Mémento de C et C++

C. Jacquemard et G. Laurent

2011

Table des matières

Assertissement

Ce document propose un condensé de la syntaxe du langage illustré de nombreux exemples. Ce document ne prétend pas à l'exhaustivité. Ce n'est ni une documentation technique, ni un cours complet sur le langage C et C++, encore moins un cours de programmation.

Chapitre 1

Langage C

1.1 Symboles, identificateurs et commentaires

1.1.1 Symboles de base du langage

Lettres : `A, ... , Z, a, ... , z`
Chiffres : `0, ... , 9`
Caractères spéciaux : `+ - * / _ = < > ^ ~ ( )[ ] { } . , ; : '" \ | & % ! ?` et l'espace.

1.1.2 Identificateurs

Un identificateur commence par une lettre ou _, suivi d'une combinaison de lettres, de chiffres et _ (l'espace et autres symboles spéciaux sont proscrits).

Le langage C distingue les majuscules des minuscules dans les identificateurs.

1.1.3 Mots réservés

Un certain nombre d'identificateurs sont réservés et ne peuvent être redéfinis par l'utilisateur ou utilisés autrement que suivant la syntaxe prévue. Les mots réservés du C et du C++ sont :

```
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
```

1.1.4 Commentaires

Un commentaire dans un programme C se place entre les balises `/*` et `*/`.

1.2 Types simples

1.2.1 Type énuméré

Un type énuméré est défini par l'énumération ordonnée des identificateurs représentant les constantes du type, entre accolades.

```
enum identificateur_de_type { liste_des_constantes_du_type };
```

Les éléments de la liste sont séparés par des virgules.

```
enum couleur { pique, coeur, carreau, trefle };
  /* pique est codé par la valeur 0, coeur par la valeur 1, etc. */

enum jour    { Lundi=1, Mardi, Mercredi, Jeudi, Vendredi, Samedi };
  /* Lundi est codé par la valeur 1, Mardi par la valeur 2, etc. */

enum booleen { FAUX=0, VRAI=1 };
```

1.2.2 Type entier : char, short, int, long, unsigned

La différence entre ces différents types est uniquement sur le nombre d'octets de codage en mémoire :

char	entier sur 1 octet (-128 à 127)
short	entier sur 2 octets (-32 768 à 32 767)
int	entier sur 2 ou 4 octets suivant les compilateurs et les processeurs
long	entier sur 4 octets (-2 147 483 648 à 2 147 483 647)
long int	équivalent à **long**

Tous ces types entiers peuvent être précédés du mot-clé **unsigned**.

unsigned char	entier sur 1 octet (0 à 255)
unsigned short	entier sur 2 octets (0 à 32 767)
unsigned int	entier sur 2 ou 4 octets suivant les compilateurs et les processeurs
unsigned long	entier sur 4 octets (0 à 4 294 967 295)

1.2.3 Type réel : float, double

La représentation flottante utilise une mantisse et un exposant codés sur un certain nombre d'octets.

float	réel sur 4 octets (3,4e-38 à 1,7e38, 7 chiffres significatifs)
double	réel sur 8 octets (1,7e-308 à 3,4e308, 15 chiffres significatifs)
long double	réel sur 10 octets (3,4e-4932 à 1,7e4932, 18 chiffres significatifs)

1.2.4 Type vide : void

Le type **void** est un type de taille mémoire nulle.

1.2.5 Type pointeur

Un pointeur (adresse) est typé par le type de l'objet pointé, précédé de *.

```
int *pe;   /* pe est un pointeur sur un entier */
char *pc;  /* pc est un pointeur sur un caractère */
```

Il existe une constante de type pointeur pré-définie dans les bibliothèques standards, la constante `NULL`.

Un pointeur du type **void** * n'est pas typé et est générique.

1.2.6 Autres types

Le langage C ne fournit pas d'autres types, notamment pas de type booléen ni de type chaîne. Néanmoins, une chaîne de caractère est habituellement stockée dans un tableau de caractères (*cf.* §1.4.2).

1.2.7 Définition de type : typedef

L'instruction **typedef** permet de nommer un nouveau type ou renommer un type existant.

```
typedef description_du_type identificateur_de_type;
```

Exemple :

```
typedef short petit_entier; /* petit_entier désignera un short */
typedef short booleen; /* booleen désignera aussi un short */
typedef char chaine[20]; /* chaine est le nom du type tableau de 20 caractères */
```

1.2.8 Conversion de type (cast)

En plus des règles standards de conversion de type dans les expressions, les identificateurs de type peuvent servir d'opérateurs de conversion.

```
float x;
int  i;
i=(int) x;
```

1.3 Constantes, variables et expressions

1.3.1 Constantes

Le langage C utilise les conventions ci-dessous pour coder des valeurs constantes :

décimal :	2	-12	+123	
octal :	012	03775		(précédé de 0)
hexadécimal :	0xa	0XA1F		(précédé de 0x ou 0X)
non signé :	12u	012u	0xa1fu	(entier suivi de u)
entier long :	12l	012L	0xa1fuL	(entier suivi de l ou L)
réel :	2.	3.14	-45.2	.15
	2.25f	3.01F		(virgule fixe : f ou F)
	10e-1	-45.2E+12		(virgule flottante : e ou E)
caractère :	'a'	'C'	'+'	(entre simples quotes)
chaîne :	"commentaire"		"Claude"	(entre doubles quotes)

Les caractères spéciaux sont précédés d'un \ (anti-slash ou contre-barre).

'\n'	nouvelle ligne
'\r'	début de ligne
'\b'	retour arrière
'\t'	tabulation
'\\'	contre-barre
'\f'	nouvelle page
'\0'	caractère NULL
'\''	simple quote
'\"'	double quote
'\ooo'	caractère de code octal ooo
'\xhh'	caractère de code hexadécimal hh

Le mot réservé **const** permet de déclarer des constantes.

```
const type nom = valeur;
```

Exemple :

```
const float pi = 3.14159;
const char CR = '\n';
```

1.3.2 Variables

Toute variable doit être déclarée avant sa première utilisation. Une initialisation peut être faite au moment des déclarations.

```
type liste_de_noms_de_variables;

type noms = valeurs;
```

Exemple :

```
char  a , b;
int  j , i = 2;
float  resultat = 0.0;
char  nombre = '1' ,  lettre = 'a';
```

1.3.3 Opérateurs arithmétiques

Opérateurs binaires :

- `+` addition
- `-` soustraction
- `*` multiplication
- `/` division (entière si les deux opérandes sont entiers, réelle sinon)
- `%` modulo (reste de la division entière)

Opérateurs unaires :

- `-` opposé
- `++` auto-incrémentation (pré ou post-fixe)
- `--` auto-décrémentation (pré ou post-fixe)

Exemples d'utilisation :

```
int  i = 1 , j , k ;
j = i++ ; /* j reçoit la valeur 1, puis i est modifié en 2 */
k = ++j ; /* j est d'abord modifiée en 2, puis k reçoit 2 */
i++ ;     /* i est modifié en 3 */
```

1.3.4 Opérateurs relationnels

Ils permettent de comparer deux opérandes. Le résultat de la comparaison est un nombre entier : 0 si le résultat est faux, 1 si le résultat est vrai.

- `<` infériorité stricte
- `>` supériorité stricte
- `<=` infériorité large
- `>=` supériorité large
- `==` égalité
- `!=` inégalité

Exemple d'utilisation :

```
int b ;
b = ( x != y );  /* b reçoit la valeur 0 si x est égal à y , 1 sinon */
```

1.3.5 Opérateurs logiques

Comme pour les opérateurs relationnels, le résultat d'une expression logique est une valeur entière 0 ou 1.

- `!` négation logique
- `&&` et logique
- `||` ou logique

Exemple :

```
c = ! ( a | | b ) && ( i<j);
```

Remarques :

– dans l'expression `(A&&B)`, si `A` est faux, `B` n'est pas évalué
– dans l'expression `(A||B)`, si `A` est vrai , `B` n'est pas évalué

1.3.6 Opérateurs sur les bits

Ils ne peuvent être employés que sur des opérandes entiers et opèrent bit à bit.

`&`	et
`|`	ou
`^`	ou exclusif
`~`	négation (complément)
`<<`	décalage à gauche
`>>`	décalage à droite

Exemple : `x<<y` retourne `x` décalé de `y` bits vers la gauche correspond à `x` multiplié par 2 puissance `y`, codé sur le même nombre de bits que `x`.

1.3.7 Opérateurs d'affectation

L'affectation est considérée comme un opérateur.

```
var = expr /* où var est une variable et expr une expression. */
```

Lors d'une affectation comme `x = expression`, l'expression est évaluée, cette valeur est affectée à la variable `x`, de plus, cette valeur est le résultat de l'opération d'affectation.

Ceci permet des affectations multiples en une seule instruction comme `x = y = z = 1` (`z` reçoit 1, le résultat de l'affectation qui est 1 est affecté à `y`, *etc.*).

Ceci permet également les affectations composées. Ces affectations combinent les opérateurs arithmétiques, logiques et de décalage avec l'affectation.

```
var op= expr /* où var est une variable, expr une expression
                et op= un opérateur. */
```

Ceci est équivalent à `v = (v)op (e)`

Arithmétiques	`+=`	`-=`	`*=`	`/=` `%=`
Logiques	`&=`	`|=`	`^=`	
Décalage	`<<=`	`>>=`		

Exemples :

```
x += 1   /* correspond à x = x+1  ou à  x++  */
a *= b   /* correspond à  a = a*b  */
h &= xFF /* correspond à  h = h & xFF  */
r <<= 4  /* correspond à  r = r << 4  */
```

Enfin, l'affectation peut être conditionnelle, c'est alors un opérateur ternaire.

```
e1 ? e2 : e3
```

Si `e1` est vrai, l'expression est égale à la valeur `e2`, sinon l'expression est égale à `e3`. Par exemple :

```
m = ( a > b ) ? a : b ;  /* m reçoit le max de a et b */
( a == b ) ? c++ : c-- ; /* on ajoute 1 à c si a est égale à b, sinon on enlève 1 à c */
```

1.3.8 Opérateur de taille (sizeof)

Cet opérateur délivre le nombre d'octets occupés en mémoire par une expression ou un type. La valeur retournée est du type `size_t` (entier non signé). Pour un type, des parenthèses sont requises, car il s'agit alors d'une fonction.

```
sizeof i
sizeof a[1]
sizeof(int)
```

```
n = sizeof a / sizeof a[0];
      /* permet de connaître la nombre d'éléments du tableau a */
```

1.3.9 Opérateurs sur les pointeurs

Il existe quelques opérateurs unaires sur les pointeurs :

*	donne accès à l'élément pointé
&	fournit l'adresse de la variable
++	incrémente le pointeur du nombre d'octets de l'objet pointé
--	décrémente le pointeur du nombre d'octets de l'objet pointé

Exemples d'utilisation :

```
int  e = 0 ;
int *pe ; /* pe est un pointeur sur un entier */
pe = &e ; /* pe reçoit l'adresse de la variable entière e */
*pe = 3 ; /* l'emplacement pointé par pe reçoit 3 */
pe++;     /* pe pointe 2 octets plus loin */
*pe = 4;
```

Les fonctions `malloc()` et `free()` de la bibliothèque standard permettent d'allouer et de libérer dynamiquement de la mémoire. Elles retournent des pointeurs du type **`void`** `*`, c'est-à-dire des pointeurs non typés. Le pointeur retourné par `malloc()` peut être converti en pointeur typé.

S'il n'y a plus de place mémoire disponible, la fonction `malloc` retourne la valeur `NULL`.

```
char  *p;   /* p est un pointeur sur un caractère */
p = malloc(100); /* alloue de la place pour 100 caractères */
...
free(p);    /* libère la place des 100 caractères */

truc *q;    /* q est un pointeur sur un type truc   défini ailleurs*/
q = (truc *)malloc(10*sizeof(truc));
            /* alloue de la place pour 10 éléments de type truc  */
```

1.3.10 Opérateur de séquence

L'opérateur de séquence (`,`) autorise une liste d'expressions là où une seule expression est permise. Dans ce cas, les expressions sont évaluées de gauche vers droite. Le type et la valeur de la liste d'expressions est donc le type et la valeur de l'expression la plus à droite.

```
for ( i=0,j=0 ; i<10 ; i++,j++ )
  ...
```

1.3.11 Priorité des opérateurs

En l'absence de parenthèses, l'ordre de priorité des opérateurs de C et C++ est consigné dans le tableau en annexe A.

1.4 Types composés

Le langage C fournit une seule structure de données, le tableau et permet la construction de nouveaux types structurés (**`struct`** et **`union`**).

1.4.1 Tableaux

Tableaux mono-dimensionnels

Un tableau est une collection d'éléments du même type accessibles par un indice.

```
type nom[taille];
```

où :
– « taille » est une expression constante entière,

– les éléments du tableau sont accessibles par un indice compris entre 0 et taille-1 ;

```
int a[10]; /* déclare une variable tableau a de 10 éléments a[0], ... , a[9] */
typedef char message[20]; /* message est le nom du type tableau de 20 caractères */
typedef float tab[10]; /* tab est le nom du type tableau de 10 réels */
```

En général, le compilateur ne vérifie pas les débordements d'indice !

Un tableau peut être initialisé lors de sa déclaration par une liste de valeurs entre accolades (si la liste est insuffisante, la fin du tableau est initialisée par des 0).

```
char alerte[10] = { 'A','t','t','e','n','t','i','o','n' '!' };
float a[] = { 1.0 , 2.0 , -3.14 }; /* réserve la place pour un tableau de 3 réels */
int b[5] = { 1,2,3 }; /* b[3] et b[4] sont mis à 0 */
```

Relations entre tableaux et pointeurs

En C, l'identificateur d'une variable tableau est équivalent à un pointeur constant pointant sur le premier élément du tableau.

```
int  a[10];
int *pa;
pa = a; /*  Le pointeur sur entier pa contient l'adresse de l'entier a[0],
            i.e l'adresse du début du tableau a.  */
pa = &a[0]; /* c'est équivalent */
```

Ainsi, `a[0]` est accessible par `*pa`, c'est-à-dire par `*(&a[0])`.

D'autre part, un pointeur peut s'auto-incrémenter. L'adresse contenue dans le pointeur est automatiquement augmentée du nombre d'octets du type de l'élément pointé. De façon générale, `a[i]` est accessible par `*(pa+i)`. Les instructions suivantes sont donc équivalentes : `a[1]=x;`, `*(pa+1)=x;` et `*(++pa)=x;`, par exemple :

```
/* trois façons équivalentes de traiter le tableau a.*/
for (i=0; i<10; i++) a[i]    = i;
for (i=0; i<10; i++) *(pa+i) = i;
for (i=0; i<10; i++) *(pa++) = i;
```

« équivalent » signifie ici que le résultat est identique pour l'utilisateur externe ; ce n'est évidemment pas identique pour la machine et ses performances.

Ne pas oublier que le nom du tableau est une adresse constante. Par conséquent, des instructions comme `a++` ou `a = pa` sont illégales.

Tableaux multi-dimensionnels

```
type nom[taille1][taille2]...[taillen];
```

Exemple :

```
int a[5][3];
/* a est un tableau de 5 lignes et  3 colonnes , a(i,j)  0<=i<=4 , 0<=j<=2 */
/* a est un tableau de 5 éléments, un élément étant un tableau de 3 entiers */
```

Les éléments sont stockés en mémoire de façon contiguë.
Un tableau peut être initialisé lors de sa déclaration par une liste de valeurs entre accolades.

```
int  a[3][2] = { 1,2,3,4,5,6 }; /* vision mono-dimensionnelle */
int  a[3][2] = { {1,2},{3,4},{5,6}}; /* vision multi-dimensionnelle */
```

De par la relation entre tableau et pointeur, `a[i]` est aussi un pointeur et `a` un pointeur sur pointeur. Les notations suivantes sont donc équivalentes :

1. notation indicée habituelle : `a[i][j]`,
2. vision du tableau bidimensionnel comme tableau de pointeurs : `*(a[i]+j)`,
3. vision du tableau bidimensionnel comme pointeur de pointeur : `*(*(a+i)+j)`,
4. vision du tableau bidimensionnel comme tableau monodimensionnel : `*(*a+ i*m+j)` soit `*(&a[0][0]+ i*m+j)`.

Allocation dynamique de tableaux

```
int  *pa;
pa = ( int *)malloc(100); /* est une façon d'allouer dynamiquement un tableau de 100 entiers
    */

char  *chaine;
chaine = malloc(30); /* est une façon d'allouer dynamiquement un tableau de 30 caractères,
    soit une chaîne de 30 caractères. */

truc  *ptruc; /* où le type truc  est défini par ailleurs */
ptruc = (truc *)malloc(10*sizeof(truc));
ptruc = malloc(10*sizeof(truc));
/* sont des façons d'allouer dynamiquement un tableau de 10 trucs. */
```

1.4.2 Chaînes de caractères

En langage C, une chaîne de caractères est habituellement défini par un tableau de caractères. Le dernier caractère d'une chaîne est le caractère spécial `'\0'`.

```
char  a[10] = "Claude";
char  b[10] = { 'C','l','a','u','d','e','\0' };
a[2]='o';
a[3]='\0'; /* a contient alors la chaîne "Clo" */

char a[] = "Claude"; /* réserve la place suffisante pour la chaîne, soit ici un tableau de 7
    caractères */
```

Le type chaine n'est pas prédéfini, il est utile de le définir :

```
typedef char chaine[30];
/* déclaration du type chaîne de 30 caractères, numérotés de 0 à 29, seuls 29 caractères sont
     utilisables à cause du caractère de fin */
```

L'affectation d'une chaîne à une valeur ne peut se faire en utilisant l'opérateur =, car il y aurait copie d'adresse uniquement et de plus, un tableau est un pointeur constant. L'affectation est donc réalisé à l'aide du sous-programme `strcpy` de la bibliothèque standard.

```
char a[10];
strcpy(a,"Crac"); /* a = "Crac" est impossible */

typedef char couleur[10];
couleur  drapeau[3];
/* drapeau  est un tableau de 3 pointeurs sur des caractères */

strcpy(drapeau[0], "bleu");
strcpy(drapeau[1], "blanc");
strcpy(drapeau[2], "rouge");

/* ou */
char *drapeau [3] = { "bleu","blanc","rouge" };
```

D'autres sous-programmes (comparaison, concaténation, *etc.*) de manipulation de chaînes de caractères sont disponibles dans la bibliothèque standard `string.h` (*cf.* §2.3).

1.4.3 Structures

Définition

Le mot réservé **`struct`** permet de définir son propre type structuré composé de champs [1].

```
struct nom_de_la_structure { déclarations_des_champs ; };
```

Exemples :

```
struct complexe {
        float x,y ; };

struct complexe a,b ; /* a et b sont deux variables de type complexe */

struct date {
```

1. Une structure est communément appelée *enregistrement*.

```
         char jour[10];
         char mois[12];
         int annee; }  d1, d2;
        /* d1 et d2 sont deux variables de type date */
```

Pour éviter de répéter le mot **struct** lors de la déclaration des variables, on peut définir un nouveau type à l'aide de **typedef** :

```
typedef struct {
          int dim;
          float t[30]; } vecteur;

vecteur v;

typedef struct {
          int jour;
          char mois[12];
          int  annee; } date;

date d;

typedef struct {
          char nom[20],prenom[20];
          date date_de_naissance; } individu;

individu  moi, lui, classe[30];

/* moi et lui  sont deux variables de type individu, classe un tableau de 30 individus */
```

On peut définir des structures récursives ou auto-référencées en utilisant typedef.

```
typedef struct { individu val;
          struct noeud *fils_droit;
          struct noeud *fils_gauche; } noeud ;

noeud  *arbre;      /* arbre est une variable pointeur sur un noeud  */
```

Affectation et initialisation

L'affectation s'effectue globalement sur tous les membres des structures.

```
d1=d2;
moi=lui;
```

On peut aussi initialiser une structure lors de sa déclaration :

```
individu toto = {"Dupond","Pierre",{12,"Mars",1975}};
```

Accès aux champs

L'accès aux champs se fait par l'ajout d'un point après l'identificateur suivi du nom du champ.

```
v.dim=2;
v.t[2]=12;

a.x=0;
a.y=1; /* a est le complexe 0+i */

d={30,"Mars",1976};

moi.date_de_naissance=d;
lui.date_de_naissance.jour=4;
classe[1].date_de_naissance.annee=1985;
```

L'accès aux champs d'une structure pointée peut être simplifié par la notation `->` : `X->Y` signifie `(*X).Y` soit le champ `Y` de la structure pointée par `X`.

```
individu *toto;   /* toto est un pointeur sur un individu */
toto=malloc(sizeof(individu));
toto->date_de_naissance.annee=1996
/* équivalent à :  (*toto).date_de_naissance.annee = 1996 */
```

1.4.4 Unions

Le mot réservé **union** permet de définir des variables pouvant contenir des valeurs de types différents.

```
typedef union {
          struct { int x,y; } cartesien ;
          struct { float rho,teta; } polaire;
          } complexe;

complexe a;
```

Un complexe peut contenir soit une structure de deux entiers, soit une structure de deux réels.

Ainsi, les notations suivantes sont licites : `a`, `a.cartesien`, `a.cartesien.x`, `a.cartesien.y`, `a.polaire`, `a.polaire.rho`, `a.polaire.teta`.

```
typedef enum { type_cercle = 1 , type_rectangle } type_dessin;

typedef struct {
          type_dessin  figure;
          union {
            struct { float centre_x,centre_y,rayon; } cercle ;
            struct { float centre_x,centre_y,longueur,largeur ;} rectangle;
            } primitive_dessin;
          } objet_graphique;

objet_graphique c,r;
```

On peut donc écrire : `c.figure`, `c.primitive_dessin`, `c.primitive_dessin.cercle`, `c.primitive_dessin.cercle.rayon`, `r.figure`, `r.primitive_dessin`, `r.primitive_dessin.rectangle`, `r.primitive_dessin.rectangle.largeur`.

1.5 Structures de contrôle

1.5.1 Blocs et portée des déclarations

Une instruction est une expression suivie d'un point-virgule.

```
i=1;
k++;
l=produit(a,b);
```

Un bloc est une suite d'instructions entre deux accolades. Les instructions sont exécutées en séquence, dans l'ordre de leur écriture. Un bloc est une instruction composée. Par conséquent, partout où une instruction est licite, on peut placer un bloc.

```
{
  i=1;
  k++;
  l=produit(a,b);
}
```

Une variable peut être déclarée dans n'importe quel bloc, sous les conditions suivantes :
- elle doit être déclarée avant toute instruction du bloc,
- sa portée (son existence) est limitée au bloc où elle est déclarée.

Elle est alors connue et accessible dans le bloc et les blocs emboîtés.

```
/* Exemple 1 */
{
  int i;
  i=1;
  ..
  {
    float k=0 ;
    k=i+1;
    ...
  }
  i=2;
  ...
}
```

```
/* Exemple 2 */
{
  int i;
  ...
  i=0;
  {
    int i=10; /* on a ici un i propre au bloc emboîté */
    ...
  }
  /* on retrouve ici le i du bloc englobant qui vaut 0 */
}
```

1.5.2 Instructions de choix : if, switch

Instruction if

```
if (expression)
    instruction1;
else
    instruction2;
```

L'expression peut être de type entier, énuméré, réel ou pointeur. L'instruction 1 est exécutée si l'expression a une valeur non nulle, l'instruction 2 est exécutée si l'expression a une valeur nulle.

La partie **else** est facultative.

```
if (i==j)
  i++;

if (a>b)
  max=a;
else
  max=b;

if ((a>b)&&(i>j))
{
  a=i;
  b=j;
}
else
{
  a=j;
  b=i;
}
```

Instruction switch

C'est une instruction à choix multiples. Elle permet de remplacer des séquences ou des cascades de `si`.

```
switch ( expression )
{
  case constante1 : instructions1;
  case constante2 : instructions2;
  ...
  case constanten : instructionsn;
  default         : instructions;
}
```

L'expression doit être de type entier ou énuméré. Le type des constantes doit être le même que celui de l'expression.

Suivant la valeur de l'expression, le branchement s'effectue au cas correspondant et *tout* ce qui suit est exécuté jusqu'à la rencontre d'un **break**. Si aucune constante n'est égale à l'expression, le branchement se fait sur **default**. Ce cas **default** est facultatif.

```
char c;
int i=0,j=0,k=0;
...
switch (c)
{
  case 'a' : i=1;
  case 'b' : j=1;
  case 'c' : k=1; break;
  case 'q' : j=2; break;
  default  : k=2;
}
```

On peut grouper plusieurs constantes, comme dans l'exemple suivant :

```
int i,k;
...
switch ( i%6 )
{
  case 0 : k++; break;
  case 1 :
  case 2 : k--; break;
  case 3 :
  case 4 :
  case 5 : k*=2; break;
}
```

1.5.3 Instructions d'itérations : while, do, for

Instruction while

```
while (expression)
    instruction;
```

L'instruction est exécutée tant que l'expression a une valeur non nulle.

```
int compteur=0;
while (compteur<10)
{
  ...
  compteur++;
}
```

Instruction do

```
do
    instruction;
while (expression);
```

Contrairement au **while**, l'instruction est exécutée avant la première évaluation de l'expression.

```
s=0;
i=1;
do
{
  s+=a[i]*b[i];
  i++;
}
while (i<=10);
```

Instruction for

```
for (expression1 ; expression2 ; expression3)
    instruction ;
```

Exemples :

```
s=0;
for (i=1;i<=n;i++)
  s+=a[i]*b[i];

for (i=0;i<10;i++)
{
  if (a[i])
    k=0;
  a[i]=i;
}
```

Remarques :

– cette instruction est équivalente à :

```
expression1;
while (expression2)
{ instruction;
  expression3
};
```

– une boucle **for** peut utiliser une variable de type énumérée,

– les expressions et l'instruction peuvent être vides, c'est-à-dire omises (mais les points-virgules restent !),

```
s=0;
i=1;
for ( ;  i<=n; i++)
  s+=a[i]*b[i] ;

s=0;
for (i=1; ; i++)
{
  if (i>n)
    break;
  s+=a[i]*b[i];
}
/* le break est le seul moyen de sortir d'une telle boucle */

s=0;
for (i=1; i<=n; )
{
  s+=a[i]*b[i];
  i++;
}

for (s=0, i=1; i<=n; s+= [i]*b[i], i++)
  ;
```

L'instruction **break** permet de sortir de l'itération (**while**, **do** ou **for**). Elle correspond à un **goto** à la première instruction suivant l'itération.

L'instruction **continue** permet d'ignorer le reste des instructions du corps de l'itération et à poursuivre cette dernière.

```
s=0;
for (i=1; i<=n ; i++)
{
  if (s>max)
    break;
  else
    s+=a[i];
}

i=0;
s=0;
while (i<=n+1)
{
  i++;
  if (i%2)
    continue;
  s+=a[i];
}
```

1.5.4 Instructions de ruptures de séquence

Instruction goto

Toute instruction peut être précédée d'une étiquette (suivie de :) à laquelle peut se référer une instruction **goto**. L'étiquette et les **goto** s'y référant doivent appartenir à la même fonction.

```
i=1;
...
if (k==i)
  goto ailleurs;
...
ailleurs : i=2;
...
```

Instruction break

Comme cela a été déjà présenté, **break** est une instruction qui permet de sortir d'une instruction itérative (**while**, **do**, **for**) ou d'une instruction de choix multiple (**switch**).

Instruction exit

La fonction **void** `exit(`**int** `valeur);` termine l'exécution du programme. Elle peut se placer n'importe où dans le programme ou dans une fonction.

Par convention, une valeur nulle (on peut aussi utiliser la constante nulle `EXIT_SUCCESS`) signifie que le programme s'est terminé normalement, une valeur non nulle signifie que le programme s'est terminé avec une erreur (constante `EXIT_FAILURE`).

1.6 Fonctions

1.6.1 En-tête d'une fonction (ou signature, ou prototype)

```
type  nom_de_la_fonction ( liste_de_paramètres_formels ) ;
```

Les paramètres de la liste sont séparés par des virgules. La liste des paramètres peut être vide (**void**).

Une fonction retourne une valeur du type indiqué. Si le type de la fonction n'est pas précisé, la fonction est par défaut de type **int**.

Une fonction peut ne rien retourner (procédure) ; elle sera alors déclarée de type **void**.

```
int puissance (int x,int y);
/* retourne l'entier x puissance y */

void message_d_alerte(void);
/* ne retourne rien */

complexe *add_complexe(complexe a,complexe b);
/* retourne un pointeur sur le résultat complexe a+b */
```

Le nom des arguments est facultatif dans les prototypes de fonctions ; les types suffisent.

```
int puissance(int,int);

complexe *add_complexe(complexe,complexe);
```

L'en-tête d'une fonction peut être placée avant son utilisation, la fonction elle-même pouvant alors être décrite en dehors du programme ou dans un autre fichier.

1.6.2 Définition d'une fonction (corps)

```
type  nom_de_la_fonction ( liste_de_paramètres_formels )
{
    déclarations;
    instructions;
}
```

La valeur retournée se fait par l'instruction **return** qui provoque ensuite la sortie de la fonction.

```
return expression;
```

Les fonctions peuvent naturellement être récursives.

```
int puissance ( int x, int y ) /* retourne l'entier x puissance y */
{   int   i , r  = 1;
    for (i=1 ; i<=y ; i++) r*=x;
    return r;
}

typedef  struct { float  x,y ; }  complexe ;

complexe   *add_complexe ( complexe a , complexe b )
{   complexe  *r ;
    r=malloc(sizeof(complexe));
    r->x = a.x+b.x;   r->y = a.y+b.y;
    return r;
} /* retourne un pointeur sur le résultat complexe a+b */

complexe sous_complexe ( complexe a , complexe b )
{   complexe   r ;
    r.x = a.x-b.x;   r.y = a.y-b.y;
    return r;
} /* retourne le résultat complexe a-b */
```

1.6.3 Passage des paramètres

En C, le seul type de passage de paramètres est le passage par valeur. La valeur d'une variable passée en paramètre ne peut donc être modifiée !

```
int fois_deux(int i)
{
  i=i*2;
  return i;
}

/* L'appel : */
i=3;
j=fois_deux(i);
/* ne modifie pas la valeur de i. */

void echange float x , float y )
{
  float z=x;
  x=y;
  y=z;
}

/* L'appel : */
echange(a,b);
/* n'échange pas les valeurs de a et b. */
```

Donc, si l'on veut modifier des paramètres ou calculer plusieurs résultats par une fonction, il faut transmettre l'adresse de ces paramètres. Les paramètres formels de la fonction seront donc des pointeurs et les paramètres effectifs des constantes adresses ou des variables pointeurs.

```
int fois_deux(int *i)
{
  *i=*i*2;
  return *i;
}

/* L'appel : */
i=3;
j=fois_deux(&i);

/* modifie la valeur de i. */

void echange(float *x,float *y)
{
  float z=*x;
```

```
  *x=*y;
  *y=z;
}

/* L'appel : */
echange(&a,&b);
/* échange les valeurs de a et b. */
```

Le mot réservé **const** dans la signature d'une fonction permet de préciser que le contenu du paramètre ne sera pas modifié.

```
void ecrire ( const char  message[] );

float prod_scalaire ( const  float  a[] , const float  b[] , int  n );
```

Rappelons que l'identificateur de tableau est assimilé à un pointeur. Par conséquent, un tableau passé en paramètre est forcément passé par adresse.

```
void mult_2_tab ( float  a[20] , int  n)
{   int i;
    for ( i =1 ; i<=n ; i++)  a[i] = 2*a[i] ;
}

/* L'appel : */
mult_2_tab(t,10);
/* va modifier le tableau t. */
```

On peut utiliser cette particularité pour les chaînes de caractères implantées comme tableaux. Cela permet également de ne pas dimensionner les tableaux en paramètres :

```
void mult_2_tab ( float  a[] , int  n)
```

1.6.4 Programme principal

Le programme principal est lui-même une fonction de nom `main()` qui sera exécutée en premier. La structure générale d'un programme C peut donc être :

```
déclarations

prototypes de fonctions

type main() /* par défaut, le type est int */
{ déclarations
  instructions
}

définitions des fonctions
```

Exemple :

```
typedef  struct { float  x,y ; }  complexe ;
complexe   *add_complexe ( complexe a, complexe b ) ;
int   fois_deux ( int *i ) ;
void  echange (float *x , float *y ) ;

int main()
{   int i=3 , j;  float a=5 , b=123.25;
    j=fois_deux(&i) ;   printf("i = %d \n",i);
    echange(&a,&b); printf("a = %f \n",a);  printf("b = %f \n",b);
    return 0;
}

int fois_deux ( int *i )  { *i = *i*2;  return *i; }
void echange (float *x , float *y )   { float  z=*x  ; *x=*y ; *y=z; }
complexe *add_complexe ( complexe a, complexe b )
{ /* corps de la fonction add_complexe */
}
```

L'arrêt du programme peut se faire par la fonction `exit(v)` (*cf.* §1.5.4).

La fonction `main()` possède trois paramètres disponibles. Son prototype officiel est :

```
int main ( int  argc , char *argv[] , char *envp[]   ) { ... }
```

Chacun de ces paramètres est facultatif. Ces paramètres permettent de récupérer et traiter des arguments donnés sur la ligne de commande lors du lancement du programme exécutable :

- `argc` est le nombre d'arguments sur la ligne de commande ;
- `argv` est un tableau de pointeurs sur des caractères contenant les arguments de la ligne de commande (`argv[0]` contient le nom du programme exécutable) ;
- `envp` est un tableau de pointeurs vers les variables de l'environnement.

```
/* Soit le programme exécutable prog lancé par la commande
   C:\>prog 1   claude

   argc  vaut  3
   argv[0]  pointe sur la chaîne  "prog"
   argv[1]  pointe sur la chaîne  "1"
   argv[2]  pointe sur la chaîne  "claude" */

void main ( int argc ,  char *argv[])
{   if  (argc ==0) exit;
        switch  ( argv[1][0] )
        {   case '1' : traiter (  argv[2] );
            case '2' : ...
        ...   }
```

1.6.5 Pointeurs sur fonctions

Bien que le nom d'une fonction ne soit pas une variable, on peut définir un pointeur sur une fonction de la façon suivante :

```
type  ( *nom_de_la_fonction )  ( liste_de_paramètres_formels );
```

Ceci est souvent utilisé pour faire des tableaux de fonctions ou passer des fonctions en paramètres.

```
float  mult ( float x , float y ) { return x*y ;} ;
float  add  ( float x , float y ) { return x+y ;} ;
float  sous ( float x , float y ) { return x-y  ;} ;

float  operation_fois_mille ( float  (*f) ( float  , float ) , float x , float y)
{ return  (*f)(x,y)*1000 ;};

int main()
{   float a=2 , b=3 , c ;
    float (*t[3])( float , float );

    t[0] = mult;  t[1] = add;  t[2] = sous;

    c = (*t[1]) ( a , b );              /* c reçoit add(a,b) */
    c = operation_fois_mille(sous,a,b); /* c reçoit sous(a,b)*1000 */
    return 0;
}
```

1.6.6 Nombre variable de paramètres

C propose une façon d'écrire des fonctions ayant un nombre indéterminé de paramètres, comme la fonction `printf(char *format , ... )`.

Le fichier à inclure `stdarg.h` contient les définitions du type `va_list(liste variable d'arguments )` et de trois fonctions de manipulation de la liste :

`va_start(v,premier)`	fait pointer v sur le premier argument ;
`va_arg(v type)`	retourne l'argument pointé et incrémente le pointeur v (type est le type de l'argument à retourner) ;
`va_end(v)`	doit être appelée en final pour remettre en ordre les pointeurs.

Exemple :

```
int somme(int, ... );
// calcule la somme d'une liste non vide d'entiers se terminant par 0
// ...  est l'ellipse pour une liste d'arguments de taille variable

int somme(int e, ... )
{
    int s,i;
```

```
    va_list v;
    va_start(v,e);
    s=e;
    if (e!=0)
        while ((i=va_arg(v,int))!=0)
            s+=i;
    va_end(v);
    return s;
}

...
j=somme(0);
j=somme(1,2,0);
j=somme(2,4,6,8,10,12,0);
```

1.7 Classes mémoire des variables

Le langage C propose différentes classes d'allocation mémoire pour les variables. Ces classes vont différencier la visibilité, la durée de vie et l'initialisation par défaut des variables.

1.7.1 Variable automatique

C'est une variable déclarée dans un bloc. Le mot clé est **auto**. C'est la déclaration par défaut lorsqu'aucun mot clé ne spécifie la déclaration.

La variable est allouée dans la pile d'exécution. La durée de vie et la visibilité de cette variable sont limitées au bloc. L'initialisation sera effectuée à chaque entrée dans le bloc si une initialisation est indiquée ; il n'y a pas d'initialisation par défaut.

1.7.2 Variable globale

Une variable globale est déclarée à l'extérieur de tout bloc et de toute fonction. Elle est définie de sa déclaration jusqu'à la fin du fichier. Accessible de n'importe quel endroit du programme, sa durée de vie est le temps d'exécution du programme lui-même.

Pour y accéder depuis un autre fichier, il faudra la déclarer externe dans ce dernier. Elle est stockée dans le segment de données du programme et est initialisée à 0 par défaut au chargement du programme.

1.7.3 Variable externe

Le mot clé **extern** permet de déclarer une variable localement, mais ne lui réserve pas de place mémoire. On peut ainsi accéder à une variable globale définie dans un autre fichier.

extern est également utilisé pour accéder à une fonction définie dans un autre fichier [2].

1.7.4 Variable statique

Déclarée par le mot clé **static**, elle ressemble à une variable globale. Sa durée de vie est le temps d'exécution du programme lui-même. Elle est stockée dans le segment de données du programme et est initialisée à 0 par défaut au chargement du programme.

Mais sa visibilité est plus localisée :
- définie à l'intérieur d'un bloc, elle n'est visible que dans le bloc ;
- définie à l'extérieur de tout bloc, elle n'est visible que dans le fichier et ne peut être accédée à partir d'un autre fichier.

Elle conserve sa valeur d'une exécution du bloc à l'autre.

1.7.5 Variable registre

La déclaration d'une telle variable doit être faite à l'intérieur d'un bloc par le mot clé **register**. Le but est de spécifier au compilateur de conserver cette variable dans un registre (si c'est possible) dans un souci d'optimisation de temps d'exécution.

La taille de la variable doit bien sûr être inférieure à la taille d'un registre du processeur et son adresse est inaccessible.

2. L'inclusion de fichiers d'en-têtes permet d'éviter ces re-déclarations de fonctions.

1.7.6 Exemple

```
int  g;     /* globale */

int f1();

main()
{
  int  a ;    /* automatique */
  register  int r ; /* registre */
  extern int e ;  /* externe */
  ...
  { int i = 1;  /* automatique */
    ...
   g = i++;
   a=e;          /* e est une variable définie dans un autre fichier */
  }
  ...
  a=a+g;
}

int f()
{
  int i;    /* automatique */
  static int n=1; /* statique */    /* initialisée une seule fois et non à chaque appel de f,
                n conserve sa valeur entre deux appels de f */
  ...
  n++;
  i=g;
}
```

1.8 Préprocesseur

Le préprocesseur réalise une phase de précompilation dont le but est d'inclure des fichiers, de définir des constantes symboliques et des macro-instructions et de réaliser des compilations conditionnelles.

Les instructions pour le préprocesseur commencent par un **#**.

```
#instruction   arguments
```

Les instructions possibles sont : `include`, `define`, `undef`, **`if`**, `elif`, **`else`**, `endif`, `ifdef`, `ifndef`.

1.8.1 Inclusion de fichiers

```
#include  <nom_fichier> /* inclus un fichier du répertoire standard du C */
#include  "nom_fichier" /* inclus un fichier quelconque */
```

L'inclusion est utilisée pour insérer des fichiers contenant des modules complets ou des fichiers d'entêtes (traditionnellement suffixés `.h`).

```
#include <stdio.h>
#include <stdlib.h> /* permet d'utiliser tous les types, constantes et fonctions */
#include <string.h> /* de la bibliothèque standard  */
#include <math.h>   /* dont les en-têtes sont dans ces fichiers */
#include "vecteur.h"
#include "matrice.h"
```

1.8.2 Constantes symboliques et macro-instructions

L'instruction **`#define`** permet de définir des constantes symboliques, d'associer une chaîne de caractères à un identificateur ou d'écrire des macro-instructions.

```
#define  identificateur
#define  identificateur   chaîne de caractères
#define  identificateur   ( paramètres )  chaîne de caractères
```

Le préprocesseur remplacera alors l'identificateur par la chaîne de caractères.

```
#define ERREUR    /* définit la constante symbolique  ERREUR */

#define boolean  short

#define  Max  200
float  a[Max] ;     /* max  est remplacé par 200 */

#define  min(a,b)  (a<b)?a:b
i = min(x,y) ;      /* min(x,y) sera remplacée par   (x<y)?x:y  */

#define  suffixage(nom)  #nom ".cpp"
ch=suffixage(toto)    /* suffixage(toto) sera remplacé par  toto.cpp */
```

Attention aux pièges classiques, par exemple :

```
#define  carre(a)  a*a
x=2;  r = carre(x);   /* r = x*x     => r vaut 4 */
x=2;  r = carre(x+1); /* r = x+1*x+1 => r vaut 5 et non 9 */
x=2;  r = carre(x++); /* r = x++*x++ => r vaut 6 et non 9 et x vaut 4 et non 3 */
```

La commande **#undef** identificateur permet de supprimer la définition d'une constante symbolique.

```
#undef ERREUR
```

1.8.3 Compilation conditionnelle

La compilation conditionnelle peut se faire grâce aux commandes du préprocesseur suivantes :

```
#if   expression_constante
#else
#endif
#elif   expression_constante   /* équivalent  else if */
```

Exemple :

```
#if    SYSTEME == SUN
#define  Fichier_d_en_tete  "sun.h"
#elif  SYSTEME == MSDOS
#define  Fichier_d_en_tete  "msdos.h"
#else    #define  Fichier_d_en_tete  "divers.h"
#endif
#include  Fichier_d_en_tete
```

Les commandes **#ifdef** et **#ifndef** permettent de tester si une constante symbolique est définie ou non.

```
#ifndef  pi
#define  pi  3.14159
#endif

#ifndef matrice_h   /*  texte du fichier matrice.h */
#define matrice_h
... /* déclarations de matrice */
... /* et prototypes des fonctions */
#endif
```

Chapitre 2

Bibliothèques standards du C

Ce chapitre présente une sélection des possibilités offertes par la bibliothèque standard du C.

2.1 Entrées/sorties sur les fichiers standards

Les fonctions d'entrées-sorties sont accessibles par l'inclusion **#include** <stdio.h>.
Trois unités standards d'entrée-sortie sont ouvertes à l'exécution d'un programme :
- stdin l'entrée standard ;
- stdout la sortie standard ;
- stderr l'erreur standard.

Sans redirection, par défaut stdin est associé au clavier, sdtout et stderr à l'écran.

2.1.1 Entrées/sorties de caractères

En-têtes :

```
int getchar(void);
/* retourne un caractère lu sur l'entrée standard (par défaut le clavier) */

int putchar(int c);
/* écrit le caractère c sur la sortie standard (par défaut l'écran) */
```

Exemple d'utilisation :

```
char c;
c=getchar();
if (c!='\n')
    putchar(c);
else
    putchar('F');
```

2.1.2 Entrées/sorties de chaînes de caractères

En-têtes :

```
char *gets(char *s);
/* lit une chaîne de caractères sur l'entrée standard et la range dans s (le caractère de fin
     de ligne est remplacé par le caractère de fin de chaîne '\0') */

int puts(const char *s);
/* écrit la chaîne s suivie d'un retour à la ligne sur la sortie standard */
```

Exemple d'utilisation :

```
char s[20];
gets(&s);
puts(s);

char *ss;
ss=malloc(20);
gets(ss);
puts(ss);
```

2.1.3 Entrées/sorties formatées

`scanf()` et `printf()` sont utilisées pour entrer et sortir des variables de types quelconques, suivant un format précisé.

En-têtes :

```
int scanf(const char *format,liste_d_adresses_de_variables);
/* range dans la liste de variables les valeurs lues sur l'entrée standard,
   suivant le format précisé dans la chaîne de caractères
   et retourne le nombre de valeurs correctement affectées. */

int printf(const char *format,liste_d_expressions);
/* écrit les valeurs des expressions sur la sortie standard,
  suivant les formats répartis dans la chaîne de caractères
  et retourne le nombre de caractères transmis en sortie. */
```

Si `liste_d_expressions` est vide, `printf` écrit la chaîne de caractères contenue dans le format (par exemple `printf("Hello world !");`).

Un élément de format général est de la forme `%[aligne][taille][.précis][T][type]` où les différents éléments (facultatifs) sont :

- `aligne` indique l'alignement souhaité :
  - `+` les valeurs numériques seront précédées de + ou -
  - `-` alignement à gauche (par défaut, alignement à droite)
- `taille` indique le nombre de caractères minimum ;
- `précis` indique la précision retenue pour l'affichage :
  nombre minimum de chiffres pour les types entiers (`i`, `d`, `o`, `u`, `x`, `X`)
  nombre de chiffres après la virgule pour les types réels (`e`, `E`, `f`, `F`)
  nombre maximum de chiffres significatifs pour le type réel (`g`, `G`)
  nombre maximum de caractères pour le type chaîne de caractères (`s`)
- `T`
  - `l` indique une valeur entière au format long pour les types entiers (`i`, `d`, `o`, `u`, `x`, `X`) ou une valeur réelle au format double pour les types réels (`e`, `f`, `g`)
  - `h` indique une valeur entière au format court pour les types entiers (`i`, `d`, `o`, `u`, `x`, `X`)
- `type` indique le type de l'affichage :

`c`	caractère
`s`	chaîne de caractères
`i` ou `d`	entier décimal
`x` ou `X`	entier hexadécimal
`u`	entier non signé
`f` ou `F`	réel en virgule fixe
`e` ou `E`	réel en virgule flottante
`g` ou `G`	le plus court des types réels
`p`	pointeur

Rappel : le caractère `'n'` fait changer de ligne.

Exemple :

```
int i;
float r;
char *ch,*form;

printf("entrer la valeur de i : ");
scanf("%d",&i);
printf("i  =  %d\n",i);

ch=(char *)malloc(20*sizeof(char));

scanf("%d%f%s", &i , &r , ch );
printf("i  =  %5d  r = %10.3e  ch =%s\n", i,r,ch);
/* équivalent à : form = "i  =  %5d  r = %10.3e  ch =%s " ;  printf(form, i,r,ch);  */
```

Les sorties sur l'écran sont :

```
entrer la valeur de i : 2
i  =  2
```

```
-3 +123.45 Claude
i  =     -3  r =  1.234e+02  ch =Claude
```

Remarque : dans un `scanf`, la chaîne de caractères du format ne peut contenir du texte, mais uniquement des éléments de format.

2.2 Entrées/sorties sur les fichiers

2.2.1 Accès aux fichiers

Un fichier est accessible par un pointeur de fichier déclaré par : `FILE * fichier` (`FILE` est défini dans `stdio.h`).

L'ouverture de fichier se fait par la fonction `fopen()`, la fermeture par `fclose()`.

En-têtes :

```
FILE * fopen(const char *nom_de_fichier,const char *mode_d_ouverture);
/* retourne un pointeur de fichier; si le fichier n'a pu être ouvert, retourne NULL. */

int fclose (FILE *fichier);
/* ferme le fichier (après une mise à jour réelle du fichier physique)
   retourne la constante pré-définie EOF en cas d'erreur, 0 sinon. */

int feof(FILE *fichier);
/* retourne une valeur non nulle si l'indicateur de fin de fichier est positionné. */
```

Les modes d'ouverture sont :

`"r"`	ouverture en lecture
`"w"`	ouverture en écriture (écrasement si le fichier existe, création sinon)
`"a"`	ouverture pour ajout en fin de fichier
`"r+"`	ouverture en lecture/écriture (mise à jour de fichier existant)
`"w+"`	ouverture en lecture/écriture (écrasement si le fichier existe, création sinon)
`"a+"`	ouverture en lecture n'importe où et en écriture en fin de fichier

Remarque : `b` indique qu'il s'agit d'un fichier binaire (`"rb"`, `"wb"`, `"ab"`, `"r+b"`, `"w+b"`, `"a+b"`).

Exemple :

```
FILE *fic;
fic=fopen("données","r");
if (fic==NULL)
{
    /*erreur en ouverture, fichier non trouvé */
}
else
{
    while (!feof(fic))
    {
        /* Lecture du fichier... */
    }
    fclose(fic);
}
```

2.2.2 Entrées/sorties de caractères dans un fichier

En-têtes :

```
int fgetc(FILE *fichier);
/* retourne la valeur entière du caractère courant pointé dans le fichier;
   retourne EOF si la fin de fichier est atteinte. */

int fputc(char c,FILE *fichier);
/* écrit le caractère c dans le fichier;
   retourne le caractère lui-même ou EOF en cas d'erreur. */
```

Remarque : `getchar()` est équivalent à `fgetc(stdin)`, `putchar(c)` est équivalent à `fgetc(c ,stdout)`.

Exemple d'utilisation :

```
/* lecture du fichier "essai" caractère par caractère i compte les caractères */
FILE *f;
char c;
int i=1;
f=fopen("essai","r");

if (f==NULL)
{
    printf("erreur");
    exit(1);
}

do
{
    c=fgetc(f);
    printf("%dème c = %c\n",i,c) ;
    i++;
}
while (c!=EOF);
fclose(f);
```

2.2.3 Entrées/sorties de chaînes de caractères dans un fichier

En-têtes :

```
char * fgets(char *s,int n,FILE *fichier);
/* lit au plus les n-1 caractères courants pointés dans le fichier, et les place dans la
    chaîne s; s'arrête si on rencontre un caractère de fin de ligne (qui est alors mis dans
    la chaîne s) ou de fin de fichier; la chaîne s est complétée par un '\0'; retourne s si
    la fin de fichier est atteinte, NULL en cas d'erreur. */

int fputs(const char s,FILE *fichier);
/* écrit la chaîne s  dans le fichier, retourne EOF en cas d'erreur */
```

Remarque : `gets(&s)` est équivalent à `fgets(&s,stdin)`, `puts(s)` est équivalent à `fgets(s, stdout)`.

Exemple d'utilisation :

```
/* lecture du fichier "essai" ligne par ligne (inférieures à 200 caractères) i compte les
    lignes */

FILE *f;
char *s ;
int i=1;
s=malloc(200);
if ((f=fopen("essai","r"))==NULL)
{
    printf("erreur");
    exit(2);
}
while (fgets(s,200,f)!=NULL)
{
    printf("%dème chaîne = %s\n",i,s) ;
    i++;
};
fclose(f);

/* lecture du fichier "essai" 5 caractères à la fois au plus i compte les groupes */
FILE *f;
char *s ;
int i=1;
s=malloc(6);
f=fopen("essai_texte","r");
if (f==NULL)
{
    printf("erreur");
    exit(2);
}
while (fgets(s,6,f) != NULL)
{
    printf("%dème chaîne = %s\n",i,s) ;
    i++;
```

```
};
fclose(f);
```

2.2.4 Entrées/sorties formatées dans un fichier

En-têtes :

```
int fscanf(FILE *fichier,const char *format,liste_d_adresses_de_variables);
/* fonctionne comme scanf() mais sur le fichier précisé retourne le nombre de valeurs
    correctement affectées. */

int fprintf(FILE *fichier,const char *format,liste_de_variables);
/* fonctionne comme printf() mais sur le fichier précisé retourne le nombre de caractères
    écrits dans le fichier. */
```

Remarque : `scanf(...)` est équivalent à `fscanf(stdin,...)`, `printf(...)` est équivalent à `fprintf(stdout,...)`.

Il existe des fonctions analogues pour lire et écrire dans des chaînes de caractères :

```
int sscanf(char *s,const char *format,liste_d_adresses_de_variables)
/* fonctionne comme scanf() mais dans la chaîne s */

int sprintf (char *s,const char *format,liste_de_variables)
/* fonctionne comme printf() mais dans la chaîne s */
```

2.2.5 Entrées/sorties non formatées (ou binaires) dans un fichier

Il s'agit ici de manipuler des blocs d'octets.

En-têtes :

```
size_t fread(void *ptr,size_t t,size_t n,FILE *fichier);
/* lit au plus n objets de taille t dans le fichier et les range dans le tableau ptr retourne
    le nombre d'objets correctement lus (erreur si ce nombre est inférieur à n) */

size_t fwrite(void *ptr,size_t t,size_t n,FILE *fichier);
/* écrit n objets de taille t du tableau ptr dans le fichier retourne le nombre d'objets
    correctement écrits (erreur si ce nombre est inférieur à n) */
```

`size_t` est le type entier non signé.

Exemple d'utilisation :

```
float a[30],b[10];
FILE *f;

/* écrit les 10 premiers réels du tableau a dans le fichier */
f=fopen(nom,"w");
fwrite(a,sizeof(float),10,f);
fclose(f);

/* remplit les 8 premiers éléments du tableau b par des réels lus dans le fichier essai */
f=fopen("essai","r");
fread(b,sizeof(float),8,f)
fclose(f);
```

On peut aussi, par exemple, s'en servir pour ranger et récupérer des structures dans des fichiers par un seul ordre de lecture/écriture.

La fonction `fseek` permet l'accès direct.

En-têtes :

```
int fseek(FILE * f,long deplacement,int origine);
/* positionne le pointeur de fichier sur l'octet origine+deplacement retourne 0 si elle
    réussit, une valeur non nulle en cas d'erreur. */
```

Quelques constantes sont prédéfinies pour le paramètre origine :

`SEEK_SET` : début du fichier

`SEEK_CUR` : position courante

`SEEK_END` : fin du fichier

Exemple d'utilisation :

```
/* Lecture d'un fichier non formaté de réels. */

float x;
f=fopen("essai","r+");
fseek( f,(n-1)*sizeof(float),0); /* positionne le pointeur de fichier sur le n-ième réel du
    fichier */
fread(&x,sizeof(float),1,f); /* place dans x un réel, soit le n-ième du fichier */

x=x*10; /* modification de x */

fseek(f,(n-1)*sizeof(float),0); /* repositionne le pointeur de fichier sur le n-ième réel du
    fichier */
fwrite(&x,sizeof(float),1,f); /* modifie le n-ième réel du fichier */
```

Voir également les autres fonctions disponibles dans la bibliothèque standard (`stdio.h`), entre autres :

```
int ftell(FILE * f);
/* retourne la position courante du pointeur de fichier (-1 en cas d'erreur) */

void rewind(FILE * f);
/* équivalent à fseek(f,0,0) */
```

2.3 Outils sur les chaînes de caractères

La bibliothèque `string.h` fournit en autres des outils sur les chaînes de caractères qui permettent leur manipulation sans se soucier de leur implantation. Les principaux outils sont :

En-têtes :

```
unsigned int strlen(const char* ch);
/* retourne la longueur (nombre de caractères) de la chaîne ch */

char* strcpy(char* ch1, const char* ch2);
/* procédure qui copie la chaîne ch2 dans la chaîne ch1 réalise l'affectation de tableau
    impossible en C : ch1 = ch2 */

char* strcat(char* ch1, const char* ch2);
/* procédure qui concatène la chaîne ch2 au bout de la chaîne ch1, ch1 est donc modifiée */

int strcmp(const char* ch1, const char* ch2);
/* fonction qui compare les deux chaînes ch1 et ch2 par ordre alphabétique, retourne une
    valeur négative si ch1<ch2, nulle si ch1==ch2 et positive si ch1>ch2 */
```

Exemple d'utilisation :

```
chaine A, B, C;

printf("Entrer un mot : ");
scanf("%s",A);
strcpy(B,"Ens2m Besançon");
strcpy(C,"Ensmm");

if (strlen(A)+strlen(B)<29)
  strcat(A,B);

if (strcmp(A,C)<0)
  printf("Erreur\n");
else
  printf("A = %s\n",A);
```

Le type chaine est défini par **`typedef char`** `chaine[30];`.

2.4 Outils mathématiques

La bilbiothèque `math.h` fournit les fonctions mathématiques de bases.

En-têtes :

```
double exp(double x);
/* exponentielle de x */
```

```
double log(double x);
/* logarithme naturel de x */

double log10(double x);
/* logarithme en base 10 de x */

double pow(double x, double y);
/* x puissance y */

double sqrt(double x);
/* racine carrée */

double fabs(double x);
/* valeur absolue de x */
```

Pour les fonctions trigonométriques les angles sont en radians.

En-têtes :

```
double sin(double x);
/* sinus de x */

double cos(double x);
/* cosinus de x */

double tan(double x);
/* tangente de x */

double asin(double x);
/* arcsinus de x */

double acos(double x);
/* arccosinus de x */

double atan(double x);
/* arctangente de x */

double atan2(double y, double x);
/* arctangente de y/x */

double sinh(double x);
/* sinus hyperbolique de x */

double cosh(double x);
/* cosinus hyperbolique de x */

double tanh(double x);
/* tangente hyperbolique de x */
```

2.5 Gestion des erreurs

Pour gérer simplement les erreurs lors de l'exécution, il existe la fonction standard `assert(expression)` accessible avec **`#include`** `<assert.h>`.

Si l'expression vaut 0, `assert` va afficher un message d'erreur sur la sortie standard et arrêtera le programme par un appel à `abort()`.

Exemple :

```
assert(a%2==0);     // nécessite que a soit pair

assert(i<=C.dim);   // nécessite que i soit ou égal à la dimension
assert(v.n==u.n);   // nécessite u et v de même dimension

assert(f!=NULL);    // vérifie si un fichier f s'est ouvert correctement
```

On peut inhiber la fonction `assert` en définissant la variable symbolique `NDEBUG`.

On peut à l'aide d'`assert()` se fabriquer une fonction regroupant tous les invariants que doit vérifier une variable et ainsi l'appeler à chaque manipulation de la variable.

Chapitre 3

Langage C++

Le véritable apport du C++ est l'implantation des notions de classes et d'objets. Auparavant, nous allons passer en revue quelques nouveautés de C++ par rapport au C.

3.1 Quelques nouveautés du C++

3.1.1 Commentaires

Un commentaire C++ commence par les caractères // et se termine en fin de ligne.

```
//  Cette ligne est un commentaire
//  Programme d'essai
void test(int x);     // teste la valeur de x
```

3.1.2 Déclarations

En C++, les déclarations sont devenues exécutables ; ceci permet de mélanger déclarations et instructions et de ne plus respecter la structure `bloc = { déclarations ; instructions }`.

```
main()
{
    int j;
    j=2;
    ...
    for (int i=1; i<2*j; j++)
    {
        char c;
        ...
    }
    double x=3.14;
    ...
}
```

3.1.3 Opérateur de portée : :

L'opérateur de portée permet d'accéder à une variable globale normalement masquée par une variable locale de même nom.

```
int i;        //  i global
main()
{
  int i;      //  i local
  {
    ...
    if ( i == ::i ) //  teste l'égalité entre le i local et le i global
      ... ;
  }
}
```

Il permettra surtout de préciser l'appartenance d'une méthode à une classe ou l'appartenance d'une classe à un espace de nom.

3.1.4 Espace de nom

Le C++ introduit le concept d'espace de nom pour permettre l'utilisation d'un même nom de classe ou de fonctions dans plusieurs bibliothèques.

Les classes, les types et les fonctions peuvent être déclarés dans un espace de nom grâce à la commande **namespace**.

```
namespace perso
{
    class Vecteur2D { ... };

    double produitScalaire(Vecteur2D U,Vecteur2D V);
}
```

Les éléments ainsi définis sont accessibles par l'intermédiaire de l'opérateur de portée :

```
...
perso::Vecteur2D A,B;
...
perso::produitScalaire(A,B);
...
```

L'instruction **using** permet de spécifier un emploi systématique d'un espace de nom pour une classe ou une fonction :

```
using perso::Vecteur2D; // dorénavant Vecteur2D désigne perso::Vecteur2D

using perso::produitScalaire; // dorénavant produitScalaire désigne
                              // perso::produitScalaire
```

On peut aussi demander l'emploi systématique d'un espace de nom pour l'ensemble d'un espace de nom :

```
using namespace perso; // dorénavant Vecteur2D désigne perso::Vecteur2D et
                       // produitScalaire désigne perso::produitScalaire
```

Remarque : la bibliothèque standard du C++ est définie dans l'espace de nom `std` (*cf.* §4).

3.1.5 Opérateur de gestion mémoire

Les opérateurs **new**, **delete** et **delete** `[]` (opérant sur des pointeurs) permettent dynamiquement de créer des objets et de libérer la place mémoire occupée par ces objets. Ils remplacent l'utilisation des fonctions `malloc()` et `free()`.

L'opérateur **new** appelle automatiquement le constructeur par défaut une fois l'espace mémoire réservé.

Les opérateurs **delete** et **delete** `[]` appellent automatiquement le destructeur de chaque objet avant de libérer la mémoire.

```
int  *x;
x=new int;
*x=5;
*x=*x+1;
delete x;

double *y;
y=new double[20];
y[3]=3.14;
delete [] y;

Vecteur2D *v;
v=new Vecteur2D(3,5);
v->afficher();
delete v;
```

3.1.6 Type référence

C++ autorise la définition du type référence :

```
type &nom=valeur;
```

Exemple :

```
int i=0;
int &ref=i; // L'objet ref permet de se référer directement à i.
i++;        //  i = ref = 1, ref est devenu un synonyme de i
ref++;      //  i = ref = 2.
```

Un objet de type référence doit toujours être initialisé et ne se déréférence jamais.

3.1.7 Passage de paramètre par référence

La notion précédente de type référence autorise donc le passage de paramètre par référence.

```
void echange(double &x, double &y)
{
  double z=x;
  x=y;
  y=z;
}
```

L'appel `echange(a,b)` va effectivement échanger les valeurs de `a` et `b` (*cf.* §1.6.3).

3.1.8 Type booléen

Le C++ fournit un type booléen nommé **`bool`**. Un variable de type **`bool`** peut prendre deux valeurs : **`true`** ou **`false`**.

```
bool trouve;

trouve=false;

while (!trouve)
    ...
```

Remarque : des conversions implicites permettent d'utiliser également 0 pour **`false`** et toute valeur non nulle pour **`true`**.

3.1.9 Arguments facultatifs et par défaut

Les fonctions et méthodes acceptent dans leur prototype des arguments avec des valeurs par défaut. Lors de l'appel, si ces arguments sont absents, la fonction utilisera ces valeurs par défaut. *Ces arguments doivent être placés en fin de la liste d'arguments.*

```
double somme(double a,double b=1, double c=0); // prototype

double somme(double a,double b, double c)
{
    return a+b+c ;
}

q=somme(10,20,30);
q=somme(10,20); // équivalent à q=somme(10,20,0);
q=somme(10);    // équivalent à q=somme(10,1,0);
```

3.1.10 Surcharge des fonctions

C++ permet de définir plusieurs fonctions différentes avec le même nom.

```
void somme(void);
int somme(int,int);
double somme(double a,double b);
point3d somme(point3d,point3d);
```

Le compilateur se charge alors de déterminer l'appel de la bonne fonction, s'il n'y a pas d'ambiguïté, suivant le nombre et le type des paramètres (mécanisme de liaison statique).

```
int i;
double x;
point3d p,q,r;
i=somme(2,3);
x=somme(x,3.14);
q=somme(p,r);
```

Attention aux ambiguïtés ! La surcharge et les arguments facultatifs rendent le cas suivant interdit car ambigu :

```
double f(double x);
double f(double x,double y=1);

...
double a,b;
a=f(b); // erreur : l'appel de $f(b)$ n'est pas résolu
```

3.1.11 Fonctions en ligne

Les fonctions ou méthodes en ligne, déclarées par **inline**, remplacent avantageusement les macro-instructions et leurs pervers effets de bord (*cf.* §1.8.2).

La déclaration **inline** indique au compilateur de remplacer chaque appel de la fonction par son code dans le code généré.

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

x=2;
r=carre(x); // r=4 et x=2
x=2;
r=carre(x+1); // r=9 et x=2
x=2;
r=carre(x++); // r=4 et x=3
```

3.2 Classes et objets C++

3.2.1 Syntaxe générale

La syntaxe générale de déclaration d'une classe est la suivante :

```
class nom_de_la_classe
{
private:

    // champs et méthodes privés

protected:

    // champs et méthodes protégés

public:

    // champs et méthodes publics

};
```

Les membres (champs ou méthodes) déclarés en section **private** ne sont accessibles que par les membres de la classe. **private** est facultatif ; c'est le mode par défaut.

Les membres de la section **protected** sont accessibles par les membres de la classe et ceux des classes dérivées. Quant aux membres de la section **public**, ils sont accessibles par tous.

La méthode peut être définie à l'intérieur de la classe. Cependant, la classe peut ne contenir que les prototypes des méthodes, les méthodes étant définies à l'extérieur de la classe. Dans ce dernier cas, le nom de la méthode doit être précédé du nom de la classe suivi de `::`.

Comme les variables, les objets peuvent être globaux, statiques, automatiques ou constants. Seules les méthodes déclarées constantes (**const**) auront le droit de manipuler des objets constants.

Un champ déclaré **static** sera partagé par tous les objets de la classe. Il devra donc être initialisé par l'implantation d'une variable globale.

3.2.2 Constructeur et destructeur

C++ propose une méthode particulière appelée *constructeur* dont le nom est le nom même de la classe et qui sera invoquée à chaque définition d'objet.

De même, une méthode appelée *destructeur* sera invoquée automatiquement quand un objet devra être détruit, c'est-à-dire à la fin de sa durée de vie (fin de bloc ou appel de **delete**).

Le nom du destructeur est le nom de la classe précédé de ~. Le destructeur n'admet ni paramètre, ni valeur de retour.

3.2.3 Exemple

Déclaration d'une classe `Vecteur2D` :

```
class Vecteur2D
{
private :
    double abs,ord;

public :
    Vecteur2D(double abscisse=0,double ordonnee=0); // constructeur de moi-même, par défaut
        le vecteur nul

    double getAbscisse(void); // retourne l'abscisse de moi-même
    double getOrdonnee(void); // retourne l'ordonnée de moi-même

    void setAbscisse(double x); // modifie l'abscisse de moi-même par la valeur x
    void setOrdonnee(double y); // modifie l'ordonnée de moi-même par la valeur y

    void afficher(void); // affiche moi-même à l'écran sous la forme (abscisse, ordonnée)
    void acquerir(void); // acquiert moi-même au clavier

    double norme(void); // retourne la norme de moi-même
    double produitScalaire(Vecteur2D V); // retourne le produit scalaire de V et de moi-même
    Vecteur2D addition(Vecteur2D V); // retourne la somme de V et de moi-même

};
```

Définition des méthodes de la classe `Vecteur2D` :

```
Vecteur2D::Vecteur2D(double abscisse,double ordonnee)
{
    abs=abscisse;
    ord=ordonnee;
}

double Vecteur2D::getAbscisse(void)
{
    return abs;
}

double Vecteur2D::getOrdonnee(void)
{
    return ord;
}

void Vecteur2D::setAbscisse(double x)
{
    abs = abs + x;
}

void Vecteur2D::setOrdonnee(double x)
{
    ord = ord + x;
}

void Vecteur2D::afficher(void)
{
    cout<<"("<<abs<<";"<<ord<<")"<<endl;
}

void Vecteur2D::acquerir(void)
```

```
{
    cout<<"entrer abscisse puis ordonnee du vecteur : ";
    cin>>abs>>ord;
}

double Vecteur2D::norme(void)
{
    return sqrt(abs*abs+ord*ord);
}

double Vecteur2D::produitScalaire(Vecteur2D V)
{
    return abs*V.abs+ord*V.ord;
}

Vecteur2D Vecteur2D::addition(Vecteur2D V)
{
    return Vecteur2D(abs+V.abs,ord+V.ord);
}
```

Quelques instanciations et manipulations de `Vecteur2D` :

```
Vecteur2D   u,v(5,4),w;

u.acquerir();
cout<<"vecteur u = ";
u.afficher();
cout<<"vecteur v = ";
v.afficher();

w=u;
cout<<"vecteur w = u = ";
w.afficher();

cout<<"norme de u = "<<u.norme()<<endl;
cout<<"produit scalaire u.v = "<<u.produitScalaire(v)<<endl;

w=u.addition(v);
cout<<"vecteur u + v = ";
w.afficher();

Vecteur2D *z;
z = new  Vecteur2D(5);
*z=u;
cout<<"vecteur *z = u = ";
z->afficher();
delete z;
```

3.2.4 Pointeur this

Le mot clé **this** dans une méthode désigne l'adresse de l'objet sur lequel s'applique la méthode.
Ainsi, la méthode `Vecteur2D::getAbscisse(void)`, peux s'écrire :

```
double Vecteur2D::getAbscisse(void)
{
    return (*this).abs;
}
```

ou encore :

```
double Vecteur2D::getAbscisse(void)
{
    return this->abs;
}
```

L'utilisation de **this** est nécessaire lorsque l'on a explicitement besoin de l'objet courant en entier.

3.2.5 Méthodes et classes friend

La déclaration **friend** d'une méthode ou d'une classe permet de violer la protection des champs et des méthodes déclarés **private**.

Le mot clé **friend** doit apparaître dans le prototype, mais pas dans la définition, lorsque la définition est effectuée à l'extérieur de la classe.

```
class Vecteur2D
{
    ...
    friend class Base2D;
    ...
};

class Base2D
{
private:
    Vecteur2D U,V;
public:
    ...
    void afficher();
    ...
};

void Base2D::afficher()
{
    cout<<"("<<U.abs<<";"<<U.ord<<")x("<<V.abs<<";"<<V.ord<<")"<<endl;
}
```

La déclaration **friend** s'avère pratique lors de la surcharge d'opérateurs dont le premier argument n'est pas un objet de la classe. Elle permet notamment de définir des fonctions « banales », c'est-à-dire non liées comme méthodes à des objets d'une classe. Par exemple, on peut surcharger la définition de la fonction `produitScalaire` par :

```
class Vecteur2D
{
    ...
    friend double produitScalaire(Vecteur2D U,Vecteur2D V);
    ...
};

double produitScalaire(Vecteur2D U,Vecteur2D V)
{
    return U.abs*V.abs+U.ord*V.ord;
}
```

Ainsi, l'utilisateur aura deux façons de calculer le produit scalaire de ses vecteurs u et v :
- soit par `ps=u.produitScalaire(v);`, méthode associée à u,
- soit par `ps=produitScalaire(u,v);`, fonction habituelle indépendante de u et de v.

Il va de soi qu'une utilisation abusive de cette notion de ***friend*** *va à l'encontre des notions de modularité, d'encapsulation et d'abstraction des implantations qui devraient être le souci de toute programmation orientée objet.*

3.2.6 Surcharge des opérateurs

En C++, les opérateurs peuvent être surchargés, de même que les fonctions. Il suffit de le déclarer par le mot clé **operator**.

Presque tous les opérateurs peuvent être surchargés :

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

Par exemple, on peut compléter la classe `Vecteur2D` en ajoutant dans la déclaration :

```
class Vecteur2D
{
    ...
    Vecteur2D operator+(const Vecteur2D & V);  // renvoie le vecteur V + moi-même
    Vecteur2D operator*(double a); // retourne le vecteur égal à moi-même multiplié par un
        scalaire
    bool      operator==(const Vecteur2D & V); // retourne vrai si V est identique à moi-même
        , faux sinon
```

```
    ...
};

ostream & operator<<(ostream & F,Vecteur2D V); // envoie abscisse et ordonnée du vecteur V
    dans le fichier F
istream & operator>>(istream & F,Vecteur2D & V); // récupère le vecteur V depuis le fichier F
```

La définition de ces opérateurs surchargés peut être :

```
Vecteur2D Vecteur2D::operator+(const Vecteur2D & V)
{
    return Vecteur2D(abs+V.abs,ord+V.ord);
}

Vecteur2D Vecteur2D::operator*(double a)
{
    return Vecteur2D(a*abs,a*ord);
}

bool Vecteur2D::operator==(const Vecteur2D & V)
{
    return ((abs==V.abs) && (ord==V.ord));
}

ostream & operator<<(ostream & F,Vecteur2D V)
{
    F<<V.getAbscisse()<<" "<<V.getOrdonnee();
    return F;
}

istream & operator>>(istream & F,Vecteur2D & V)
{
    double x,y;
    F>>x>>y;
    V=Vecteur2D(x,y);
    return F;
}
```

Exemple d'utilisation :

```
cout<<"entrer un vecteur : ";
cin>>v;
cout<<"vecteur v = "<<v<<endl;

w=u+v;
cout<<"vecteur u + v = "<<w<<endl;

cout<<"vecteur u * 3.0 = "<<u*3.0<<endl;

cout<<"test u==v = "<<(u==v)<<endl;
```

Une convention existe pour distinguer les opérateurs unaires pré et post-fixés :
- la déclaration **void** `Vecteur2D::operator++(void)` désigne l'opérateur préfixé utilisé par `++u` ;
- la déclaration **void** `Vecteur2D::operator++(int)` est pour l'opérateur post-fixé utilisé par `u++`.

3.2.7 Classe canonique

En théorie, toute classe devrait au moins comporter un constructeur, un constructeur par copie, l'opérateur d'affectation (**operator**=) et un destructeur.

Si la classe ne contient pas d'éléments dynamiques instanciés par **new** dans le constructeur, le constructeur par copie, le destructeur et l'opérateur d'affectation peuvent ne pas être définis. Ils seront alors définis par défaut par le compilateur.

En revanche, si la classe contient au moins un élément dynamique instancié par **new** dans le constructeur, ces trois méthodes doivent être explicitement définies. L'oubli de cette règle provoque au mieux des fuites de mémoires et au pire des bogues particulièrement difficiles à repérer.

Exemple de classe vecteur contenant un tableau dynamique :

```
class Vecteur
{
private:
    int dim; // dimension du vecteur
    double *tab; // pointeur sur les éléments
public:
    Vecteur(int dimension=0); // Constructeur
    Vecteur(const Vecteur &V); // Constructeur de recopie
    ~Vecteur(void); // Destructeur
    Vecteur & operator=(const Vecteur &V); // Opérateur d'affectation
    ...
};

Vecteur::Vecteur(int dimension)
{
    dim=dimension;
    if (dim!=0)
        tab=new double[dim];
}

Vecteur::Vecteur(const Vecteur &V)
{
    dim=V.dim;
    if (dim!=0)
    {
        tab=new double[dim];
        for (int i=0; i<dim; i++)
            tab[i]=V.tab[i];
    }
}

Vecteur::~Vecteur(void)
{
    if (dim!=0)
        delete [] tab;
}

Vecteur & Vecteur::operator=(const Vecteur &V)
{
    if (dim==V.dim)
    {
        for (int i=0; i<dim; i++)
            tab[i]=V.tab[i];
    }
    else
    {
        if (dim!=0)
            delete [] tab;
        dim=V.dim;
        if (dim!=0)
        {
            tab=new double[dim];
            for (int i=0; i<dim; i++)
                tab[i]=V.tab[i];
        }
    }
    return *this;
}
```

3.3 Classes dérivées et héritage

3.3.1 Classes dérivées

Soit une classe A, dite classe de base. Une classe B est dite dérivée de la classe de base A et hérite de ses membres en la déclarant par :

```
class B : public A
{
    // membres propres à la classe B
};
```

Les membres propres peuvent être de nouveaux champs et de nouvelles méthodes.

Remarque : la déclaration **class** B **: protected** A rend les membres publics de A protégés dans B ; la déclaration **class** B **: private** A rend tous les membres de A privés dans B.

Soit la classe Vecteur2D définie précédemment, on définit la classe Vecteur3D en la dérivant de la précédente :

```
class Vecteur3D : public Vecteur2D
{
private:
    double z;
public:
    void afficher(void);
    void acquerir(void);
};

void Vecteur3D::afficher(void)
{
    Vecteur2D::afficher();
    cout<<"  "<<z;
}

void Vecteur3D::acquerir(void)
{
    Vecteur2D::acquerir();
    cout<<"entrer la coordonnées en z : ";
    cin>>z;
}
```

Les champs abs et ord de la classe Vecteur2D peuvent être déclarés **protected** et non **private** pour permettre aux classes dérivées d'y avoir accès.

Dans les méthodes de la classe Vecteur3D, on fait appel aux méthodes de la classe de base en précisant le nom de la classe suivi de l'opérateur : :, par exemple Vecteur2D::afficher();.

3.3.2 Constructeur et destructeur dérivés

En général, le constructeur de la classe dérivée utilise le constructeur de la classe de base. Pour ce faire, une syntaxe particulière est employée : dans la définition du constructeur de la classe dérivée, on ajoute les arguments destinés au constructeur de la classe de base entre parenthèses et précédés de deux points.

Sur l'exemple précédent, le constructeur s'écrit par exemple :

```
class Vecteur3D : public Vecteur2D
{
    ...
    Vecteur3D(double abscisse=0,double ordonnee=0,double altitude=0);
    ...
};

Vecteur3D::Vecteur3D(double abscisse=0,double ordonnee=0,double altitude=0)
    : Vecteur2D(abscisse,ordonnee)
{
    z=altitude;
}
```

Les arguments (abscisse,ordonnee) seront pris par le constructeur de la classe de base Vecteur2D.

Le destructeur d'une classe dérivée fait automatiquement appel au destructeur de la classe de base.

3.3.3 Méthodes virtuelles

Une méthode déclarée **virtual** devra être redéfinie dans les classes dérivées. Cette technique sera surtout utilisée pour des besoins de polymorphisme lorsque la classe des objets n'est connue qu'au moment de l'exécution ; le mécanisme de déclenchement de la « bonne » méthode se fera alors dynamiquement et non plus statiquement.

Imaginons une utilisation des classes `Vecteur2D` et `Vecteur3D` précédentes où l'on déclarerait un pointeur sur un `Vecteur2D` que l'on instanciera dynamiquement, tantôt par un `Vecteur2D`, tantôt par un `Vecteur3D` (ce qui est possible puisqu'un `Vecteur3D` est dérivé d'un `Vecteur2D`).

```
int main()
{
    Vecteur2D *p;
    p = new Vecteur2D(1,1);
    p->afficher();              // on obtient   1  1
    p = new Vecteur3D(2,3,4);
    p->afficher();              // on obtient   2  3       !!!!
    ((Vecteur3D *)p)->ecrire(); // on obtient   2  3  4
}
```

La méthode invoquée est celle de la classe du pointeur p, soit celle de la classe `Vecteur2D`. Pour obtenir le résultat souhaité, il faut convertir le pointeur p en pointeur sur un objet `Vecteur3D`.

Pour obtenir cette liaison dynamique avec la classe de l'objet pointé et non du pointeur, il faut déclarer virtuelle la méthode de la classe de base par le mot clé **`virtual`** :

```
class Vecteur2D
{
    ...
    virtual void afficher(void);
    ...
};

int main()
{
    Vecteur2D *p;
    p = new Vecteur2D(1,1);
    p->afficher();              // on obtient   1  1
    p = new Vecteur3D(2,3,4);
    p->afficher();              // on obtient   2  3  4
}
```

Remarques :
- seul un constructeur ne peut être virtuel ;
- si une classe comporte des méthodes virtuelles, il est prudent de déclarer le destructeur lui aussi virtuel ;
- on peut déclarer une méthode virtuelle pure par l'ajout au prototype de `= 0` ; cela évite d'implanter la méthode dans la classe de base ; il ne pourra alors exister d'objet de cette classe qualifiée d'abstraite ! (seuls des pointeurs pourront être utilisés) ; par exemple :

```
class ClasseAbstraite
{
    ...
    virtual double methode_virtuelle_pure( void ) = 0;
    ...
};
```

3.3.4 Héritage multiple

Une classe peut dériver de plusieurs autres classes de base et hériter ainsi des membres de chacune.

```
class B : public A1, public A2, ... , public An
{
    // membres propres à la classe B
};
```

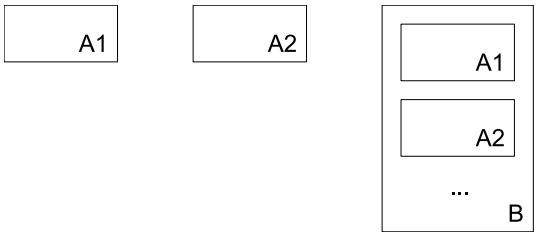
Remarques :
- une classe ne doit pas hériter directement plusieurs fois de la même classe. Tous les Ai doivent donc être distincts ;
- chaque classe de base peut être publique ou privée, par exemple : **`class`** `B` **`: public`** `A1,` **`private`** `A2 { ... };`
- si les classes de base `Ai` possèdent des méthodes de même nom, l'ambiguïté sera levée par l'opérateur de portée (`::`) ;
- les constructeurs des classes de base seront exécutés en premier, suivant l'ordre des déclarations des classes `A1,...,An` puis le constructeur de la classe dérivée sera exécuté.

Exemple :

```
class A1
{
protected:
    int x1,y1;
public:
    A1(int a,int b)
    {
        x1=a ;
        y1=b ;
    };
    virtual void ecrire()
    {
        cout<<x1<<y1<<endl;
    };
    ...
}

class A2
{
protected:
    int x2;
public:
    A2(int a)
    {
        x2=a;
    };
    virtual void ecrire()
    {
        cout<<x2<<endl;
    };
    ...
}

class B : A1, A2
{
private :
    int x,y;
public  :
    B(int a,int b,int c,int d) : A1(a,b), A2(c)
    {
        x = c ;
        y = d ;
    };
    void ecrire()
    {
        A1::ecrire();
        A2::ecrire();
        cout << x << y ;
    };
    ...
}
```



3.3.5 Classes virtuelles

On peut trouver des cas où une classe dérive indirectement plusieurs fois d'une classe de base.
Exemple :

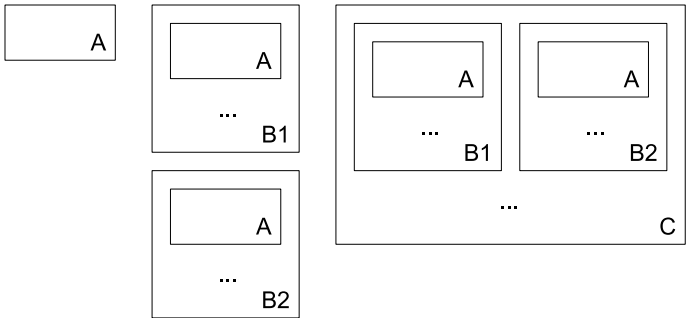
```
class A { ... };

class B1 : public A { ... };
```

```
class B2 : public A { ... };

class C  : public B1, public B2 { ... };
```

On se retrouve alors avec l'objet de la classe `C` contenant deux sous-objets de la classe `A`.



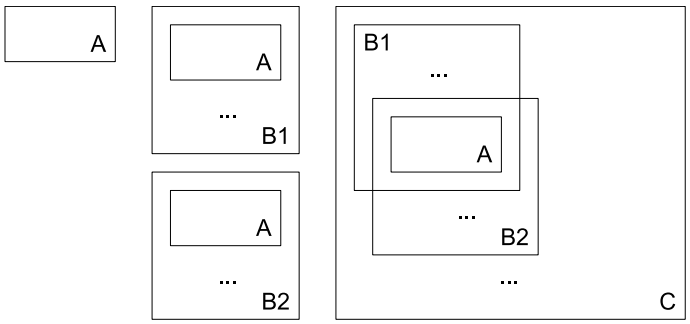
Si cet effet est indésirable, pour lever des ambiguïtés, pour résoudre les conflits et économiser la place mémoire, il suffit de déclarer les classes `B1` et `B2` comme virtuelles :

```
class A { ... };

class B1 : virtual public A { ... };
class B2 : virtual public A { ... };

class C  : public B1, public B2 { ... };
```

Un objet de la classe `C` possédera ainsi en un seul exemplaire les membres de la classe `A`.



3.4 Classes génériques

Il s'agit ici de classes paramétrées par un ou plusieurs types. La déclaration d'une telle classe est précédée du mot clé **`template`** suivi des paramètres entre chevrons.

```
template < class e1,class e2,...,class en > class nom_de_la_classe {...};
```

Les paramètres peuvent être un type de base (**`int`**, **`double`**, ...) ou un identificateur de classe.

```
template < class element > class Couple {...}; // couple de 2 éléments de même type

template < class element, int dim > class Tableau {...};  // tableau d'éléments de taille dim
```

On fait précéder l'implantation d'une méthode par **`template < class`** `e1,` **`class`** `e2,...,` **`class`** `en >` et, de plus, la méthode est préfixée par le nom de la classe rappelant le paramétrage.

```
template < class e1,class e2,...,class en >
    type_de_la_méthode nom_de_la_classe<e1,e2,...,en>::nom_de_la_méthode(arguments)
{
    ...
}
```

Par exemple, on veut se fabriquer une classe « Couple » comme étant un couple d'éléments de type quelconque, pour pouvoir ensuite utiliser des couples d'entiers, de réels, de complexes, d'individus, *etc.*

Déclaration de la classe :

```
template <class element> class Couple
{
private :
    element x,y;
public :
    void afficher(void);
    void acquerir(void);
};
```

Implantation des méthodes de la classe :

```
template <class element> void Couple<element>::afficher(void)
{
    cout<<"couple  ="<<x<<"  "<<y<<endl;
};

template <class element> void Couple<element>::acquerir(void)
{
    cout<<"entrer les éléments du couple ";
    cin>>x;
    cin>>y;
};
```

Utilisation :

```
main()
{
    Couple<int> a; // couple d'entiers
    a.acquerir();
    a.afficher();
    Couple<double> b; // couple de réels
    b.acquerir();
    b.afficher();
    Couple<Complexe> c; // couple de complexes; la classe complexe et les opérateurs
    c.acquerir();           // surchargés << et >> étant définis par ailleurs
    c.afficher();

    Couple<Complexe> t[10]; // t est un tableau de 10 couples de complexes
}
```

Remarque : dans le cas de classes paramétrées, la seule inclusion **#include** "nom_de_la_classe.h" dans le fichier utilisateur peut conduire à des erreurs d'édition de liens si l'implantation des méthodes est extérieure dans un fichier nom_de_la_classe.cpp. Certains compilateurs nécessitent **#include** "nom_de_la_classe.cpp" dans le fichier utilisateur.

3.5 Gestion des exceptions

La gestion des exceptions proposée par C++ est effectuée à l'aide des mots clés **throw**, **catch** et **try**.

Une exception peut être d'un type pré-défini (entier ou chaîne de caractères, par exemple) ou une instance d'une classe définie par le programmeur.

Dans un bloc repéré par **try**, une exception est soulevée à l'aide de **throw** et capturée par **catch**.

```
try
{
    // bloc d'instruction comportant des  throw  exception
    // ou des appels de fonctions comportant des throw  exception
}
catch (type_1 exception)
{
    // traitement de l'exception
}
...
catch (type_n exception)
{
    // traitement de l'exception
}
```

L'exception soulevée sera capturée par le premier catch comportant un type compatible.

Si aucun type n'est compatible, l'exception sera capturée par une fonction unexpected() qui, par défaut, fait appel à abort().

L'ellipse **catch**(...) capture tous les types d'exception.

Lors d'appels imbriqués de fonctions, une exception est d'abord traitée par la fonction la plus profonde. Si elle ne peut être capturée par cette fonction, elle est retournée à la fonction appelant.
Lorsque une exception est déclenchée, les objets créés dans le bloc **try** sont détruits.

On peut préciser les prototypes des fonctions en ajoutant avec **throw** les types d'exceptions qu'elles peuvent soulever :

```
type nom_de_fonction(paramètres) throw (types exceptions) ;
```

Exemple :

```
typedef  int  tableau[10];

void  fonction_exemple_exagere (tableau t, int i, double x) throw(int, string);
// cette fonction peut soulever des exceptions de type entier ou chaîne de caractères
// range la partie entière de x dans t[i]

main()
{
    tableau  a;
    int  k;
    double   r;
    try
    {
        cout<<"taper l'indice : ";
        cin>>k;
        cout<<"taper la valeur : ";
        cin>>r;
        fonction_exemple_exagere(a,k,r);
    }
    catch (string)
    {
        cout << "avertissement :  " << e ;
    }
    catch (int e)
    {
        switch (e)
        {
        case 0:
        {
            cout <<"débordement d'indice bas " << i ;
            break;
        }
        case 1:
        {
            cout <<"débordement d'indice haut "<< i ;
            break;
        }
        }
        exit(1);
    }
    catch (double  e)
    {
        cout<<"valeur trop grande : "<<r ;
        exit(2);
    }
    catch (...)
    {
        cout << "exception non cataloguée !" ;
        exit(3);
    }
    return 0;
}

void  fonction_exemple_exagere (tableau t, int i, double x)
{
    if (i<0)
        throw 0;
    if (i>9)
```

```
        throw 1;
    if (abs(x)>=32767)
    {
        throw x ;
    };
    t[i]=abs(x);
    if ((i==0)||(i==9))
    {
        throw "extrémité du tableau";
    };
    throw "tout est conforme ";
}
```

Chapitre 4

Bibliothèques standards du C++

Ce chapitre présente une sélection des possibilités offertes par la bibliothèque standard du C++.

4.1 Entrées/sorties standards

4.1.1 Entrées/sorties formatées

La bibliothèque `iostream` (incluse par **`#include`** `<iostream>`) définit de nouveaux classes et opérateurs d'entrées-sorties sur les types standards [1].

Les entrées-sorties sont supportées par les classes `istream` (flot d'entrée), `ostream` (flot de sortie) et `iostream` (flot d'entrée-sortie, classe dérivée des deux précédentes).

Les variables et constantes suivantes sont également définies :

- `cin` objet de type `istream` relié à l'entrée standard (le clavier par défaut) ;
- `cout` objet de type `ostream` relié à la sortie standard (l'écran par défaut) ;
- `endl` pour passer à la ligne suivante.

Les sorties se font par l'opérateur << :

```
ostream& operator<<(type x);
```

Par exemple :

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

int main()
{
    cout << "Hello world !" << endl;
    return 0;
}
```

ou :

```
#include <iostream>

int main()
{
    std::cout << "Hello world !" << std::endl;
    return 0;
}
```

ou encore :

```
#include <iostream>
using std::cout;
using std::endl;

int main()
```

1. Avant que le C++ ne soit normalisé, `<iostream.h>` était le seul fichier d'en-tête existant livré avec les compilateurs de l'époque. La normalisation ISO du C++ en 1998 a défini que `<iostream>` serait l'en-tête standard pour les entrées-sorties. L'absence de `.h` dans son nom indique qu'il s'agit désormais d'un en-tête standard, et donc que toutes ses définitions font partie de l'espace de nom standard `std`. Il est en de même avec tous les fichiers d'en-tête standards en C++.

```
{
    cout << "Hello world !" << endl;
    return 0;
}
```

Les entrées se font par l'opérateur >> :

```
istream& operator>>(type & x);
```

Par exemple :

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

int main()
{
    double x,y;
    cout<<"entrer x ";
    cin>>x;
    cout<<"entrer y ";
    cin>>y;

    cout<<" x = ";
    cout<<x;
    cout<<" y = ";
    cout<<y;
    cout<<endl;
    return 0;
}
```

On peut regrouper plusieurs expressions d'entrées-sorties en une seule :

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

int main()
{
    double x,y;
    cout<<"entrer x et y ";
    cin>>x>>y;

    cout<<" x = "<< x <<" y = "<< y <<endl;
    return 0;
}
```

Si l'on souhaite prendre en compte tous les caractères et ne pas éliminer les blancs, tabulations, retours à la ligne et caractères de fin de fichier, on peut utiliser les méthodes suivantes :

```
int       get ( void );
istream&  get ( char& c );
istream&  getline ( char * s , int  n , char delim = '\n' );
```

4.1.2 Manipulateurs

La bibliothèque <iomanip> fournit des outils de formatage appelés manipulateurs pour mettre en forme les données lors de l'affichage à l'écran :

- `setw(n)` fixe la largeur de la zone d'affichage à n caractères,
- `setprecision(n)` fixe le nombre de chiffres à afficher d'un nombre réel,
- `setfill('X')` remplit la zone libre par des caractères X,
- `hex` pour lire ou afficher le prochain nombre en hexadécimal,
- `oct` pour lire ou afficher le prochain nombre en octal,

Exemple :

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

void main( void )
{
  int n;
```

```
  double a;
  n=86;
  a=-13.141592;

  cout<<"Affichage normal  : "<<n<<endl;
  cout<<"Affichage formate : "<<setw(10)<<n<<endl;
  cout<<"Affichage normal  : "<<a<<endl;
  cout<<"Affichage formate : "<<setprecision(3)<<a<<endl;
  cout<<"Affichage normal  : "<<setw(10)<<"ENSMM"<<"Besancon"<<endl;
  cout<<"Affichage formate : "<<setw(10)<<setiosflags(ios::left)<<"ENSMM"
  cout<<"Besancon"<<endl;
}
```

Résultat produit à l'écran :

```
Affichage normal  : 86
Affichage formate :         86
Affichage normal  : -13.1416
Affichage formate : -13.1
Affichage normal  :      ENSMMBesancon
Affichage formate : ENSMM     Besancon
```

Ces manipulateurs font appels aux méthodes `fill()`, `width()`, `precision()` de la classe `ios`. On peut donc aussi écrire :

```
cout.fill('_');
cout.width(10);
cout.precision(3);
cout<<a<<endl;
```

Ce qui donne :

```
_____-13.1
```

4.2 Entrées/sorties sur les fichiers

La bibliothèque `<fstream>` définit des classes permettant de lire et d'écrire dans des fichiers formatés :
- `ifstream` classe des fichiers ouverts en lecture, classe dérivée de `istream`;
- `ofstream` classe des fichiers ouverts en écriture, classe dérivée de `ostream`;
- `fstream` classe des fichiers ouverts en lecture-écriture, classe dérivée de `iostream`.

Ces classes de flots fichiers héritent des méthodes de leurs classes mères auxquelles s'ajoutent quelques méthodes suivantes particulières.

L'ouverture d'un fichier s'effectue implicitement lors de sa déclaration par la méthode constructeur ou explicitement par la méthode `open()`.

```
/* constructeurs sans ouverture */
ifstream::ifstream(void);
ofstream::ofstream(void);
fstream::fstream(void);

/* constructeurs avec ouverture */
ifstream::ifstream(const char* nom,int mode=ios::in,int prot=filebuf::openprot);
ofstream::ofstream(const char* nom,int mode=ios::out,int prot=filebuf::openprot);
fstream::fstream(const char* nom,int mode,int prot filebuf::openprot);

/* ouverture explicite */
void ifstream::open(const char* nom,int mode=ios::in,int prot=filebuf::openprot);
void ofstream::open(const char* nom,int mode=ios::out,int prot=filebuf::openprot);
void fstream::open(const char* nom,int mode,int prot filebuf::openprot);
```

La valeur de protection par défaut `openprot` représente le droit de lecture-écriture pour le propriétaire du fichier et le droit de lecture pour tous.

La valeur du mode est à choisir parmi les valeurs : `ios::in`, `ios::out`, `ios::app`, `ios::trunc`, `ios::ate`, `ios::nocreate` ou `ios::noreplace`.

Le mode peut être une concaténation de plusieurs valeurs, par exemple : `ios::in | ios::out | ios::noreplace`

Les autres méthodes sont :

```
void      close(void);
streampos tellg(void); //  retourne la position courante dans le fichier
ifstream& seekg(streampos p); // déplacement absolu
ifstream& seekg(streampos dep,ios::seek_dir pos); // déplacement relatif
// le paramètre pos peut être ios::beg, ios::cur ou ios::end
```

Comme les classes des bibliothèques d'entrées-sorties, les classes de flots fichiers possèdent un ensemble de variables indicatrices de l'état d'un flot. Cet état peut être consulté en appelant les méthodes suivantes :

```
int eof(void);
int bad(void);
int fail(void);
int good(void);
int rdstate(void);
void clear(int i=0);
```

4.3 Les classes conteneurs de la STL

La bibliothèque générique standard (Standart Template Library) est une bibliothèque C++, normalisée par l'ISO (document ISO/CEI 14882). L'une des implantations les plus diffusées de la STL a été développée historiquement par Hewlett-Packard, puis par Silicon Graphics [2].

La STL fournit un ensembles de classes génériques, appelées conteneurs, permettant de générer les structures de données les plus répandues telles que les tableaux, les listes (chaînées), les tableaux associatifs, *etc.* Il existe deux catégories de conteneurs : les conteneurs séquentiels et les conteneurs associatifs.

4.3.1 Les conteneurs séquentiels

Les éléments d'un conteneur séquentiel sont stockés selon un ordre : le deuxième suit le premier, le troisième suit le deuxième, *etc.* On peut parcourir le conteneur selon cet ordre (du premier au dernier). Enfin, quand on insère ou quand on supprime un élément, on le fait à une place qu'on a explicitement choisie.

Les trois principaux conteneurs séquentiels sont : les tableaux ou collections statiques (classe `vector`), les listes chaînées (classe `list`) et les files (classe `deque`). Ces trois classes ont une interface proche mais se distinguent par des implantations différentes.

	`vector`	`deque`	`list`
Accès à un élément	O(1)	O(1)	O(N)
Insertion/suppression d'un élément en queue	O(1)	O(1)	O(1)
Insertion/suppression d'un élément en tête	O(N)	O(1)	O(1)
Insertion/suppression d'un élément au milieu	O(N)	O(N)	O(1)

Les bibliothèques à inclure portent le nom des conteneurs (`<vector>`, `<list>` et `<deque>`). Toutes les classes sont définies dans l'espace de nom `std`.

Méthodes communes

Les trois conteneurs `vector`, `deque` et `list` ont un certain nombre de méthodes communes (**`operator`** `=`, `size`, `empty`, `clear`, **`operator`**`==`, **`operator`**`<`, `back`, `push_pack`, `pop_back`, `begin`, `end`, `insert`, `erase`) qui sont détaillées dans les exemples suivants.

Construction

```
list<int>       L;         // La liste L est vide

list<int>       A(10);     // La liste A contient 10 entiers non initialisés
vector<double>  V(20);     // Le vecteur V contient 20 réels non initialisés

list<int>       B(10,0);   // La liste B contient 10 entiers initialisés à 0
deque<int>      D(20,-1);  // La file D contient 20 entiers initialisés à -1
```

2. Une documentation complète est disponible sur le site de Silicon Graphics : http ://www.sgi.com/tech/stl/

Dans les exemples suivants, `X` et `Y` désigneront arbitrairement des conteneurs `vector`, `deque` ou `list`.

Affectation et recopie

Les opérateurs d'affection et de recopie (passage par valeur) sont définis et utilisables directement.

```
X=Y; // copie effective des éléments de X dans Y
     // X et Y appartenant à la même classe
```

Nombre d'éléments (dimension)

```
int N=X.size();    // N reçoit la dimension de X
if (X.empty()) ... // Teste si X est vide
```

Vider un conteneur

```
X.clear(); // X.size() vaut maintenant 0
```

Comparaison entre conteneurs

Ces opérateurs supposent que des opérateurs correspondants (==, <, *etc.*) soient définis sur les éléments.

```
if (X==Y) ... // vrai si X et Y sont identiques (même éléments, même taille)
if (X<Y) ...  // comparaison lexicographique entre X et Y
```

Ajout/suppression en queue

Les trois conteneurs disposent de méthodes d'accès, d'ajout et de suppression du dernier élément en `O(1)`.

```
int x=X.back();  // x reçoit la valeur du dernier élément de X
X.push_back(45); // ajoute un élément de valeur 45 à la fin X
X.pop_back();    // supprime le dernier élément
```

Accès aux éléments, notion d'itérateur

Pour tous les conteneurs, l'accès aux éléments peut se faire par l'intermédiaire d'un itérateur. Un itérateur est un objet défini par la classe conteneur (d'où la déclaration `list<int>::iterator` par exemple) qui se comporte comme un pointeur sur les éléments du conteneur. Les deux exemples qui suivent sont valables également pour `vector` et `deque`.

```
list<int>::iterator il; // itérateur sur les éléments d'une liste d'entiers
il=L.begin();           // il pointe sur le premier élément de L
*il=35;                 // affecte la valeur 35 au premier élément de L
il++;                   // avance d'un élément
cout<<*il;              // affiche la valeur du deuxième élément de L

il=L.begin();           // il pointe sur le premier élément de L
while (il!=L.end())     // tant qu'il y a encore des éléments
{
    cout<<*il;          // affiche l'élément pointé par il
    il++;               // avance d'un élément
}
```

Insertion/suppression d'un élément

```
L.insert(il,5);   // insère un élément devant l'élément pointé par il
V.erase(il);      // supprime l'élément pointé par il
                  // après cet appel il n'est plus valide
il=V.erase(il);   // supprime l'élément pointé par il et il pointe sur
                 // l'élément suivant
```

Méthodes du conteneur `vector`

Le conteneur `vector` correspond au concept de tableau ou collection statique.

Accès direct aux éléments

Dans un conteneur `vector`, l'accès aux éléments (et leur modification) est direct avec l'opérateur `[]`. Les indices courent de `0` à `V.size()-1`.

```
for (int i=0 ;i<V.size();i++)
    V[i]=2*V[i];          \\ double toutes les valeurs
```

Ajout/suppression en queue

Comme `list` et `deque`, le conteneur `vector` dispose des méthodes d'accès, d'ajout et de suppression du dernier élément en `O(1)`. Si la dimension du conteneur diminue, les espaces mémoires en surplus sont conservés. Si la dimension du conteneur augmente et dépasse la taille maximum qu'il a atteint dans le passé, le conteneur procède à une réallocation en mémoire (opération très lente en `O(N)`).

Modification de la dimension

On peut changer à tout moment la dimension d'un conteneur `vector` (ce qui provoque une réallocation en mémoire).

```
vector<int> V;  // V est créé vide (dimension nulle)
V.resize(10);   // V est maintenant de taille 10
V.resize(20,4); // V contient maintenant 20 entiers initialisés à 4
```

Utilisation des méthodes communes nécessitant un itérateur

Pour obtenir de manière directe un itérateur sur un élément du conteneur, on utilise l'opérateur `&`.

```
vector<int> V(10,3); // V=<3,3,3,3,3,3,3,3,3,3>
V.insert(&V[3],7);   // V=<3,3,3,7,3,3,3,3,3,3>
V.erase(&V[2],7);    // V=<3,3,7,3,3,3,3,3,3>
```

Méthodes du conteneur `deque`

Le conteneur `deque` dispose des mêmes fonctionnalités que `vector` mais il permet en plus l'accès, l'ajout et la suppression en tête en `O(1)` avec les méthodes : `front`, `push_front` et `pop_front`. Naturellement, les autres opérations sur `deque` sont légèrement plus lentes que sur `vector`.

```
deque<int> D;
D.push_front(45); // ajoute un élément de valeur 45 en tête D
D.pop_front();    // supprime le premier élément
int x=D.front();    // x reçoit la valeur du premier élément de D
```

Méthodes du conteneur `list`

Le conteneur `list` est une liste doublement chaînée. L'accès direct n'est pas possible (avec l'opérateur `[]`), il faut obligatoirement utiliser un itérateur.

Accès/ajout/suppression d'élément

Le conteneur `list` dispose des méthodes d'accès, d'ajout et de suppression en tête et en queue : `front`, `back`, `push_front`, `pop_front`, `push_back`, `pop_back`.

Autres méthodes

```
L.sort();    // trie la liste L
L.merge(B);  // fusion de deux listes ordonnées
L.remove(0); // retire tous les éléments nuls de L
L.unique();  // L n'a plus de doublons !

L.remove_if(Pair); // retire tous les éléments pairs de L
// suppose qu'une fonction Pair est définie, par exemple :
// bool Pair(int n)
// { return(n%2); }
```

Algorithmes génériques

Les algorithmes génériques sont des sous-programmes qui manipulent un conteneur quelle que soit sa classe. Ils sont définis dans la bibliothèque <algorithm>. Ces algorithmes supposent que les opérateurs == et < sont définis sur les éléments. Les quelques exemples qui suivent sont valables pour vector, deque et list.

Algorithmes de recherche

```
list<int>::iterator il;
il=find(L.begin(),L.end(),10); // renvoie un itérateur sur le premier
                               // élément égal à 10 si il existe et
                               // l'itérateur L.end() sinon
il=find_if(L.begin(),L.end(),Pair()); // Trouve le premier élément pair
```

Algorithme de recherche dichotomique (pour un conteneur trié)

```
list<int>::iterator il;
il=binary_search(L.begin(),L.end(),10);
```

Algorithmes de recherche de maximum ou de minimum

```
list<int>::iterator il;
il=min_element(L.begin(),L.end());
il=max_element(L.begin(),L.end());
```

Algorithme de tri

```
sort(V.begin(),V.end());
```

4.3.2 Les conteneurs associatifs

Les conteneurs associatifs ont pour vocation de retrouver une information non plus en fonction de sa place dans le conteneur mais en fonction de sa valeur ou d'un partie de sa valeur appelée clé. Par exemple, un dictionnaire peut être modélisé par un conteneur associatif contenant des articles dont la clé d'accès serait définie par le mot correspondant à l'article.

Les deux conteneurs associatifs les plus importants sont map et multimap.

4.4 La classe string

La classe string de la bibliothèque <string> offre un cadre pratique pour manipuler les chaînes de caractères en C++. Les opérateurs usuels (affectation =, concaténation avec +, *etc.*) ainsi que les entrées-sorties standards (cout, cin) sont utilisables directement.

Le petit programme ci-dessous illustre la facilité d'utilisation de cette classe :

```
string A,B(", ca va ?");
A="bonjour";
A=A+" antoine";
A=A+B;
A[0]='B';
B=A;
cout<<A<<endl;
if (A==B) cout<<"A est identique à B"<<endl;
```

Le code suivant décrit les principales méthodes de cette classe sous la forme d'une définition de classe simplifiée (la définition réelle utilise la classe générique basic_string et est peu lisible.

```
class string
{
private:
  ...
public:
  /// construit moi-même avec une chaîne de caractère entre
  /// guillemets
  string(char * ch="");
```

```
  /// constructeur de recopie
  string(const string & s);

  /// libère la mémoire
  ~string();

  /// moi-même reçoit s
  void operator=(const string & s);

  /// moi-même reçoit une chaîne de caractère entre guillemets
  void operator=(char * ch);

  /// renvoie vrai si moi-même est avant s dans l'ordre
  /// alphabétique
  bool operator<(string s);

  /// renvoie vrai si moi-même et s sont identiques
  bool operator==(string s);

  /// renvoie moi-même suivi de la chaîne s (concaténation)
  string operator+(string s);

  /// renvoie le ième caractère de moi-même
  char & operator[](int i);

  /// renvoie la taille de moi-même
  int size(void);

  /// renvoie vrai si moi-même est vide
  bool empty(void);

  /// renvoie la position de la première occurence de s
  /// dans moi-même, si s n'est pas dans moi-même alors le
  /// résultat est <0 ou >size()
  int find(string s);

  /// insert s à la ième position dans moi-même
  void insert(int i,string s);

  /// supprime n caractères de moi-même à partir du ième caractère
  void erase(int i,int n);

  /// remplace n caractères de moi-même à partir du ième caractère
  /// par ceux de s
  void replace(int i,int n,string s);

  /// renvoie une sous-chaîne de moi-même de taille n à partir
  /// du ième caractère
  string substr(int i,int n);

  /// renvoie un pointeur sur un tableau de caractères contenant
  /// la chaîne stocké par moi-même
  char * c_str(void);
};
```

Annexe A

Priorité des opérateurs C et C++

Les opérateurs sont classés du plus prioritaire (niveau 17) au moins prioritaire (niveau 1).

Niveau	Opérateur	Descriptif
17	`::` `::`	opérateur de portée globale opérateur de portée de la classe

16	`->` `.`
	`[ ]`
	`( )`
	`sizeof`
		sélection d'un membre de structure
		indexation de tableau
		appel d'une fonction
		taille en octets

15	`++` `--`
	`~`
	`!`
	`+` `-`
	`*`
	`&`
	`( )`
	`new delete`
		auto incrémentation et décrémentation
		négation bit à bit
		négation logique
		plus et moins unaires
		accès à une variable pointée
		adresse d'une variable
		conversion de type
		opérateurs de gestion mémoire

14	`->*` `.*`	sélection d'un membre de structure
13	`*` `/` `%`	opérateurs multiplicatifs
12	`+` `-`	opérateurs additifs
11	`>>` `<<`	opérateurs de décalage
10	`<` `<=` `>` `>=`	opérateurs de relation
9	`==` `!=`	égalité et inégalité
8	`&`	et bit à bit
7	`^`	ou exclusif bit à bit
6	`|`	ou bit à bit
5	`&&`	et logique
4	`||`	ou logique
3	`? :`	affectation conditionnelle
2	`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`	opérateurs d'affectation et d'affectation composée
1	`,`	opérateur de séquence

Annexe B

Codes de caractères ASCII

Le tableau B.1 contient les valeurs décimales (et hexadécimales) du jeu de caractères ASCII (American Standards Committee for Information Interchange).

Ctrl	Déc	Hex	Car	Code	Déc	Hex	Car	Déc	Hex	Car	Déc	Hex	Car
^@	0	00		NUL	32	20		64	40	@	96	60	`
^A	1	01		SOH	33	21	!	65	41	A	97	61	a
^B	2	02		STX	34	22	"	66	42	B	98	62	b
^C	3	03		ETX	35	23	#	67	43	C	99	63	c
^D	4	04		EOT	36	24	$	68	44	D	100	64	d
^E	5	05		ENQ	37	25	%	69	45	E	101	65	e
^F	6	06		ACK	38	26	&	70	46	F	102	66	f
^G	7	07		BEL	39	27	'	71	47	G	103	67	g
^H	8	08		BS	40	28	(	72	48	H	104	68	h
^I	9	09		HT	41	29	)	73	49	I	105	69	i
^J	10	0A		LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B		VT	43	2B	+	75	4B	K	107	6B	k
^L	12	0C		FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D		CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E		SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F		SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10		DLE	48	30	0	80	50	P	112	70	p
^Q	17	11		DC1	49	31	1	81	51	Q	113	71	q
^R	18	12		DC2	50	32	2	82	52	R	114	72	r
^S	19	13		DC3	51	33	3	83	53	S	115	73	s
^T	20	14		DC4	52	34	4	84	54	T	116	74	t
^U	21	15		NAK	53	35	5	85	55	U	117	75	u
^V	22	16		SYN	54	36	6	86	56	V	118	76	v
^W	23	17		ETB	55	37	7	87	57	W	119	77	w
^X	24	18		CAN	56	38	8	88	58	X	120	78	x
^Y	25	19		EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A		SUB	58	3A	:	90	5A	Z	122	7A	z
^[	27	1B		ESC	59	3B	;	91	5B	[	123	7B	{
^\	28	1C		FS	60	3C	<	92	5C	\	124	7C	|
^]	29	1D		GS	61	3D	=	93	5D	]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^-	31	1F	▼	US	63	3F	?	95	5F	_	127	7F	⌂*

* Le code ASCII 127 correspond au code de suppression. Avec MS-DOS, ce code produit le même effet qu'ASCII 8 (BS). Le code de suppression peut être généré par la combinaison de touches CTRL + RET. ARR.

TABLE B.1 – Codes ASCII.

Le jeu de caractères étendus inclut 128 autres caractères dédiés aux graphiques et dessins (jeu souvent intitulé « jeu IBM de caractères »). Le jeu de caractères étendus varie selon la police sélectionnée. Le tableau B.2 contient le jeu de caractères étendu par défaut d'une application console sous Windows.

Déc	Hex	Car
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Déc	Hex	Car
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	░
177	B1	▒
178	B2	▓
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Déc	Hex	Car
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	▉
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Déc	Hex	Car
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	

TABLE B.2 – Codes ASCII étendus (console Windows).

Index