de ce fichier. Les problèmes surviennent lorsqu'un même programme utilise plusieurs fichiers. Si la même fonction est utilisée avec les mêmes arguments (donc les mêmes types) dans deux fichiers différents, on se retrouve avec deux fonctions équivalentes de même nom et l'éditeur de liens à la fin de la compilation génère une erreur ou un avertissement. On se retrouve donc avec du code en double dans la version exécutable du programme. Ceci s'applique si la définition de la fonction générique est incluse dans les deux fichiers.

Un deuxième problème peut survenir si la définition est incluse dans un seul fichier et que plusieurs fichiers l'utilisent. En effet, le compilateur effectuera son travail pour les fichiers ne contenant pas la définition et considérera les appels aux fonctions génériques comme des fonctions externes existantes. Il n'y aura pas de problèmes si le compilateur a généré une fonction avec les mêmes types pour un autre fichier. Par contre, l'éditeur de liens verra une erreur si aucune fonction n'a été créée ailleurs pour le type de données spécifié à l'appel de cette fonction.

Il faut donc être vigilant à l'utilisation de ce genre de fonction.

Note: Le comité ANSI C++ continue toujours un débat intensif et passionné sur la façon d'implanter les fonctions génériques. Il est donc fortement recommandé de ne pas utiliser ce type de fonction pour éviter des problèmes de compatibilité entre compilateurs.

6.7 Pointeurs aux fonctions.

On peut déclarer une variable pour pointer à une fonction. Ceci se nomme un appel indirect à une fonction. La variable en question doit être spécifiée avec exactement le même type de retour et les mêmes arguments. On définit un pointeur à une fonction tout comme un prototype sauf qu'on doit associer l'opérateur de déréférence ('*') au nom de l'identificateur.

Exemple:

```
int min(int,int);
int *pf(int,int);              // Erreur: considéré comme une fonction à part
                               // entière nommée pf
int (*pf)(int,int);            // ok: variable pf pointe à une fonction de type
                               // int avec 2 arguments du même type
```

La façon d'initialiser ou d'affecter est simple, il suffit de l'affecter au nom de n'importe quelle fonction ayant exactement le même canevas (type de retour, type des arguments et nombre d'arguments).

Exemple:

```
int (*pf)(int,int) = min;      // Initialisation;
void (*pf2)(int,int);
int (*pf3)(int,int,int);
...
pf = min;       // Affectation.
pf2 = pf;       // Erreur: Type de retour différent.
pf3 = pf;       // Erreur: nombre d'arguments différents.
```

Pour invoquer un appel indirect, on procède exactement de la même façon qu'un appel direct.

Exemple:

```
int i=0, j=0;
...
min(i,j);       // Appel direct
pf(i,j);        // Appel indirect
```

L'appel indirect de la forme:

```
pf(i,j);
```

est exactement l'équivalent de la forme plus longue:

```
(*pf)(i,j);
```

7 Entrées / Sorties (streams)

Les fonctionnalités d'entrées/sorties du C++ ne font pas partie du langage mais sont fournies dans une librairie standardisée.

Cette librairie est implantée sous forme de classes avec plusieurs méthodes disponibles ainsi que les opérateurs '<<' et '>>'. Chaque opérateur "pointe" vers sa destination. L'opérateur '>>' est implanté sous la forme:

```
>> x
```

Cette syntaxe indique une entrée dans x tandis que:

```
<< x
```

indique que les informations sortent de x.

Cette librairie est utilisée pour les interactions avec les périphériques (fichiers, écran, clavier). On peut aussi définir ses propres classes pour une communication avec un port série, imprimante, etc...

Ce chapitre se veut un résumé très bref des fonctionnalités d'entrées/sorties. Pour une discussion plus complète et une référence sur toutes les possibilités de cette librairie, consultez le manuel de référence du compilateur utilisé.

7.1 Sortie

L'affichage à l'écran se fait à l'aide de l'entité `cout`. Cette dernière est une variable globale du type sortie (`ostream`). Chaque type de base est intégré sous l'opérateur '<<' et l'affichage de texte ou de valeurs peut se faire sans formattage spécifique (il existe par défaut).

Exemple:

```
int i = 43;
char *s = "Bonjour!";
...
cout << s << i;
```

On obtient à l'écran:

```
Bonjour!43
```

Les constantes littérales peuvent aussi être en argument de l'opérateur. On remarque également que puisque l'opérateur '<<' est un opérateur avec associativité à gauche, les arguments sont traités à partir de la gauche (il ne faut pas se laisser confondre par l'orientation de l'opérateur '<<' ou '>>').

Exemple:

```
cout << s << " " << 42 << " i = " << i << '\n';
```

On obtient à l'écran:

```
Bonjour! 42 i = 43
```

Plusieurs manipulateurs sont disponibles pour modifier le format par défaut de l'affichage. Un modificateur est manipulé de la même façon qu'une variable.

Manipulateur	Utilité	Disponibilité
dec	Format numérique décimal	Entrée/Sortie
endl	Écrit une fin de ligne	Sortie
ends	Écrit un caractère nul ('\0')	Sortie
flush	Vide le tampon mémoire	Sortie
hex	Format numérique hexadécimal	Entrée/Sortie
oct	Format numérique octal	Entrée/Sortie
unsetf(long *f*)	Désactive les indicateurs spécifiés dans *f*.	Entrée/Sortie
fill(int *ch*)	Initialise le caractère de remplissage à *ch*	Entrée/Sortie
setf(long *f*)	Active les indicateurs spécifiés dans *f*.	Entrée/Sortie
precision(int *p*)	Initialise la précision des points flottants (nombre de décimales) à *p*.	Entrée/Sortie
width(int *w*)	Initialise la largeur du champ d'affichage à *w*.	Entrée/Sortie

Exemple:

```
int i = 36;
cout << hex << i << endl << oct << i << endl << dec << i << endl;
cout.precision(2);
cout << 20.384 << endl
cout.width(10);
cout << i;
```

On obtient à l'écran:

```
24
44
36
20.38
36
```

On écrit dans un fichier exactement de la même façon. On doit d'abord par contre se définir une variable de type `ofstream` pour un fichier d'écriture seulement. Les indicateurs `ios::out` et `ios::app` donnent le mode d'ouverture du fichier.

Exemple:

```
ofstream fichier;
...
fichier.open("nom.dat",ios::out);
fichier << "Allo" << endl;
fichier.put('c');
fichier.close();
```

Dans le fichier, on a:

```
Allo
c
```

L'ouverture d'un fichier peut aussi se faire directement au constructeur.

Exemple:

```
ofstream fichier("nom.dat",ios::app);
```

On peut vérifier directement la variable pour tester si le fichier a été correctement ouvert.

Exemple:

```
if(!fichier) // traiter l'erreur.
```

Note aux habitués du C: Cette librairie a été instaurée pour éviter tout appel aux fonctions C du type `printf()`. En effet, on vous rappelle que ce type de fonction n'effectue aucune vérification sur les types à traiter. Il est donc fortement recommandé d'éviter l'utilisation de ce type de fonction. Les sorties standards du type `stdout`, `stderr`, `stdin` ont été remplacées par `cout`, `cerr` et `cin` respectivement.

7.2 Entrée

Le fonctionnement des entrées est exactement le même que pour les sorties à une différence près puisque l'on utilise l'opérateur '>>'. Pour saisir les entrées au clavier, on utilise la variable globale `cin`.

Exemple:

```
int i;
...
cout << "Entrez la variable i : ";
cin >> i;
cout << endl << "Variable lue: " << i;
```

Résultat:

```
Entrez la variable:           // on attend une réponse du clavier
                              // (jusqu'à la touche retour)
Entrez la variable: 54
Variable lue: 54
```

Les fichiers se lisent aussi de la même façon. Une variable pour un fichier est du type `ifstream`. L'ouverture se fait également avec un constructeur ou la méthode `open()` avec comme indicateur `ios::in` pour indiquer une lecture.

Les manipulateurs disponibles sont indiqués dans la table de la section précédente.

8 Les classes

La section qui suit décrit les particularités des classes, soit la description et l'utilisation dans le milieu de la programmation.

Qu'est-ce qu'une classe ?

- Une classe définit un groupe d'objets similaires.
- Une classe est définie par des attributs (variables) et par des méthodes (fonctions) associées.
- Une classe est décrite par un nom qui devient un nouveau type.
- Une classe est décomposée en blocs privés (`private`), protégés (`protected`) et publics (`public`). Ces blocs permettent l'encapsulation des données.
- Une classe peut hériter des attributs et/ou des méthodes d'une ou de plusieurs classes.

Les classes sont la force de la programmation objet, car elles permettent de modulariser l'information.

Définition:

```
class nom-classe
{
   // attributs et méthodes
};
```

8.1 Terminologie

- Membres d'une classe: les membres d'une classe sont les attributs et les méthodes de cette classe.
- Encapsulation des données: les données et le code associé sont isolés dans une même structure.
- Polymorphisme: la même méthode peut s'appliquer à plusieurs classes.
- Classe: une classe est à la fois un module et un type. Une classe définit un groupe d'objets similaires.

- Objet: un objet est une instance de classe créée à l'exécution. La nature et les actions d'un objet sont définies par sa classe.
- Méthode: une méthode peut être une procédure ou une fonction définie à l'intérieur d'une classe.
- Attribut: variable définie dans une classe.
- Surcharge: plusieurs méthodes d'une classe peuvent partager le même nom.
- Partie publique: la partie publique de la définition est accessible par les classes clients et les héritiers.
- Partie privée: la partie privée de la définition est accessible seulement par les objets de cette classe.
- Client - Fournisseur: le client est celui qui fait la demande d'une variable ou l'exécution d'une routine. Le fournisseur est celui qui reçoit le message et exécute la demande.
- Héritage: le principe est de décrire une nouvelle classe non pas à partir de zéro, mais à partir d'une extension ou d'une spécialisation d'une classe - ou à partir de plusieurs classes dans le cas de l'héritage multiple. Les classes définies après la classe principale vont hériter des caractéristiques de la classe principale de façon hiérarchique.
- Agrégation: l'agrégation est une relation de composé-composant entre deux classes. Le composant "fait partie du" composé.

8.2 Les attributs

Les attributs sont des variables définies dans une classe. Les attributs contiennent l'information pertinente à la représentation d'une classe.

Exemple:

```
class Noeud
{
   int x; // position en X
   int y; // position en Y
   int z; // position en Z
};
```

Une classe peut être le type d'un attribut (agrégation), si et seulement si, il a été préalablement défini.

Exemple:

```
class Coordonnees
{
   int x, y, z;
};

class Noeud
{
   Coordonnees position;
};
```

8.2.1 Attributs statiques

Un attribut peut être déclaré statique. Ceci permet d'avoir une seule copie de l'attribut en mémoire et non une copie par objet. Cette variable peut être un indicateur ou un compteur relié à la classe et aux objets de celle-ci. Par le fait qu'elle occupe une place en mémoire, il est possible d'y avoir accès:

```
nom-classe :: identificateur
```

Exemple:

```
class ListeElement
{
public:
   static int nombre;
};

int ListeElement::nombre = 1;

main()
{
   ListeElement::nombre = 5;
   cout << "nombre = " << ListeElement::nombre << endl;

   ListeElement elem1;

  elem1.nombre = 7;
  cout << "nombre = " << elem1.nombre << endl;
}
```

L'exemple démontre qu'il y a deux façons d'avoir accès à la variable statique, soit par un des objets de la classe (`elem1.nombre`) soit directement par la classe (`ListeElement::nombre`). Dans ce cas, il faut une initialisation et une définition de l'attribut à l'extérieur de la classe.

8.3 Les méthodes

Les méthodes sont des procédures ou fonctions définies pour une classe. Les méthodes sont le traitement de l'information associée aux attributs.

Exemple:

```
class Noeud
{
   int x, y, z;
public:
   int positionX();   // déclaration de la méthode
};

int Noeud::positionX()   // définition de la méthode
{
   return(x);
}

main()
{
  Noeud n1;
  int   valeur;
  valeur = n1.positionX();  // Accès à la méthode
}
```

Généralement, on retrouve seulement le prototype de la méthode dans la définition de la classe. Mais il est aussi possible de retrouver la définition de la méthode à même la définition de la classe. Ceci est une bonne pratique seulement pour les méthodes courtes à exécution rapide. Une méthode de plus d'une ligne devrait être définie à part entière.

8.3.1 Méthodes macros (inline)

Une méthode peut être déclarée macro pour avoir une plus grande vitesse d'exécution. Il y a deux façons de déclarer une méthode macro: premièrement, si elle est définie à l'intérieur d'une classe et deuxièmement, si elle est déclarée avec le mot réservé `inline` pour une définition externe à la classe.

Exemple:

```
class Noeud
{
   int x, y, z;

   int positionX() { return(x); }; // macro à la compilation
   inline int positionY();
};

inline int Noeud::positionY()
{
   return(y);
}

main()
{
   Noeud n1;
   int   x1, y1;

   x1 = n1.positionX();
   y1 = n1.positionY();
}
```

Le seul problème dans cet exemple vient du fait que, si la méthode est trop longue, le compilateur considère ces définitions comme toute autre fonction.

8.3.2 Méthodes statiques (static)

Une méthode peut être déclarée statique. Ceci permet d'avoir accès à la méthode sans avoir d'objet. Mais la méthode statique qui n'est pas contenue par un objet n'a accès qu'aux attributs statiques de sa classe.

Exemple:

```
class ListeElement
{
public:
   static int nombre;
   void initialisation() { nombre = 5; }
   static int nombreElement() { return(nombre);}
};

int ListeElement::nombre = 7;

main()
{
   int n;

   n = ListeElement::nombreElement();
   cout << "nombre : " << n << endl;

   ListeElement elem1;

   elem1.initialisation();
   n = elem1.nombreElement();
   cout << "nombre : " << n << endl;
}
```

Une méthode statique a accès à tous les membres si elle possède un objet de sa classe.

8.3.3 La surcharge d'une méthode

La surcharge (overloading) permet de créer plusieurs méthodes avec le même nom, si et seulement si les arguments sont différents. Bien sûr, chaque surcharge doit avoir sa propre définition. L'algorithme de surcharge est fait à la compilation.

Exemple:

```
void fonction(int);
void fonction(int, char);
void fonction(double);
```

8.4 Visibilité à l'intérieur d'une classe (publique, privée, protégée)

Pour diminuer la corruption de l'information, les classes permettent de restreindre l'accès à ses membres (attributs et méthodes). Ceci s'appelle l'encapsulation des membres. Il y a trois blocs de visibilité possible dans une classe:

- La partie privée (`private`) de la définition est accessible seulement par les membres de la classe.
- La partie protégée (`protected`) de la définition est accessible seulement par les membres de la classe et les objets des classes héritières.
- La partie publique (`public`) de la définition est accessible par tous.

Il faut prendre note que par défaut les membres d'une classe sont privées.

Exemple:

```
class Element_T3
{
   int       numero; // partie privée
public:
   void afficher_information(); // partie publique
   void dessiner();
};
```

8.5 Méthodes amis (friend)

Dans certains cas, la visibilité d'une méthode est trop restrictive. Alors, il est possible pour une méthode ne faisant pas partie de la classe d'avoir accès aux attributs ou aux méthodes définies dans la partie privée de la classe, ceci en la déclarant `friend` soit ami de la classe.
La raison de l'utilisation de `friend` est pour permettre à deux classes d'utiliser la même méthode.

Exemple:

```
class Matrice;
class Vecteur
{
   int *p;
   int dim;
   friend Vecteur mult(Vecteur &v, Matrice &m);
public:
   int bs;
   Vecteur() { dim = 10; p = new int[dim]; bs = dim - 1; };
   Vecteur(int n);
   ~Vecteur() { delete p; };
   int& element(int i);
};

Vecteur::Vecteur(int n)
{
   if (n <= 0)
      exit(1);
   dim = n;
   p = new int[dim];
   bs = dim - 1;
}

int& Vecteur::element(int i)
{
   if ( i < 0 || i > bs )
      exit(1);
   return( p[i]);
}


class Matrice
{
   int **p;
   int dim1, dim2;
   friend Vecteur mult(Vecteur &v, Matrice &m);
public:
   int bs1, bs2;
   Matrice(int d1, int d2);
   ~Matrice();
   int& element(int i, int j);
};
```

```
Matrice::Matrice( int d1, int d2)
{
   if( d1 < 0 || d2 < 0 )
      exit(1);
   dim1 = d1;
   dim2 = d2;
   p = new int*[dim1];
   for (int i=0; i < dim1; i++)
      p[i] = new int[dim2];
   bs1 = dim1 - 1;
   bs2 = dim2 - 1;
}

Matrice::~Matrice()
{
   for(int i = 0; i <= bs1; ++i);
      delete p[i];
   delete p;
}

int& Matrice::element( int i, int j)
{
   if (i < 0 || i > bs1 || j < 0 || j > bs2)
      exit(1);
   return (p[i][j]);
}

Vecteur mult(Vecteur& v, Matrice& m)
{
   if (v.dim != m.dim1)
      exit(1);
   Vecteur rep(m.dim2);
   for (int i = 0; i <= m.bs2; ++i)
   {
      rep.p[i] = 0;
      for ( int j = 0; j <= m.bs1; ++j )
      {
         rep.p[i] += v.p[j] * m.p[i][j];
      }
   }
   return(rep);
}

main()
{
   Vecteur vec(5);
   Matrice mat(5,5);

   for (int i = 0; i <= mat.bs2; ++i)
   {
      vec.element(i) = i;
      for (int j = 0; j <= mat.bs1; ++j)
      {
         mat.element(j,i) = j + i;
      }
   }

   mult( vec, mat);
}
```

L'utilisation de `friend` est controversée, car elle brise l'encapsulation des membres de la classe. Ce mécanisme peut être très utile, mais il est facile d'en abuser et de rendre le code difficile à maintenir.

8.6 Constructeurs et Destructeurs

8.6.1 Constructeur

Un constructeur est une méthode de la classe qui porte le même nom que la classe à laquelle elle se rapporte. Le constructeur est utilisé pour l'initialisation des attributs et pour l'allocation de la

mémoire de manière automatique. L'appel du constructeur se fait à la définition d'un objet de cette classe.

L'ordre d'appel des constructeurs dans le cas d'une classe a une ou plusieurs classes de base: le constructeur de la classe de base est appelé avant celui de la classe héritière.

Le constructeur peut être surchargé, il peut y avoir plusieurs constructeurs avec des paramètres différents.

Le constructeur ne peut pas retourner un type ou une valeur de retour.

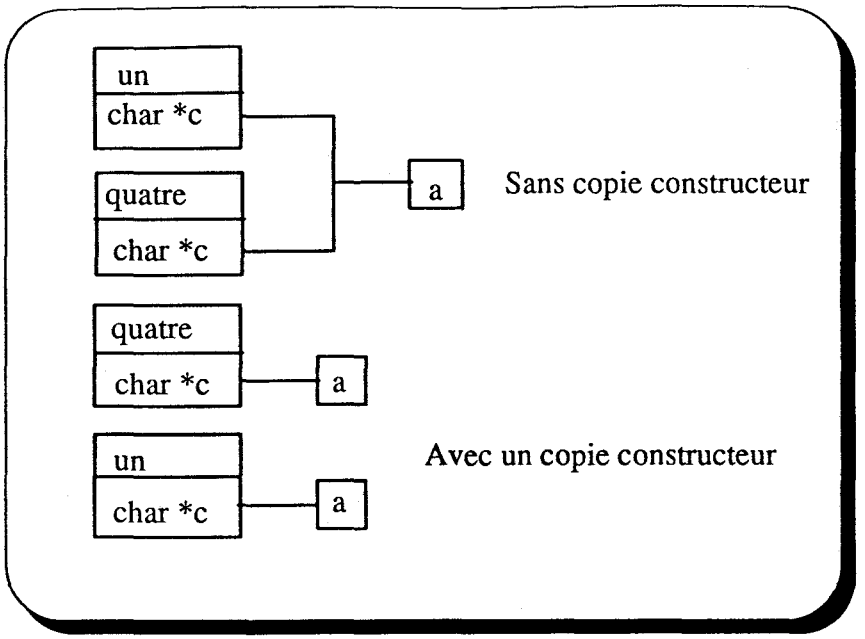
Exemple:

```
class X
{
   char *c;
public:
   X() { c = new char[2]; c[0] = 'a';}
   X(char x) { c = new char [2]; c[0] = x; }
   X(const X& x) {c = new char[2]; c[0] = x.c[0];}
   char& valeur() { return(c[0]); }
};


main()
{
   X un;
   X deux('b');
   X trois = 'c';
   X quatre = un;
   X cinq(deux);

   cout << "un = " << un.valeur() << endl;
   cout << "deux = " << deux.valeur() << endl;
   cout << "trois = " << trois.valeur() << endl;
   cout << "quatre = " << quatre.valeur() << endl;
   cout << "cinq = " << cinq.valeur() << endl;
}
```

Cette exemple fait appel à trois constructeurs différents. Lors de l'initialisation de la variable `un` le constructeur par défaut sera appelé (`X::X()`). Dans le cas de la variable `deux` et `trois`, le constructeur (`X::X(int)`) sera appelé et ceci de manière identique. Dans le cas de la variable `quatre` et `cinq`, le copie constructeur (`X::X(const X&)`) sera appelé et aussi de manière identique. Le copie constructeur est quelque fois nécessaire pour une initialisation utilisant des pointeurs, pour permettre d'occuper un espace mémoire distinct.

Exemple:

```
main()
{
   X un;
   X quatre = un;
}
```



8.6.2 Destructeur

Un destructeur est une méthode de la classe qui porte le même nom que cette classe et qui précédé par le caractère "~".
Le destructeur est un mécanisme complémentaire au constructeur, qui automatise la désallocation.
L'appel du destructeur se fait de manière automatique lorsque le programme atteint l'extérieur du bloc d'appel de la variable (voir déclaration de variable).

Exemple:

```
class Stack
{
   char *s;
   int  maxlen;
public:
   Stack(int size) {   s = new char[size];
                       maxlen = size;}        // constructeur
   Stack() {    s = new char[100];
                maxlen = 100; }               // constructeur
   ~Stack() { delete s; }                     // destructeur
};
```

8.7 Le pointeur implicite "this"

Le mot réservé `this` est une déclaration implicite d'un pointeur référencé à l'objet qu'il représente. Le pointeur `this` est surtout utile comme valeur de retour pour les opérateurs ou les méthodes.

Exemple:

```
class DListe
{
   DListe   *avant, *apres;
public:
   void ajouter( DListe *);
}

void DListe::ajouter( DListe *ptr)
{
   ptr->apres = apres;
   ptr->avant = this;
   apres->avant = ptr;
   apres = ptr;
}
```

8.8 Concepts d'affectation et d'initialisation

Dans cette section il faut démystifier la différence entre deux concepts: l'initialisation et l'affectation. Une initialisation est effectuée lors de la définition d'une variable. Cette initialisation fait appel au copie constructeur. Une affectation est par contre une opération à part entière et est effectuée alors que la variable est déjà définie.

Exemple:

```
class String
{
...
public:
   String(int);                          // constructeur
   String(const &String);                //  copie constructeur
   operator=(const &String);
   ~String();
};

main()
{
   String s1(10), s2(20);                // constructeur
   String s3 = s1;                       // copie constructeur
   s1 = s2;                              // affectation
}
```

L'affectation doit appeler le destructeur de s1 avant d'allouer de l'espace et copier s2.
Le constructeur par copie est appelé lors d'une initialisation pour les arguments des fonctions ainsi que pour les valeurs retournées.

8.9 Surcharge Opérateur

Une classe permet de redéfinir des opérateurs qui fonctionnent avec les objets de la classe. La surcharge d'un opérateur se fait à l'aide du mot réservé `operator`. La surcharge d'un opérateur permet de définir les opérateurs de manière spéciale relativement à une classe spécifique. Pour surcharger un opérateur, on doit le définir dans la classe à laquelle il s'applique. La surcharge peut être faite sur des opérateurs binaires ou unitaires (voir liste des opérateurs disponibles section 4.1). Il faut prendre note que la syntaxe et l'ordre de priorité des opérateurs

demeurent inchangés et que seulement les opérateurs prédéfinis par le C++ peuvent être surchargés. Un opérateur a toujours un argument implicite qui est l'opérande de gauche. À la définition des méthodes des opérateurs binaires, on a un seul argument qui est l'opérande de droite. Dans le cas des opérateurs unitaires, aucun argument n'est nécessaire.

Opérateur surchargeable

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

Exemple:

```
class Trois_D
{
   int x,y,z; // Coordonnée 3-D
public:
   Trois_D operator+(Trois_D t);
   Trois_D& operator=(Trois_D t);
   Trois_D& operator++(void);
   void voir(void);
   void assigner( int mx, int my, int mz);
};

// Surcharge de + d'un opérateur binaire
Trois_D Trois_D::operator+(Trois_D t1)
{
   Trois_D temp;

   temp.x = x + t.x;
   temp.y = y + t.y;
   temp.z = z + t.z;
   return temp;
}

// Surcharge de =
Trois_D& Trois_D::operator=(Trois_D t)
{
   x = t.x;
   y = t.y;
   z = t.z;
   return *this;
}

// Surcharge de ++ d'un opérateur unitaire
Trois_D& Trois_D::operator++()
{
   x++;
   y++;
   z++;
   return *this;
}

// Voir les coordonnées x,y,z
void Trois_D::voir(void)
{
   cout << x << "," << y << "," << z << "\n";
}

// Assigner les coordonnées
void Trois_D::assigner(int mx, int my, int mz)
{
   x = mx;
   y = my;
   z = mz;
}
```

```
main()
{
   Trois_D     a, b, c;

   a.assigner(1, 2, 3);
   b.assigner(10, 10, 10);

   a.voir();
   b.voir();

   c = a + b;
   c.voir;

   c = b = a;
   c.voir();
   b.voir();
}
```

Résultat du programme:

```
1, 2, 3
10, 10, 10
11, 12, 13
1, 2, 3
1, 2, 3
```

8.9.1 Surcharge de l'opérateur [ ]

La surcharge de l'opérateur [] est très utile. Il permet de rendre l'allocation de vecteur plus sécuritaire. La surcharge de cette opérateur peut avoir une valeur de retour en utilisant &.

Exemple:

```
class Vecteur
{
   int   *p;
   int   dim;
public:
   Vecteur(int x) {dim = x; p = new int[dim];}
   ~Vecteur() {delete[] p;}
   int&    operator [] (int i);
};

int&   Vecteur::operator [] (int i)
{
   return (p[i]);
}

main()
{
   Vecteur vect(5);

   vect[2] = 10;
}
```

8.10 Héritage

La programmation orientée-objet utilise des classes pour définir une application. L'héritage est un lien qui définit la relation d'appartenance entre deux classes. Le lien d'héritage permet aux membres d'une classe d'être utilisés dans une autre classe.

La relation d'héritage sur les classes permet de:

- Spécialiser et généraliser les classes.

- Réutiliser les définitions des classe
- Grouper les aspects communs à plusieurs classes

Définition:

```
class nom-classe: (public|private|protected) nom-classe-base
{
   déclarations
};
```

Les deux points séparent le nom de la classe héritière (nom-classe) du nom de la classe de base (nom-classe-base). Entre les deux points et la classe de base, il y a la visibilité des membres hérités. Si la visibilité est publique, cela signifie que les membres publics de la classe de base seront publics dans la classe héritière. Si par contre la visibilité est privée, cela signifie que les membres publics de la classe de base seront privés dans la classe héritière (voir tableau ci-dessous). Généralement, la visibilité des membres hérités sera public.

Tableau: Visibilité des membres vs la visibilité de l'héritage

Membres\Héritage	Public	Privé	Protégé
Publics	publics	privés	protégés
Privés	pas accès	pas accès	pas accès
Protégés	protégés	privés	protégés

8.10.1 Extension

L'extension est la prolongation de la classe de base. Cela permet à la classe héritière d'ajouter de nouvelles caractéristiques (attributs) et/ou de nouvelle fonctionnalités (méthodes) à la classe de base.

Exemple:

```
class Personne
{
   char *nom;
public:
   Personne(char *n) { nom = n; }; // Constructeur
   void print() { cout << "Nom: " << name << endl; }
};
class Enseignant: public Personne
{
   int numero_cours;
   int maxCours;
public:
   Enseignant( char *nom): Personne(nom);
};
```

Note: Les constructeurs et les destructeurs d'une classe de base ne sont pas hérités.

8.10.2 Spécialisation

La spécialisation est une prolongation de la classe de base tout comme une extension, mais tout en spécialisant certaines fonctionnalités (méthodes).

Exemple:

```
class Etudiant: public Personne
{
public:
   Etudiant( char *nom): Personne(nom){}; // Constructeur
   void print();
};

void Etudiant::print()
{
   printf("Etudiant: ");
   Personne::print();   // appel de la méthode "print" de la classe de base Personne
}
```

8.10.3 Héritage multiple

L'héritage n'est pas limité a une classe de base. Il est possible qu'une classe puisse hériter de plusieurs classes.

Exemple:

```
class Enseignant
{
   ...
};
class Chercheur
{
   ...
};
class Professeur: public Enseignant, public Chercheur
{
   ...
};
```

8.11 Méthodes virtuelles (virtual)

La surcharge d'une méthode se fait par un algorithme de différence de type et ceci est fait à la compilation. Si le mot réservé `virtual` est utilisé devant une méthode, alors la méthode est choisie de manière dynamique lors de l'exécution. Les méthodes virtuelles sont pointées par une table associée aux objets. La méthode appelée dépend du type spécifique de l'objet. De cette façon, les méthodes virtuelles permettent aux classes héritières d'avoir différentes versions de la méthode associée à la classe de base. Il faut prendre note que généralement les méthodes virtuelles décrites dans la classe de base ne seront jamais utilisées. Une méthode virtuelle ne peut pas être statique, mais elle peut être amie `(friend)`. Un constructeur ne peut pas être virtuel, mais les destructeurs le peuvent.

Exemple:

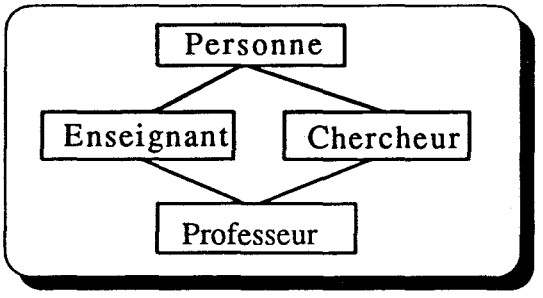
```
class Element
{
public:
   virtual void afficher();
};

class ElementT3:public Element
{
public:
   void afficher();
};
```

8.12 Classe de base virtuelle (virtual)

Lorsque qu'il y a de l'héritage multiple, une classe de base peut être spécifiée plus d'une fois. Alors une classe héritière pourrait avoir la même classe de base plusieurs fois. Pour ne pas avoir de problèmes de compilation, il faut que la classe porte le préfixe `virtual` virtuelle.

Exemple:

Diagramme de relation des classes




```
class Personne
{
   ...
};

class Enseignant:public vitual Personne
{
   ...
};

class Chercheur:public virtual Personne
{
   ...
};
class Professeur: public Enseignant, public Chercheur
{
   ...
};
```

8.13 Classe abstraite

La classe abstraite est une classe qui définit des caractéristiques (attributs) et des méthodes qui s'appliquent à plusieurs classes héritières. La classe abstraite n'est pas utilisée comme une entité (objet) en tant que telle, mais elle sert de base et de définition pour les classes héritières.

Exemple:

```
class Figure
{
   point centre;
   ...
public:
   virtual void rotation(int)= 0;
   virtual void dessiner() = 0;
}
class Cercle:public Figure
{
   int rayon;
public:
   void rotation(int);
   void dessiner();
};
```

La classe `Figure` est abstraite, car au moins une de ses méthodes n'est pas définie. Alors, il n'est pas possible d'avoir un objet de cette classe.

8.14 Classe générique

Une classe générique permet de définir une classe et de l'utiliser avec plusieurs types de données, au lieu de définir la même classe pour les différents types de données.

Exemple:

```
template <class Type> class Vecteur
{
   Type        *data;
   int         dim;
public:
   Vecteur(int);
   ~Vecteur()   {delete[] data;};
   Type&   operator[](int i) {return data[i];};
};

template <class Type> Vecteur<Type>::Vecteur(int n)
{
   data = new Type[n];
   dim = n;
};

main()
{
   Vecteur<int> x(5); // initialisation d'un vecteur de 5 entiers
}
```

L'utilisation de classe générique est possible dans le cas de petit programme à fichier unique. Car la définition et la déclaration de la classe doit être effectuée dans le même fichier. Il faut

prendre note que les classes génériques ne sont pas défini par la norme ANSI C++. Ceci rend son utilisation non-portable.

8.15 Classe dans une classe

Une classe peut être contenu dans une classe. L'utilisation de cette pratique est réservé à l'encapsulation d'une classe par une autre, alors la classe encapsulée est invisible aux autres classes. Ceci n'est cependant pas un bon style de programmation, car le code peu devenir illisible.

Exemple:

```
class Liste
{
   class ListeItem
   {
   public:
        ListeItem(int val = 0) {...}
   }
}
```

8.16 Union

L'union est une partie spéciale de la classe. Cette structure est utilisée pour sauver de l'espace. L'espace mémoire utilisée par l'union est l'espace nécessaire pour contenir la donnée du type le plus long. Chacun des membres de la structure débute à la même adresse mémoire. Un seul attribut peut être conservé à la fois. Dans l'exemple qui suit l'espace réservé en mémoire sera équivalent au type `double`.

Exemple:

```
class Jeton
{
public:
   int numero;
   union {
      char    cval;
      char    *sval;
      int     ival;
      double  dval;
   } val;
}

main()
{
   Jeton jeton;

   jeton.val.dval = 99.9;
}
```

8.17 Structure

La structure est une définition similaire à la classe. La grande différence vient du fait que les membres (attributs et méthodes) sont publics par défaut. La structure ne devrait pas être utilisée dans la programmation par objet, car elle ne fait ni plus ni moins qu'une classe.

Exemple:

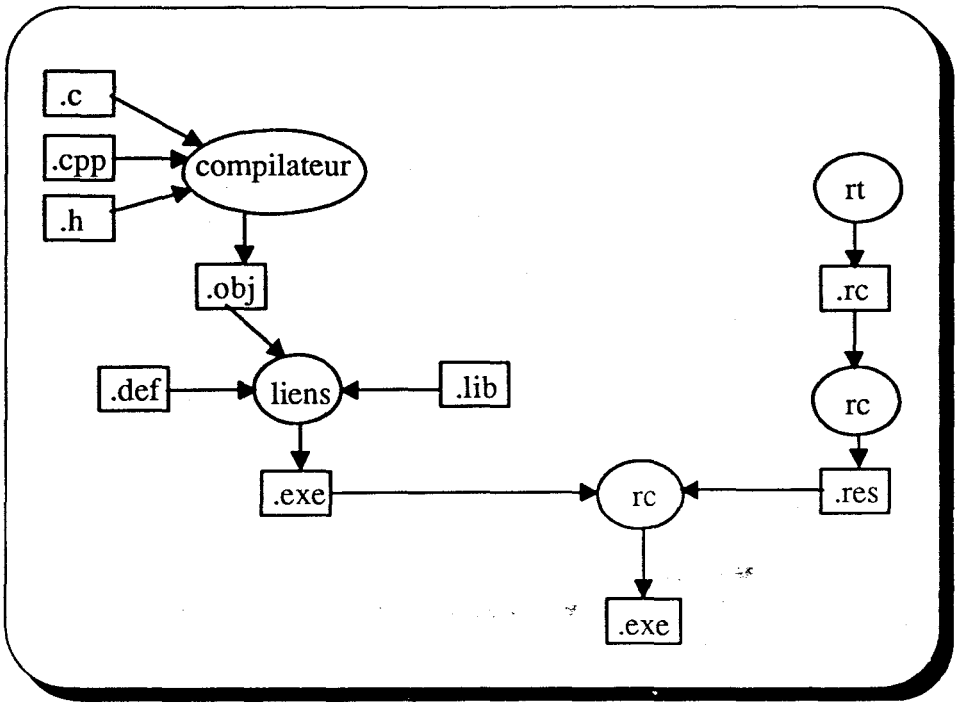
```
struct Jeton {
   int jeton;
   int  ival;
};

typedef struct Jeton jeton;

main()
{
   jeton jet;
   jet.ival = 10;
}
```

9 Environnement

9.1 Présentation des fichiers

Il y a deux types de fichier qui sont utilisés. Les fichiers "Entête" qui décrivent et définissent les classes, alors que les fichiers "Sources" définissent les méthodes des classes. Chaque fichier "Entête" possède l'extension ".h". Chaque fichier "Sources" possède l'extension ".cpp". Il faut prendre note que l'on retrouve en général une seule classe par fichier.

9.2 Compilation avec C++



Un programme le moindrement complexe est habituellement décomposé en plusieurs fichiers sources (.cpp) et plusieurs fichiers entêtes (.h). Puisque les fichiers entêtes contiennent les énoncés déclaratifs, ils seront inclus dans chaque fichier source. La façon d'inclure un fichier d'entête est avec la syntaxe suivante:

```
#include "fichier.h"
#include <fichier.h>
```

La différence entre inclure un fichier entre guillemets ("") ou entre crochets (<>) existe uniquement dans l'indication des répertoires de recherche du fichier. Une notation entre guillemets cherche le fichier dans le répertoire courant d'abord et ensuite dans les répertoires d'inclusion (variable d'environnement INCLUDE) tandis qu'une notation entre crochets cherche le fichier dans les répertoires d'inclusion uniquement. De façon générale, les fichiers d'entêtes de système sont entre crochets et les fichiers propres au programme sont entre guillemets.

Le compilateur interprète donc chaque fichier source et, s'il compile sans erreur, génère un fichier objet (.obj) pour chacun d'eux. Soulignons le fait que le compilateur ne fait aucune différence entre un fichier entête ou un fichier source. C'est le préprocesseur qui, lors de l'inclusion d'un fichier, remplace l'énoncé `#include` par le contenu du fichier entête. Le compilateur n'y voit donc que du code.

Par la suite, l'éditeur de liens rassemble tous les fichiers objets avec les librairies pour créer l'exécutable final. Dans le cas d'un programme fonctionnant sous Windows ou PM, l'éditeur de liens considère un fichier de définition du programme (.def) et aussi un fichier de ressources contenant, en général, les icônes utilisés par le programme.

Le travail de l'éditeur de liens est de relier les fonctions, méthodes et variables globales entre les fichiers. En effet, puisque seulement les déclarations se retrouvent dans les fichiers entêtes, un fichier ne contenant pas la définition d'une fonction mais y faisant appel est relié à cette fonction lors de l'édition des liens seulement. À la compilation de ce fichier, le compilateur sait que cette fonction existe grâce à la déclaration et ne se soucie donc plus de sa définition.

9.2.1 Préprocesseur

Le préprocesseur est celui qui prépare un fichier source avant sa compilation. Ceci est effectué de façon transparente pour l'usager, mais plusieurs instructions sont disponibles et très pratiques. Cette section s'attarde sur les instructions les plus utilisées seulement.

D'abord, chaque instruction doit se retrouver sur la première colonne d'une ligne et doit débuter par le caractère #. Cette instruction monopolise la ligne où elle se trouve et aucune autre ne peut s'y trouver.

L'instruction `#define` défini un symbole quelconque pour représenter une valeur ou tout autre texte.

Exemple:

```
#define BLEU 28
#define CHAINE "Bonjour!"
```

Dans cet exemple, le préprocesseur enregistre les symboles et partout où il rencontre le symbole BLEU ou CHAINE, ils sont remplacés par leurs valeurs associés (28 ou "Bonjour!"). Un symbole peut être défini sans nécessairement avoir une valeur associée.

Exemple:

```
#define TURBO
```

Les instructions **#ifdef(#ifndef)-#endif** sont utilisées pour effectuer une compilation conditionnelle.

Exemple:

```
#ifdef BLEU
...     // partie 1
#else
...     // partie 2
#endif
...
#ifndef TURBO
...     // partie 3
#else
...     // partie 4
#endif
```

Dans cet exemple, si le symbole BLEU est défini, la partie 1 est compilée, la partie 2 est ignorée et vice-versa. Dans la même optique, si le symbole TURBO n'est pas défini, la partie 3 est compilée et la partie 4 est ignorée.

Enfin, rappelons que l'instruction pour les fichiers d'entêtes **#include** est traitée par le préprocesseur.

9.3 Variables externes et statiques

Une variable globale comme toute autre variable n'est définie qu'à un seul endroit dans le programme. Avec plusieurs fichiers, une variable globale doit être déclarée externe pour indiquer au compilateur que cette variable existe mais qu'elle est définie ailleurs. C'est encore l'éditeur de liens qui effectue la relation entre les accès à cette variable. La syntaxe est la suivante pour accéder une variable globale définie dans un autre fichier:

```
extern int varGlobale;
```

Rappelons que ceci n'est pas une définition mais bien une déclaration. La nuance entre les deux est claire, car une définition alloue l'espace mémoire nécessaire à la variable créée tandis qu'une déclaration ne fait qu'indiquer l'existence de cette variable.

Une variable globale peut être définie avec le mot-clé `static` pour indiquer que cette variable est globale mais seulement au fichier où elle est définie. Les autres fichiers n'y ont pas accès.

```
static int varGlobale;
```

Une variable peut également être statique à l'intérieur d'une fonction. La variable sera donc persistante en mémoire et gardera sa valeur d'un appel à l'autre de la fonction.

Exemple:

```
int min(int x, int y)
{
   static int i = 0;   // Initialisation
   i++;                // Compte le nombre d'appels à cette fonction
   if(x < y) return x;
   return y;
}
```

Dans cet exemple, à chaque appel de `min`, on incrémente le compteur. Ce dernier est toujours en mémoire et ne perd pas sa valeur lorsque l'on quitte la fonction.

10 Annexe A

Cette annexe contient tout le code source des exemples donnés lors de l'exposé du cours.

exemple.h

```
// Protection du fichier d'entête avec les directives du préprocesseur.
#ifndef __exemple_h
#define __exemple_h

// Définition des alias de types
typedef long Entier;
typedef Entier* EntierP;
typedef float Reel;

// Définition des constantes
const Entier TAILLE = 10;
const Entier COINX = 20;
const Entier COINY = 10;

enum Booleen { Faux, Vrai };

#endif
```

exemple.cpp

```
#include <iostream.h>
#include <stdio.h>
#include <new.h>
#include <conio.h>
#include <stdlib.h>
#include "exemple.h"
#include "exvect.h"
#include "exmat.h"
#include "exmin.h"

// Déclaration des fonctions (prototypes)
void plusDeMemoire();
void menu();

main()
{
   // On indique la fonction à appeler lorsqu'un appel à new échoue.
   set_new_handler(plusDeMemoire);

   menu();

   clrscr();
   cout << "Merci!\nA la prochaine...";

   return 1;
}


void menu()
{
   Booleen terminer = Faux;
   while(!terminer)
   {
      // On affiche le menu à l'écran
      clrscr();
      gotoxy(COINX,COINY);
      cout << "1. Calcul vecteurs" << endl;
      gotoxy(COINX,COINY+1);
      cout << "2. Calcul matrices" << endl;
      gotoxy(COINX,COINY+2);
      cout << "3. Explosion memoire" << endl;
```

```
      gotoxy(COINX,COINY+3);
      cout << "4. Minimums" << endl;
      gotoxy(COINX,COINY+4);
      cout << "Q. Quitter" << endl;

      // On attend une réponse de l'usager
      gotoxy(COINX,COINY+6);
      char rep;
      cin >> rep;
      switch(rep)
      {
      case '1':
         // On choisit l'exemple des vecteurs
         vecteurs();
         break;
      case '2':
         // On choisit l'exemple des matrices
         matrices();
         break;
      case '3':
         // On fait exploser la mémoire pour provoquer un appel à la
         // la fonction plusDeMemoire (par la variable _new_handler)
         EntierP tmp;
         tmp = new Entier[1000000000];
         delete[] tmp;
         break;
      case '4':
         // On choisit l'exemple des minimums
         minimums();
         break;
      case 'Q':
      case 'q':
         // On quitte
         terminer = Vrai;
         break;
      default:
         ;
      }
   }
}


void plusDeMemoire()
{
   cerr << "Désolé, plus de mémoire...\nBye Bye!";
   exit(1);
}
```

exvect.h

```
#ifndef __exvect_h
#define __exvect_h

#include "exemple.h"

// Déclaration des fonctions pour les vecteurs (prototypes)
EntierP mult(EntierP,Entier,Entier);
EntierP mult(EntierP,Entier,EntierP,Entier);
void affiche(EntierP,Entier=TAILLE);
void vecteurs();

#endif
```

exvect.cpp

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include "exvect.h"


void vecteurs()
{
   // Vecteur statique
   Entier vect[TAILLE] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

   // Vecteur dynamique
   EntierP pVect = new Entier[TAILLE];

   // On initialise le vecteur dynamique
   for(Entier i=0; i < TAILLE; i++)
      pVect[i] = i << 1;

   // On multiplie le vecter statique par 3
   EntierP tmp = mult(vect,TAILLE,3);
   affiche(tmp);
   // On efface le résultat après l'avoir affiché
   delete[] tmp;
   getch();

   // On multiplie les deux vecteurs entre eux
   tmp = mult(vect,TAILLE,pVect,TAILLE);
   affiche(tmp);
   delete[] tmp;
   getch();

   affiche(pVect);
   delete[] pVect;
   getch();

}


// Fonction d'affichage
void affiche(EntierP vect,Entier taille)
{
   clrscr();
   for(Entier i=0; i < taille; i++)
      cout << i << " : " << vect[i] << endl;
}


// Fonction de multiplication par un scalaire qui retourne un vecteur
// nouvellement créé.
EntierP mult(EntierP vect, Entier taille, Entier scal)
{
   EntierP retour = new Entier[taille];

   for(Entier i=0; i < taille; i++)
      retour[i] = vect[i] * scal;

   return retour;
}
```

```
// Fonction de multiplication entre deux vecteurs qui retourne un troisième
// vecteur nouvellement créé.
EntierP mult(EntierP vect1, Entier taille1, EntierP vect2, Entier taille2)
{
   Entier taille;
   taille = (taille1 > taille2)? taille2:taille1;

   EntierP retour = new Entier[taille];

   for(Entier i=0; i < taille; i++)
      retour[i] = vect1[i] * vect2[i];

   return retour;
}
```

exmat.h

```
#ifndef __exmat_h
#define __exmat_h

#include "exemple.h"

// Déclaration des fonctions des matrices
void affiche(Reel*[],Entier = TAILLE);
void affiche(Reel[][TAILLE],Entier = TAILLE);
void matrices();

#endif
```

exmat.cpp

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include "exmat.h"

void matrices()
{
   // Matrice statique avec initialisation
   Reel mat[TAILLE][TAILLE] =
  {{0.0F,1.0F,2.0F,3.0F,4.0F,5.0F,6.0F,7.0F,8.0F,9.0F},
   {10.0F,11.0F,12.0F,13.0F,14.0F,15.0F,16.0F,17.0F,18.0F,19.0F},
   {20.0F,21.0F,22.0F,23.0F,24.0F,25.0F,26.0F,27.0F,28.0F,29.0F},
   {30.0F,31.0F,32.0F,33.0F,34.0F,35.0F,36.0F,37.0F,38.0F,39.0F},
   {40.0F,41.0F,42.0F,43.0F,44.0F,45.0F,46.0F,47.0F,48.0F,49.0F},
   {50.0F,51.0F,52.0F,53.0F,54.0F,55.0F,56.0F,57.0F,58.0F,59.0F},
   {60.0F,61.0F,62.0F,63.0F,64.0F,65.0F,66.0F,67.0F,68.0F,69.0F},
   {70.0F,71.0F,72.0F,73.0F,74.0F,75.0F,76.0F,77.0F,78.0F,79.0F},
   {80.0F,81.0F,82.0F,83.0F,84.0F,85.0F,86.0F,87.0F,88.0F,89.0F},
   {90.0F,91.0F,92.0F,93.0F,94.0F,95.0F,96.0F,97.0F,98.0F,99.0F}};

   // Vecteur de pointeurs à des Reel (matrice dynamique)
   Reel *mat2[TAILLE];

   for(Entier i=0; i < TAILLE; i++)
   {
      // On crée chaque rangée de la matrice dynamique
      mat2[i] = new Reel[TAILLE];
```

```
      // On initialise chaque élément
      for(Entier j=0; j < TAILLE; j++)
         mat2[i][j] = i*j;
   }

   affiche(mat);
   getch();
   affiche(mat2);
   getch();

   // On efface chaque rangée de la matrice dynamique
   for(i=0; i < TAILLE; i++)
      delete[] mat2[i];
}


// Fonction d'affichage de la matrice statique
void affiche(Reel mat[][TAILLE],Entier taille)
{
   clrscr();
   for(Entier i=0; i < taille; i++)
   {
      for(Entier j=0; j < TAILLE; j++)
         cout << i << "," << j << ":" << mat[i][j] << " ";
      cout << endl;
   }
}


// Fonction d'affichage de la matrice dynamique
void affiche(Reel **mat,Entier taille)
{
   clrscr();
   for(Entier i=0; i < taille; i++)
   {
      for(Entier j=0; j < TAILLE; j++)
         cout << i << "," << j << ":" << mat[i][j] << " ";
      cout << endl;
   }
}
```

exmin.h

```
#ifndef __exmin_h
#define __exmin_h

#include "exemple.h"

// Déclaration de la fonction générique de min
template <class Type> Type& min(Type&,Type&);

// Définition d'une macro spécialisée de min
inline Entier& min(Entier &x,Entier &y) { return (x < y)? x:y; }

// Déclaration de la macro de max
inline Entier& max(Entier&,Entier&);

void minimums();

#endif
```

exmin.cpp

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include "exmin.h"

void minimums()
{
   clrscr();

   // Pointeur à la fonction max
   Entier& (*pf)(Entier&,Entier&) = max;

   Entier chiffre1 = 98, chiffre2 = 99;

   // On se définit une référence au minimum entre chiffre 1 et chiffre2
   Entier &chiffre3 = min(chiffre1, chiffre2);

   // Le minimum est chiffre 1, chiffre 3 et chiffre 1 sont donc la même
   // varaible (modifier l'une modifie l'autre du même coup).
   cout << "Minimum entre " << chiffre1 << " et " << chiffre2
        << " est " << chiffre3 << endl;

   // chiffre3 et chiffre1 sont affectés à 102
   chiffre3 = 102;

   // Appel de max avec le pointeur à la fonction
   chiffre3 = pf(chiffre1, chiffre2);
   cout << "Maximum entre " << chiffre1 << " et " << chiffre2
        << " est " << chiffre3 << endl;

   cout << endl;
   Reel c1 = 6.5, c2 = 6.6;

   // Appel générique de min.  Une fonction min avec le type Reel est créé.
   Reel &c3 = min(c1, c2);
   cout << "Minimum entre " << c1 << " et " << c2
        << " est " << c3 << endl;

   getch();
}


// Définition de max
inline Entier& max(Entier& x, Entier& y)
{
   return (x > y)? x:y;
}


// Définition de la fonction générique de min
template <class Type> Type& min(Type& x, Type& y)
{
   return (x < y)? x:y;
}
```

construct.h

```
#ifndef __CONSTRUCT_H
#define __CONSTRUCT_H

#include "exemple.h"

class X
{
   Entier n;
public:
   X() { n = 0; }                      // Constructeur par défaut
   X(Entier x) { n = x; }              // Constructeur avec un Entier
   X(const X& x) {n = x.n;}            // Constructeur avec une classe X
   Entier& valeur() { return(n); }     // Méthode qui retourne la valeur de n
};
#endif // #ifndef __CONSTRUCT_H
```

Construct.cpp

```
#include <iostream.h>
#include "construc.h"

// -------------------
// Programme Principal
// -------------------

main()
{
   X un;                // Appel du constructeur par défaut
   X deux(1);           // Appel du constructeur avec un Entier = 1
   X trois = 2;         // Appel du constructeur avec un Entier = 2
   X quatre = un;       // Appel du Constructeur avec une classe X = un
   X cinq(deux);        // Appel du constructeur avec une classe X = deux

   cout << "un = " << un.valeur() << endl;
   cout << "deux = " << deux.valeur() << endl;
   cout << "trois = " << trois.valeur() << endl;
   cout << "quatre = " << quatre.valeur() << endl;
   cout << "cinq = " << cinq.valeur() << endl;
}
```

Destruc.h

```
#ifndef __DESTRUC_H
#define __DESTRUC_H

#include "exemple.h"

class Stack
{
   char    *s;
   Entier  longmax;
public:
   Stack(Entier dim) { s = new char[dim]; longmax = dim;}    // Constructeur
avec
                                                             // avec un Entier
   Stack() { s = new char[100]; longmax = 100; } // Constructeur par défaut
   ~Stack() { delete s;}                         // Destructeur
};
#endif // #ifndef __DESTRUCT_H
```

Destruc.cpp

```
#include <iostream.h>
#include "destruc.h"

main()
{
   Stack stack1();  // Appel du Construteur par défaut
   Stack stack2(10);// Appel du Construteur avec un Entier
}                   // Appel du destructeur
```

Mstatic.h

```
#ifndef __MSTATIC_H
#define __MSTATIC_H

#include "exemple.h"

class ListeElement
{
   static Entier nombre;                                  // Attribut statique
public:
   void initialisation() { nombre = 5; };                 // Initialise de
                                                          // l'attribut
   static Entier nombreElement() { return(nombre);}; // Retourne le nombre
                                                          // éléments
};

#endif // #ifndef __MSTATIC_H
```

Mstatic.cpp

```
#include "MStatic.h"
#include <iostream.h>

Entier ListeElement::nombre = 7;      // Initialisation de l'attribut statique

// --------------------
// Programme Principale
// --------------------

main()
{
   Entier n;

   n = ListeElement::nombreElement();   // Il est possible de savoir la valeur
                                        // de l'attribut statique sans avoir
                                        // d'objet.
   cout << "nombre : " << n << endl;

   ListeElement elem1;                  // Déclaration d'un objet

   elem1.initialisation();              // Initialisation de l'attribut
   n = elem1.nombreElement();
   cout << "nombre : " << n << endl;
}
```

Vecbase.h

```
#ifndef __VECBASE_H
#define __VECBASE_H

#include "exemple.h"

// --------------------------------
// Déclaration de la classe Vecteur
// --------------------------------

class  Vecteur
{
   EntierP   entierP;    // Pointeur au vecteur
   Entier    dim;        // Longueur du vecteur
   Entier    borneSup;   // Borne supérieur du vecteur
public:
   void      initialisation(Entier); // Méthode d'initialisation du vecteur
   Entier&   element(Entier);        // Cette méthode permet d'ajouter ou
                                     // d'obtenir un élément du vecteur
};

#endif // ifndef __VECBASE_H
```

Vecbase.cpp

```
#include <iostream.h>
#include <stdlib.h>
#include "vecbase.h"

// ------------------------------------
// Méthode d'initialisation du vecteur
// ------------------------------------

void Vecteur::initialisation(Entier n)
{
   if(n <= 0) exit(1);          // Vérifie la dimension du vecteur

   dim = n;                     // Initialisation de la dimension du vecteur
   entierP = new Entier[dim]; // Allocation de l'espace mémoire du vecteur
   borneSup = dim - 1;          // Initialisation de la borne supérieur du
                                // vecteur
}

// ---------------------------------------------------
// Permet d'ajouter ou d'obtenir un élément du vecteur
// ---------------------------------------------------

Entier& Vecteur::element(Entier position)
{
   if( position < 0 || position > borneSup)  exit(1);  // Vérificatio de la
                                                       // position
   return entierP[position];
}



// -----------------------------------------------
// Le programme ajoute et affiche les valeurs
// d'un vecteur statique et d'un vecteur dynamique
// -----------------------------------------------
```

```
main()
{
   Vecteur vecA;                  // Définition du vecteur statique
   Vecteur *vecBP = new Vecteur; // Définition du vecteur dynamique

   vecA.initialisation(5);        // Initialisation du vecteur statique
   vecBP->initialisation(5);      // Initialisation du vecteur dynamique

   for(Entier i = 0; i < 5; i++)
   {
      // Ajout d'un Entier dans les deux vecteurs

      vecA.element(i) = i;
      vecBP->element(i) = vecA.element(i) + i;
   }

   for(i = 0; i < 5; i++)
   {
      // Affiche les vecteurs

      cout << "vecA[" << i << "]  = " << vecA.element(i) << endl;
      cout << "vecBP[" << i << "] = " << vecBP->element(i) << endl;
   }
}
```

Vecconst.h

```
#ifndef __VECCONST_H
#define __VECCONST_H

#include "exemple.h"

// --------------------------------
// Déclaration de la classe Vecteur
// --------------------------------

class  Vecteur
{
   EntierP   entierP;   // Pointeur au vecteur
   Entier    dim;       // Longueur du vecteur
   Entier    borneSup;  // Borne supérieur du vecteur
public:
             Vecteur(Entier n = 5);  // Constructeur par défaut du vecteur
             ~Vecteur();             // Destructeur du vecteur
   Entier&   element(Entier);        // Cette méthode permet d'ajouter ou
                                     // d'obtenir un élément du vecteur.
};

#endif // ifndef __VECCONST_H
```

Vecconst.cpp

```
#include <iostream.h>
#include <stdlib.h>
#include "vecconst.h"

// ----------------------
// Constructeur du vecteur
// ----------------------

Vecteur::Vecteur(Entier n)
{
   if(n <= 0) exit(1);          // Vérification de la dimension du vecteur

   dim = n;                     // Initialisation de la dimension du vecteur
   entierP = new Entier[dim]; // Allocation de l'espace mémoire du vecteur
   borneSup = dim - 1;          // Initialisation de la borne supérieur du
                                // vecteur
}

// ----------------------
// Destructeur du vecteur
// ----------------------

Vecteur::~Vecteur()
{
   delete[] entierP;            // Libération de la mémoire du vecteur
}

// ---------------------------------------------------------------------
// Cette méthode permet d'ajouter ou d'obtenir un élément du vecteur
// ---------------------------------------------------------------------

Entier& Vecteur::element(Entier position)
{
   if( position < 0 || position > borneSup)  exit(1);// Véfication de la
                                                     // position
   return entierP[position];
}

// ------------------------------------------------
// Le programme ajoute et affiche les valeurs
// d'un vecteur statique et d'un vecteur dynamique
// ------------------------------------------------

main()
{
   Vecteur vecA;                    // Définition du vecteur statique
   Vecteur *vecBP = new Vecteur;  // Définition du vecteur dynamique

   for(Entier i = 0; i < 5; i++)
   {
      // Ajout d'un Entier dans les deux vecteurs

      vecA.element(i) = i;
      vecBP->element(i) = vecA.element(i) + i;
   }
   for(i = 0; i < 5; i++)
   {
      // Affiche les vecteurs
      cout << "vecA[" << i << "]  = " << vecA.element(i) << endl;
      cout << "vecBP[" << i << "] = " << vecBP->element(i) << endl;
   }
}
```

Vectopr.h

```
#ifndef __VECTOPR_H
#define __VECTOPR_H

#include "exemple.h"

class Vecteur
{
   EntierP entierP;      // Pointeur au vecteur
   Entier  dim;          // Grandeur du vecteur
public:
   Entier bs() { return (dim -1);} // Borne Supérieur du vecteur

   Vecteur() { dim = 10; entierP = new Entier[dim];}
   Vecteur(Entier n);
   ~Vecteur() { delete entierP; };
   Entier &operator[] (Entier);     // Surcharge de l'opérateur []
   Entier &element(Entier);         // Cette méthode permet d'obtenir ou
                                    // d'ajouter un Entier.
};

#endif // #ifndef __VECTOPR_H
```

Vectopr.cpp

```
#include <iostream.h>
#include <stdlib.h>
#include "vectopr.h"

// ---------------------------------
// Constructeur de la classe vecteur
// ---------------------------------

Vecteur::Vecteur(Entier n)
{
   if (n <= 0)  exit(1);        // Vérification de la dimension du vecteur

   dim = n;
   entierP = new Entier[dim];  // Allocation de la mémoire pour le vecteur
}

// -------------------------------------
// Cette méthode qui permet d'obtenir ou
// d'ajouter un entier dans le vecteur
// -------------------------------------

Entier &Vecteur::element(Entier n)
{
   if ( n < 0 || n > bs() )  exit(1); // Vérification de la position

   return( entierP[n]);               // Retourne l'entier
}

// ----------------------------------------
// Surcharge de l'opérateur []
// ----------------------------------------
Entier& Vecteur::operator[] (Entier n)
{
   if ( n < 0 || n > bs() )  exit(1);  // Vérification de la position

   return(entierP[n]);                 // Retour de l'entier
}
```

```
// -------------------
// Programme principal
// -------------------

main()
{
   Vecteur vec(5);

   for (Entier i = 0; i <= vec.bs(); ++i)
   {
      vec.element(i) = i;       // La méthode element et la surcharge de
      vec[i] = i + 1;           // l'opérateur [] donne le même résultat.
   }

   for ( i = 0; i <= vec.bs(); i++)
   {
      cout << "vecteur[" << i << "] =" << vec.element(i) << endl;
      cout << "vec[" << i << "] =" << vec[i] << endl;
   }
}
```

Opera.h

```
#ifndef __OPERA_H
#define __OPERA_H

#include "exemple.h"

class TroisD
{
   Entier x, y, z;
public:
   TroisD  operator+(TroisD t);     // Surcharge de l'opérateur binaire +
   TroisD& operator=(TroisD t);     // Surcharge de l'opérateur unitaire =
   TroisD& operator++(void);        // Surcharge de l'opétateur unitaire ++
   void voir(void);                 // Affiche la valeur de x, y et z
   void assigner( Entier mx, Entier my, Entier mz); // Assigne une valeur à x,
                                                    // y et z
};

#endif // #ifndef __OPERA_H
```

Opera.cpp

```
#include <iostream.h>
#include "opera.h"

// ---------------------------------
// Surcharge de l'operateur binaire +
// ---------------------------------

TroisD TroisD::operator+(TroisD t)
{
   TroisD temp;

   temp.x = x + t.x;
   temp.y = y + t.y;
   temp.z = z + t.z;
   return(temp);
}

// ---------------------------------
```

```
// Surcharge de l'operateur unitaire =
// ----------------------------------

TroisD& TroisD::operator=(TroisD t)
{
   x = t.x;
   y = t.y;
   z = t.z;
   return(*this);
}

// ------------------------------------
// Surcharge de l'operateur unitaire ++
// ------------------------------------

TroisD& TroisD::operator++()
{
   x++;
   y++;
   z++;
   return (*this);
}

// ------------------------------
// Affiche la valeur de x, y et z
// ------------------------------

void TroisD::voir(void)
{
   cout << x << "," << y << "," << z << endl;
}

// ------------------------------
// Assigne une valeur à x, y et z
// ------------------------------

void TroisD::assigner(Entier mx, Entier my, Entier mz)
{
   x = mx;
   y = my;
   z = mz;
}

// --------------------
// Programme principal
// --------------------

main()
{
   TroisD a, b, c;

   a.assigner(1,2,3);
   b.assigner(10, 10, 10);

   a.voir();
   b.voir();

   c = a + b;
   c.voir();

   c = a = b;
   c.voir();
   b.voir();
}
```

Friend.h

```
#ifndef __FRIEND_H
#define __FRIEND_H

#include "exemple.h"

class Vecteur;
class Matrice;

// --------------------------------
// Déclaration de la classe Vecteur
// --------------------------------

class  Vecteur
{
   EntierP   entierP;    // Pointeur au vecteur
   Entier    dim;        // Longueur du vecteur

   friend    Vecteur  *mult(Vecteur&, Matrice&); // Méthode accécible par
                                                 // d'autre classe.

public:
   Entier     borneSup;                          // borne du vecteur
              Vecteur() {dim = 5;entierP = new Entier[dim];borneSup = dim - 1;}
              Vecteur(Entier n) { dim = n; entierP = new Entier[dim];
                                  borneSup = dim - 1;}
              ~Vecteur() { delete entierP;}    // Constructeur du vecteur
   Entier&    element(Entier);                 // Cette méthode permet d'ajouter
                                               // ou d'obtenir un élément du
                                               // vecteur
};

// --------------------------------
// Déclaration de la classe Matrice
// --------------------------------

class Matrice
{
   EntierPP   entierPP;               // Pointeur à la matrice
   Entier     dimR, dimC;             // Dimensions de la matrice

   friend     Vecteur  *mult(Vecteur&, Matrice&); // Cette méthode est défini
public:                                           // dans la classe vecteur.
   Entier     borneSupR, borneSupC;               // Bornes de la matrice
              Matrice(Entier, Entier);
              ~Matrice();
   Entier&    element(Entier, Entier);
};

#endif // ifndef __FRIEND_H
```

Friend.cpp

```
#include <iostream.h>
#include <stdlib.h>
#include "friend.h"



// ------------------------------------------------
// Permet d'ajouter ou d'obtenir un élément du vecteur
// ------------------------------------------------
```

```
Entier& Vecteur::element(Entier position)
{
   if( position < 0 || position > borneSup)  exit(1);  // Véfication de la
                                                       // position
   return entierP[position];
}

// ----------------------------------
// Multiplie une matrice par un vecteur
// ----------------------------------

Vecteur *mult(Vecteur &vec, Matrice &mat)
{
   if (vec.dim != mat.dimR) exit(1);    // Vérifie la longueur du vecteur
                                        // est égal au nombre de rangés de la
                                        // matrice
   Vecteur *repP = new Vecteur(mat.dimC);

   for(Entier i = 0; i <= mat.borneSupC; ++i)
   {
      repP->entierP[i] = 0;
      for(Entier j =0; j <= mat.borneSupR; j++)
         repP->entierP[i] += vec.entierP[j] * mat.entierPP[i][j];
   }
   return repP;
}

// ---------------------------------
// Constructeur de la classe Matrice
// ---------------------------------

Matrice::Matrice( Entier d1, Entier d2 )
{
   if (d1 < 0 || d2 < 0)   exit(1);   // Vérification des dimensions de la
                                      // matrice

   dimR = d1;                         // Initialisation du nombre de rangés
   dimC = d2;                         // Initialisation du nombre de colonnes
   entierPP = new EntierP[dimR];      // Initialisation du pointeur de tête de
la matrice

   for (Entier i = 0; i < dimR; i++)
      entierPP[i] = new Entier[dimC]; // Allouer l'espace mémoire pour la
                                      // matrice

   borneSupR = dimR - 1;              // Initialisation de la borne supérieur
                                      // du nombre de rangés
   borneSupC = dimC - 1;              // Initialisation de la borne supérieur
                                      // du nombre de colonnes
}

// ---------------------------------
// Destructeur de la classe Matrice
// ---------------------------------

Matrice::~Matrice()
{
   for (Entier i = 0; i <= borneSupR; i++)
      delete entierPP[i];            // Libération de l'espace mémoire de la
                                     // matrice
   delete entierPP;                  // Libération du pointeur de tête
}
// ------------------------------------------------------------------------
```

```
// Méthode éléments qui permet ajouter ou obtenir un élément de la matrice
// ------------------------------------------------------------------------

Entier& Matrice::element(Entier i, Entier j)
{
   if(i < 0 || i > borneSupR || j > borneSupC)
      exit(1);
   return (entierPP[i][j]);
}

// ---------------------------------------
// Le programme utilise la méthode mult
// commune à la classe Vecteur et la
// classe Matrice.
// ---------------------------------------

main()
{
   Vecteur vec(5);
   Vecteur *vecRepP = new Vecteur(5);
   Matrice mat(5,5);

   for(Entier i = 0; i <= mat.borneSupC; i++)
   {
      vec.element(i) = i;
      for(Entier j = 0; j <= mat.borneSupR; j++)
         mat.element(j,i) = j + i;
   }
   vecRepP = mult(vec, mat);

   for( i = 0; i <= vec.borneSup; i++)
      cout << "vecteurRep[" << i << "] = " << vecRepP->element(i) << endl;
   delete vecRepP;
}
```

Heritage.h

```
#ifndef __HERIATGE_H
#define __HERIATGE_H

#include "exemple.h"


// ---------------
// Classe Personne
// ---------------

class Personne
{
   char *nom;
public:
   Personne(char *n) { nom = n;} // Constructeur par défaut
   void virtual afficheNom(){cout << "nom de la personne : " << nom << endl; }
};



// -----------------
// classe Enseignant
// -----------------

class Enseignant : public Personne
{
```

```
   Entier nbrCours;           // Nombre de cours
   Entier MaxCours;           // Nombre maximum de cours
public:
                                          // Constructeur
   Enseignant(char *nom) : Personne(nom) { nbrCours = 0; MaxCours = 2;}
   void ajouteCours();        // Méthode qui ajoute un cours
                              // Méthode qui affiche le nombre de cours
   void afficheNbrCours() { cout << "nbrCours = " << nbrCours << endl;}
};

// ---------------
// Classe Etudiant
// ---------------

class Etudiant : public Personne
{
public:
   Etudiant(char *nom):Personne(nom){};  // Constructeur
   void afficheNom();                    // Méthode qui affiche le nom.
};

#endif // #ifndef __HERITAGE_H
```

Heritage.cpp

```
#include <iostream.h>
#include "heritage.h"

// ---------------------------
// Méthode qui ajoute un cours
// à la classe Enseignant
// ---------------------------

void Enseignant::ajouteCours()
{
   if(nbrCours < MaxCours) nbrCours++;
   else  cout << "Le cours ne peut être ajouté" << endl;
}


// ---------------------------
// Méthode qui affiche le nom
// de l'étudiant
// ---------------------------

void Etudiant::afficheNom()
{
   cout << "Le nom de l'étudiant est : ";
   Personne::afficheNom();
}


// -------------------
// Programme principal
// -------------------

main()
{
   Personne   pers("Marc");   // Constructeur de la classe Personne
   Enseignant prof("Michel"); // Constructeur de la classe Enseignant
   Etudiant   etud("André");  // Constructeur de la classe Etudiant

   pers.afficheNom();         // Affiche le nom de la personne
```

```
    prof.afficheNom();          // Affiche le nom de l'enseignant
    etud.afficheNom();          // Affiche le nom de l'étudiant

    prof.afficheNbrCours();     // Affiche le nombre de cours de l'enseignant
    prof.ajouteCours();         // Ajoute un cours à l'enseignant
    prof.afficheNbrCours();
    prof.ajouteCours();
    prof.ajouteCours();
    prof.afficheNbrCours();
}
```

11 Annexe B

Cette annexe contient les fonctions disponibles avec les librairies standards du C++ (et du C). Seules les fonctions répondant à la norme ANSI C sont inscrites. Le nom de la fonction est suivi du fichier d'entête dans lequel se trouve sa déclaration.

Les fonctions suivies d'une astérisque ('*') ne sont pas disponibles sous des environnements graphiques tels Windows, Presentation Manager, etc...

Fonctions de classifications des caractères

isalnum	ctype.h	*isgraph*	ctype.h	*isspace*	ctype.h
isalpha	ctype.h	*islower*	ctype.h	*isupper*	ctype.h
iscntrl	ctype.h	*isprint*	ctype.h	*isxdigit*	ctype.h
isdigit	ctype.h	*ispunct*	ctype.h		

Fonctions de conversions

atof	stdlib.h	*atol*	stdlib.h	*tolower*	ctype.h
atoi	stdlib.h	*strtod*	stdlib.h	*toupper*	ctype.h

Fonctions mathématiques

abs	stdlib.h	*exp*	math.h	*rand*	stdlib.h
acos	math.h	*fabs*	math.h	*sin*	math.h
asin	math.h	*floor*	math.h	*sinh*	math.h
atan	math.h	*frexp*	math.h	*sqrt*	math.h
atan2	math.h	*labs*	stdlib.h	*srand*	stdlib.h
atof	math.h	*ldexp*	math.h	*strtod*	stdlib.h
atoi	stdlib.h	*log*	math.h	*tan*	math.h
atol	stdlib.h	*log10*	math.h	*tanh*	math.h
ceil	math.h	*modf*	math.h		
cos	math.h	*pow*	math.h		

Fonctions d'entrées/sorties

clearerr	stdio.h	*freopen*	stdio.h	*rewind*	stdio.h
fclose	stdio.h	*fscanf*	stdio.h	*scanf**	stdio.h
feof	stdio.h	*fseek*	stdio.h	*setbuf*	stdio.h
ferror	stdio.h	*ftell*	stdio.h	*setvbuf*	stdio.h
fflush	stdio.h	*fwrite*	stdio.h	*sprintf*	stdio.h
fgetc	stdio.h	*getc*	stdio.h	*sscanf*	stdio.h
fgets	stdio.h	*getchar**	stdio.h	*tmpfile*	stdio.h
fopen	stdio.h	*gets**	stdio.h	*tmpnam*	stdio.h
fprintf	stdio.h	*perror**	stdio.h	*ungetc*	stdio.h
fputc	stdio.h	*printf**	stdio.h	*vfprintf*	stdio.h
fputs	stdio.h	*putchar*	stdio.h	*vprintf**	stdio.h
fread	stdio.h	*puts**	stdio.h	*vsprintf**	stdio.h

Fonctions de manipulation

memchr	string.h	*strcpy*	string.h	*strrchr*	string.h
memcmp	string.h	*strcspn*	string.h	*strspn*	string.h
memcpy	string.h	*strcmpi**	string.h	*strstr*	string.h
memmove	string.h	*strlen*	string.h	*strtok*	string.h
memset	string.h	*strncat*	string.h	*wcstombs*	stdlib.h
strcat	string.h	*strncmp*	string.h	*wctomb*	stdlib.h
strchr	string.h	*strncpy*	string.h		
strcmp	string.h	*strpbrk*	string.h		

Fonctions de mémoire

calloc	stdlib.h	*malloc*	stdlib.h
free	stdlib.h	*realloc*	stdlib.h

Fonctions d'heure et de date

asctime	time.h	*difftime*	time.h	*localtime*	time.h
ctime	time.h	*gmtime*	time.h	*time*	time.h

12 Bibliographie

- The C++ Programming Language, second edition, Stroustrup, Addison/Wesley
- A C++ Primer, second edition, Lippman, Addison/Wesley
- Annotated C++ Reference Manual, Ellis and Stroustrup, Addison/Wesley
- Advanced C++ Programming Styles and Idioms, Coplien, Addison/Wesley
- A C++ Toolkit, Shapiro, Prentice Hall
- Mastering C++, Horstmann, Wiley
- Programming in C++, Dewhurst and Stark, Prentice Hall
- Using C++, Eckel, McGraw/Hill
- The Waite Group`s C++ Programming, Berry
- C++ for C Programmers, Pohl, Benjamin/Cummings
- Programming in a Object-Oriented Environment, Ege, Academic Press