Faculté des Sciences de Luminy
Département d'Informatique

Le langage et la bibliothèque C++

Norme ISO

Henri Garreta
Octobre 1999

Table des matières

Chapitre 1. Avant de parler des objets

1.1 Introduction

Ce polycopié expose le langage C++ (norme ISO/IEC 14882, 1998), présenté comme l'extension orientée objets du langage C (norme ANSI X3.159, 1989), supposé connu.

La structure de ce document est la suivante :

- les chapitres 2 et 4 exposent les deux notions fondamentales de la programmation orientée objets : l'*encapsulation* (chapitre 2) et l'*héritage* (chapitre 4) ;
- le chapitre 1 est un ensemble hétéroclite de *concepts divers*, non nécessairement liés à la programmation orientée objets, certains très prématurés, qui doivent être expliqués ici car ils interviennent dans les chapitres ultérieurs ;
- le chapitre 3 traite de la *surcharge des opérateurs*, une importante particularité de C++ que n'ont pas les autres langages orientés objets ;
- au chapitre 5 sont présentés les *modèles* (*template*) de fonctions et de classes, un autre mécanisme original qui est l'incarnation en C++ de la notion de généricité,
- le chapitre 6 montre comment est réalisé en C++ le mécanisme de notification et d'interception des *exceptions*, une notion qu'on rencontre dans tous les langages sérieux,
- le chapitre 7 traite du dernier apport au langage C++, les *espaces de noms* qui permettent de gérer la masse importante de noms globaux présente dans les gros programmes.

Les deux chapitres restants traitent de la *bibliothèque standard* de C++, désormais officielle, englobant notamment la bibliothèque standard de modèles (STL) et la bibliothèque de flux d'entrée-sortie (iostream) :

- le chapitre 8, *bibliothèque standard*, présente les éléments non directement liés à la STL,
- le chapitre 9, *structures de données et algorithmes*, présente la STL : utilitaires généraux, conteneurs, itérateurs, algorithmes.

Tous les exemples de ce polycopié ont été essayés, avec des succès variés, sur trois compilateurs au moins :

- *egcs* version 1.1.2, le compilateur C/C++ distribué dans Red Hat Linux version 5.1 (fin 98),
- *Microsoft Visual C++* version 6.0 (début 99),
- *Borland C++ Builder* version 3.0 (fin 98).

Malgré ses extensions et tolérances bizarres, *egcs* est le compilateur le plus proche de la norme ; tous les exemples proposés s'y compilent à peu près comme prévu, encore que certaines constructions que la norme donne pour erronées n'y suscitent que de vagues avertissements. Il n'en est pas de même de *Visual C++* et de *C++ Builder* qui présentent de plus importantes lacunes, notamment à propos de certains chapitres comme les modèles, la bibliothèque STL et, dans une moindre mesure, les exceptions.

1.2 Mots réservés

Voici la liste des mots réservés de C++ qui ne sont pas des mots réservés de C, avec des renvois aux sections où ils sont expliqués :

bool § 1.9.1	**catch** § 6.1.1	**class** § 2.1
const_cast § 1.5	**delete** § 1.10.1	**dynamic_cast** § 1.5, § 4.5.1
explicit § 3.3.1	**false** § 1.9.1	**friend** § 2.9
inline § 1.6.4	**mutable** § 2.5.3	**namespace** § 7.2
new § 1.9.1	**operator** § 3.1	**private** § 2.1, § 4.1.2
protected § 4.1	**public** § 2.1, § 4.1.2	**reinterpret_cast** § 1.5
static_cast § 1.5	**template** § 5.1	**this** § 2.1.3
throw § 6.1.1	**true** § 1.9.1	**try** § 6.1.1
typeid § 4.5.2	**typename** § 5.2.2	**using** § 4.1.2, § 7.2.1
virtual § 4.3.4, § 4.6.2	**wchar_t** § 1.9.4	

1.3 Statut des constantes

En C, une déclaration comme

```
const int n = 100;
```

installe une variable en tout point ordinaire, sauf que le compilateur tentera d'en signaler les modifications qui lui seront apparentes. En C++, **n** ressemble moins à une variable qu'à une macro (pseudo-constante introduite par un **#define**), car :

- la portée de **n** est réduite au fichier où cette déclaration est écrite,
- les expressions où **n** figure sont évaluées durant la compilation,
- en particulier, **n** peut apparaître là où une constante est requise (par exemple, comme dimension dans une déclaration de tableau).

Il y a cependant des différences avec les macros : si cela est nécessaire, le compilateur allouera un emplacement dans la mémoire pour loger la constante. Par exemple, si l'adresse de cette dernière est requise :

```
const int n = 100;

const int *p = &n;      // n doit être une lvalue
```

Notions en passant qu'affecter à un pointeur l'adresse d'un objet constant ou initialiser une référence (cf. § 1.9.2) à partir d'un objet constant n'est légitime que s'il s'agit d'un pointeur ou d'une référence sur objet constant :

```
const int n = 100;
int *p1 = &n;           // ERREUR
const int *p2 = &n;     // Oui
int &r1 = n;            // ERREUR
const int &r2 = n;      // Oui
```

1.4 Statut et placement des définitions

1.4.1 Déclarations et définitions

Autant que possible nous distinguerons la *définition* d'une variable ou d'une fonction de sa *déclaration*. L'une et l'autre sont des expressions qui indiquent le type d'une variable, ou le type et la signature d'une fonction. Mais une définition produit un objet dans la mémoire (de l'espace, éventuellement garni d'une valeur initiale, pour une variable, le code compilé d'une fonction), alors qu'une déclaration ne fait qu'annoncer (« promettre ») qu'un tel objet sera créé plus loin, ou dans un autre fichier.

Exemple, déclarations :

```
extern int n;
double puissance(double x, int n);
```

Définitions :

```
int n = -1;

double puissance(double x, int n)
    {
    ...
    code de la fonction
    ...
    }
```

ATTENTION. Dans certains cas on dit simplement *déclaration* pour « *déclaration ou définition* ». En principe, le contexte permet de résoudre l'ambiguïté.

1.4.2 Statut des définitions

En C++, une définition de variable est considérée comme une variété d'instruction. Par conséquent :

- une définition de variable n'a pas à se trouver au début de son bloc, elle peut apparaître partout où peut figurer une instruction,
- les expressions figurant dans une définition de variable à titre de valeurs initiales n'ont pas à être constantes, elles peuvent faire intervenir des valeurs qui ne seront connues que durant l'exécution du programme,
- la définition d'une variable peut déclencher un traitement complexe, donné par l'expression d'initialisation de la variable ; c'est une opération qui se place dans le temps, parmi les autres actions du programme.

Il est donc pertinent de parler de « l'exécution d'une définition de variable ». La définition d'une variable locale est exécutée lorsque le contrôle atteint le point où elle est écrite. Les définitions des variables de même niveau d'imbrication, et en particulier les définitions des variables globales, sont exécutées dans l'ordre où elles sont écrites dans le programme.

Si une variable n'est utile que pour une boucle **`for`**, on peut la déclarer dans la première expression de la boucle ; sa portée est alors réduite au corps de la boucle :

```
int main()
    {
    int nl = 5;
    int nc = 6;

    for (int i = 0; i < nl; i++)
        {
        for (int i = 0; i < nc; i++)
            cout << i << ' ';
        for (int i = nc - 1; i >= 0; i--)
            cout << i << ' ';
        cout << '\n';
        }
    }
```

On peut également déclarer une variable dans la condition d'un **`if`** ou d'un **`while`**, ou dans le bloc d'un **`switch`**, mais cette possibilité semble beaucoup moins utile.

Les variables locales statiques sot initialisées la première fois que leur fonction est appelée. Le programme suivant affiche deux fois 1 :

```
int zarbi(int x)
    {
    static int x0 = x;
    return x0;
    }
```

```
void main()
    {
    cout << zarbi(1) << "\n";
    cout << zarbi(2) << "\n";
    }
```

1.5 Conversions de type

Cette section peut être ignorée en première lecture.

Plus utile qu'en C, mais aussi dangereuse, l'opération de changement de type adopte plusieurs formes :

```
type ( expression )
( type ) expression
```

Ces syntaxes correspondent au changement de type tel que le comprend le langage C. La première est élégante :

```
int(x)
```

mais la deuxième est nécessaire si *type* n'est pas défini par une formule réduite à un identificateur :

```
(char *) t
```

Suspendant l'émission de messages d'avertissement, cette opération rend possibles les pires horreurs :

```
void machin(const int *p)
    {
    ...
    *(int *)p = 0;          // *p était "garanti" constant!
    ...
    }
```

Découragée, car concise (c.-à-d. peu visible) et dangereuse, cette opération fragilise les programmes, comme dans l'exemple précédent. Pour cette raison C++ offre d'autres opérateurs de conversion, aux noms délibérément longs et voyants, censés mettre en évidence les agissements du programmeur qui les utilise.

Conversion de bas niveau

```
reinterpret_cast< type >( expression )
```

Effectue une conversion, intrinsèquement dangereuse et mal définie, entre deux types pointeurs ou entre un type pointeur et un type entier, quels que soient les types pointés et le type entier. Exemple :

```
class Animal { ... };
class Chien : public Animal { ... };
Animal a;
...
long l = &a;                            // ERREUR
long l = reinterpret_cast<long>(&a);    // Oui
...
```

On ne peut pas utiliser cet opérateur pour inhiber l'effet d'une qualification **const** ou **volatile**.

Modification du caractère constant ou volatile d'une expression

```
const_cast< type >( expression )
```

Ici, *type* et le type de expression doivent être les mêmes, aux qualifications **const** et **volatile** près. Cette conversion sert à ajouter ou enlever de telles qualifications. Exemple :

```
const int cent = 100;
...
int *ptnbr = &cent;                     // ERREUR
int *ptnbr = const_cast<int *>(&cent); // Oui
...
```


Conversion statique

```
static_cast< type >( expression )
```

Effectue une conversion, éventuellement dangereuse (elle est sous la responsabilité du programmeur), mais définie sur toute implémentation de C++. Sont autorisées, en plus des conversions standard, les conversions :

- d'un type entier vers un type énumération,

```
enum jsem { lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche };
enum jsem j = static_cast<jsem>(3);
```

- de *B* ***** vers *D* *****, où *B* est une classe de base accessible de *D* (la réciproque est une conversion standard).

Exemple :

```
class Animal { ... };

class Chien : public Animal { ... };

Animal a;
...
Chien *p = &a;                          // ERREUR

Chien *p = static_cast<Chien *>(&a);    // Oui
...
```

Cet opérateur n'utilise pas l'identification dynamique de type (cf. § 4.5), c'est au programmeur de s'assurer du bien fondé de la conversion. Notez que cet opérateur est appelé **`static_cast`** par opposition à **`dynamic_cast`** (voir ci-dessous) qui fat ce genre de conversions en toute sécurité.

Dans le cas d'une conversion entre pointeurs, la valeur 0 est convertie en 0 ; hormis ce cas, la conversion d'un pointeur en un autre type pointeur peut en changer la valeur (§ 4.3.2).

Cette opération ne permet pas une conversion entre types sans rapport entre eux, ni entre pointeurs vers des types entre lesquels il n'y a pas de lien d'héritage, comme **`float *`** et **`int *`**. Elle ne permet pas non plus d'inhiber l'effet d'une qualification **`const`** ou **`volatile`**.

Il vaut mieux éviter l'emploi de ces opérateurs. Parfois utiles pendant le développement et la mise au point des programmes, les opérateurs **`reinterpret_cast`** et **`const_cast`** ne devraient jamais figurer dans un programme de qualité industrielle. D'autre part, l'opérateur **`static_cast`**, aussi regrettable que son ancêtre **`(`***`type`***`)`**, reste la manière d'effectuer certaines opérations tout à fait avouables, comme :

```
int i, j;
float ratio = static_cast<float>(i) / j;   // division décimale
```

Conversion sûre

```
dynamic_cast< type >( expression )
```

Cet opérateur effectue une conversion sûre, qui s'appuie sur l'identification dynamique des types polymorphes (cf. § 4.5).

CONDITIONS. Il faut que *`type`* soit de la forme D_1 ***** et expression de la forme *B* *****, où D_1 et *B* sont des classes polymorphes, et où *B* est une classe de base accessible de D_1. Si ces conditions ne sont pas réunies il y aura une erreur à la compilation ou à l'exécution.

FONCTIONEMENT. Si le type dynamique de expression est D_2 *****, avec D_1 classe de base accessible de D_2, la conversion rend une adresse valide. Sinon, elle rend 0.

```
class Fruit                 // classe polymorphe (possédant
    { ... };                // des fonctions virtuelles)

class Pomme : public Fruit
    { ... };

class Golden : public Pomme
    { ... };
```

```
void main(void)
    {
    Fruit *adrPomme = new Pomme;
    Fruit *adrGolden = new Golden;

    Golden *ptr1 = dynamic_cast<Golden *>(adrGolden);  // succès
    Pomme  *ptr2 = dynamic_cast<Pomme  *>(adrGolden);  // succès
    Pomme  *ptr3 = dynamic_cast<Pomme  *>(adrPomme);   // succès
    Golden *ptr4 = dynamic_cast<Golden *>(adrPomme);   // ECHEC
    ...
    }
```

L'opérateur **dynamic_cast** s'applique aussi à des références. A cet effet, et pour des exemples d'utilisation de cet opérateur, voir § 4.5.1.

Pointeur générique

En C, tout pointeur peut être explicitement ou implicitement converti vers le type **void *** et réciproquement. En C++, seule la « généralisation » (conversion de *T* ***** vers **void ***) est permise ; de plus, elle ne doit pas servir à inhiber l'effet de la qualification **const** ou **volatile** :

```
int i, *pi;
void *pv;
const int ci = 0;

pv = &i;        // Oui
pi = pv;        // ERREUR
pv = &ci;       // ERREUR
```

1.6 Déclaration des fonctions

1.6.1 Déclaration préalable

Une fonction doit nécessairement avoir été déclarée avant chacun de ses appels.

Les déclarations « à l'ancienne », sans indication de la signature (c'est-à-dire de la suite des types des paramètres formels), ne sont pas admises. Par conséquent, le mot **void** n'est plus requis pour indiquer qu'une fonction est sans paramètre. L'expression

```
char lirecar();
```

déclare une fonction sans paramètre, non une fonction sans contrôle des paramètres comme ce serait le cas en C ANSI. Néanmoins, par souci de compatibilité, l'expression

```
char lirecar(void);
```

est tolérée (mais déconseillée).

1.6.2 Paramètres anonymes

Si un paramètre formel d'une fonction n'est pas utilisé, on peut omettre son nom. Cela évite les mises en garde de la part des compilateurs zélés. Exemple :

```
double ch(double x, double)     // cosinus hyperbolique
    {
    double w = exp(x);
    return (w + 1 / w) / 2;
    }
```

On peut se demander pourquoi une fonction aurait des paramètres formels inutiles. C'est son contexte d'utilisation qui impose parfois de tels paramètres. Par exemple, la fonction **ch** devra être écrite comme ci-dessus si elle doit être passée comme argument à une deuxième fonction qui a prévu de recevoir comme paramètre effectif une fonction à deux arguments :


```
double cumul(double t[], int n, double (*f)(double, double), double a)
    {
    double r = 0;
    for (int i = 0; i < n; i++)
        r += (*f)(t[i], a);
    return r;
    }
```

1.6.3 Valeurs par défaut des arguments

Les paramètres formels d'une fonction peuvent avoir des valeurs par défaut. Lors de l'appel de la fonction, les paramètres effectifs correspondants peuvent alors être omis, les paramètres formels seront initialisés avec les valeurs par défaut :

```
void trier(void *table, int nombre,
                   int taille = sizeof(void *), int croissant = TRUE);
```

Appels :

```
trier(t, n, sizeof(int), FALSE);
trier(t, n, sizeof(int));  // croissant = TRUE
trier(t, n);    // taille = sizeof(void *), croissant = TRUE
```

Pour qu'un paramètre puisse avoir une valeur par défaut il faut que ce soit le dernier de la liste, ou qu'il ne soit suivi que par des paramètres qui ont des valeurs par défaut.

REMARQUE. Les valeurs par défaut des paramètres sont utilisés lors de l'appel de la fonction ; la question concerne donc la déclaration de la fonction, non sa définition :

```
void ligne(FILE * = stdout);        // déclaration
```

ici peuvent se trouver des appels de **`ligne()`** *ou de* **`ligne(fic)`**

```
void ligne(FILE *f)                 // définition
    {
    putc('\n', f);
    }
```

1.6.4 Fonctions en ligne

Si une fonction est qualifiée **`inline`** ses appels ne sont pas traduits par la séquence d'appel du langage machine sous-jacent, mais remplacés, au niveau du texte source, par le corps de la fonction.

Cela est intéressant dans le cas de fonctions courtes, pour lesquelles le coût de la séquence d'appel n'est pas négligeable devant celui du corps de la fonction :

```
inline abs(int x)
    {
    return x >= 0 ? x : -x;
    }
```

Une fonction en ligne est semblable à une macro,

```
#define ABS(x) ((x) >= 0 ? (x) : -(x))
```

mais elle est en général préférable, car elle est intégrée au langage et respecte le système des types : les appels des fonctions en ligne subissent les mêmes contrôles sémantiques que les appels des fonctions normales.

De plus, les fonctions en ligne ne craignent pas les effets de bord :

```
while (i < n)
    s += abs(a[i++]);   // Correct
```

contrairement aux macros :

```
while (i < n)
    s += ABS(a[i++]);   // ERREUR
```

Cependant les macros restent irremplaçables dans certains cas. Par exemple, la fonction suivante affiche un numéro de ligne qui n'a aucun intérêt (le numéro de la ligne où la fonction est définie), alors qu'une macro équivalente afficherait le numéro de la ligne où la macro a été appelée :

```
inline erreur(char *message)
    {
    printf("erreur %s a la ligne %d", message, __LINE__);
    }
```

De même, le service rendu par la macro suivante, qui échappe au système des types, ne peut pas être obtenu avec une fonction en ligne :

```
#define ALLOC(nombre, type) malloc(nombre * sizeof(type))
```

Comme une macro, la portée d'une fonction en ligne est réduite au fichier où la fonction est définie. Par conséquent, les fonctions en ligne sont généralement écrites dans des fichiers en-tête, qui sont inclus dans tous les fichiers sources qui appellent ces fonctions.

REMARQUE. Le développement en ligne d'une fonction qualifiée **`inline`** n'est pas garanti. Le compilateur peut juger nécessaire de la traiter comme une fonction ordinaire, par exemple si elle est récursive ou trop complexe.

Enfin, une fonction en ligne peut être développée en ligne et être *de plus* compilée comme une fonction ordinaire. C'est ce qui arrive notamment lorsque l'adresse de la fonction est requise :

```
void map(int t[], int n, int (*f)(int))
    {
    for (int i = 0; i < n; i++)
        t[i] = (*f)(t[i]);
    }

inline int abs(int x)
    {
    return x >= 0 ? x : -x;
    }

...
u = abs(v);         // développement en ligne de abs
...
map(tab, n, abs);   // ceci provoque la compilation de abs
...
```

1.7 Surcharge des noms de fonctions

Des fonctions distinctes peuvent avoir le même nom, si leurs signatures sont assez différentes :

```
int puissance(int, int);
double puissance(double, int);
double puissance(double, double);
```

La signature d'une fonction est la suite des types de ses paramètres formels.

Le type du résultat rendu par la fonction n'en fait pas partie. Par conséquent, on ne peut pas déclarer ou définir deux fonctions dont les entêtes ne diffèrent que par le résultat qu'elles rendent.

De même, deux signatures qui ne se distinguent que par le fait que des paramètres correspondants ont les types *T* dans l'une et *T***`&`** ou **`const`** *T* dans l'autre, ou bien *T***`[]`** dans l'une et *T* **`*`** dans l'autre, ou encore deux types qui sont l'un un alias (crée par une déclaration **`typedef`**) de l'autre, ou tous les deux des alias d'un troisième, ne sont pas assez différentes.

1.7.1 Résolution de la surcharge

Cette section et les deux suivantes peuvent être sautées en première lecture.

L'opération consistant à choisir la fonction qui doit être effectivement activée lors d'un appel faisant apparaître un nom surchargé s'appelle la *résolution de la surcharge*. Le compilateur la fait en construisant un ensemble de fonctions candidates viables, dont il détermine la meilleure, c'est-à-dire celle dont les paramètres formels présentent la meilleure correspondance avec les paramètres effectifs.

La résolution échoue s'il n'existe aucune fonction viable, ou bien s'il en existe plusieurs et qu'aucune n'est meilleure que les autres.

ATTENTION. Ce n'est pas parce que la résolution aboutit que l'appel est forcément légal. Le compilateur détermine d'abord la fonction à appeler, ensuite il examine les autres conditions devant être satisfaites (la fonction doit être accessible, elle ne doit pas être virtuelle pure, etc.), faute de quoi il annonce une erreur.

1.7.2 Fonctions candidates

Les fonctions candidates sont les fonctions ayant le même nom que la fonction appelée, dont la déclaration est *visible* au point d'appel.

Un identificateur déclaré dans une région déclarative masque tous les objets ayant cet identificateur pour nom, visibles dans une région déclarative englobante :

```
void f(double);

void g()
    {
    void f(int);
    f(1.2);         // ensemble des candidates: { f(int) }
    }
```

Une fonction déclarée dans une classe masque les fonctions de même nom déclarées dans les classes de base :

```
void f(double);

class A
    {
public:
    void f(double);
    };

class B : public A
    {
public:
    void f(int);
    };

B b;
b.f(1.2);       // ensemble des candidates: { B::f(int) }
```

A partir du point d'appel, les régions déclaratives sont explorées de la plus englobée à la plus englobante ; la recherche s'arrête dès la rencontre d'une fonction candidate. Conséquence : toutes les fonctions candidates appartiennent à la même région déclarative.

N.B. 1. Dans le cas de l'utilisation d'un opérateur surchargé l'ensemble des fonctions candidates est la réunion de trois ensembles : celui des fonctions membres candidates (si le premier opérande est un objet), celui des fonctions non membres candidates et celui des opérateurs prédéfinis candidats.

N.B. 2. Pour la détermination des candidates, une fonction avec m paramètres dont n avec des valeurs par défaut compte pour n+1 fonctions, ayant respectivement m-n, m-n+1, ... m paramètres.

1.7.3 Fonctions viables

Ce sont les fonctions candidates qui peuvent être appelées avec les paramètres effectifs spécifiés dans l'appel. Une fonction candidate est viable :

- si elle a autant de paramètres formels (compte tenu des éventuelles valeurs par défaut) qu'il y a de paramètres effectifs dans l'appel,
- si, pour chaque paramètre effectif, il existe une séquence de conversions implicites qui peuvent le convertir dans le type du paramètre formel correspondant.

Durant cette phase, les fonctions membres sont examinées comme si elles avaient un premier paramètre supplémentaire :

```
class A
    {
    void f(int);
    void g(float) const;
    };

A a;
a.f(0);
```

Les fonctions f et g seront examinées comme si elles avaient été déclarées (mais cela ne veut pas dire qu'on doive ou qu'on puisse les écrire de cette manière) :

```
void f(A*, int);
void g(const A*, float);
```

de même, l'appel de **f** sera examiné comme s'il avait été écrit :

```
f(&a, 0)
```

Ce premier paramètre caché est traité comme les autres, sauf qu'on s'interdit de lui appliquer les conversions définies par l'utilisateur : si **b** est un objet d'une autre classe *B*, dans un appel comme

```
b.f(0);
```

la fonction *A*`::f(int)` n'est pas une candidate viable, même si une conversion de *B* vers *A* a été définie.

1.7.4 Meilleure fonction viable

La meilleure fonction viable est déterminée en considérant la « qualité » des conversions implicites qu'il faut appliquer aux paramètres effectifs pour les amener aux types des paramètres formels.

Une meilleure conversion est :

- aucune conversion, ou une conversion triviale (comme T`[]` $\leftrightarrow$ T `*`, $T \rightarrow$ `const` T, f`()`$\leftrightarrow$ `(*`f`)()`),
- sinon, une promotion de **`char`** ou **`short`** vers **`int`**, ou de **`float`** vers **`double`**,
- sinon, une conversion standard (**`int`** $\rightarrow$ **`float`**, etc.),
- sinon, une conversion définie par l'utilisateur.

Dans ces conditions, soit A_i l'ensemble des fonctions viables qui offrent la meilleure correspondance pour le $i^{\text{ème}}$ argument. L'appel est correct si et seulement si l'ensemble $A_1 \cap A_2 \cap \ldots \cap A_n$ a un élément et un seul. Cet élément est la meilleure fonction viable.

REMARQUES. La meilleure fonction viable doit être unique :

```
void f(int, double);
void f(double, int);

f(1, 1);        // appel ambigu
f(1.0, 1.0);    // appel ambigu
```

Même dégradante, une conversion standard est meilleure qu'une conversion définie par l'utilisateur :

```
class C
    {
public:
    operator int();
    };

void f(float, int);
void f(int, C);

C c;
f(0.99, c);     // meilleure : f(int, C)
```

Cas des instances de modèles

Le statut des instances des modèles face à la surcharge est confus. La stratégie proposée par le document de normalisation consiste à faire comme si on avait ajouté à l'ensemble des candidates toutes les fonctions produites en instanciant les modèles visibles au point d'appel, puis à considérer que, si les conversions à faire sont aussi bonnes, une fonction ordinaire est une meilleure candidate qu'une instance d'un modèle.

Actuellement (juillet 1999) cette ambition se heurte au manque de maturité du mécanisme d'instanciation des modèles. Exemple, tiré d'un compilateur réputé très proche de la norme (Borland C++ 5) :

```
void f(int, float);
template<class T> void f(T, T);
```

Premier essai :

```
main()
    {
    f(1, 2);        // fonction choisie: f(T, T) avec T = int
    f(1, 'A');      // fonction choisie: f(T, T) avec T = int
    }
```

Deuxième essai :

```
main()
    {
    f(1, 'A');      // fonction choisie: f(int, float)
    }
```

Cas des séquences de conversions implicites

Pour la détermination des fonctions candidates le compilateur prend en compte non seulement les conversions implicites simples des arguments, mais aussi les séquences de conversions implicites. De telles séquences sont formées d'au plus une conversion définie par l'utilisateur *U*, éventuellement précédée et/ou suivie de conversions standard *S*. Soient donc

$$C_1 = S_{1,1}U_1S_{1,2} \quad \text{et} \quad C_2 = S_{2,1}U_2S_{2,2}$$

deux de telle séquences. Dans ces conditions, C_1 est meilleure que C_2 si et seulement si $U_1 = U_2$ et $S_{1,2}$ est meilleure que $S_{2,2}$.

Cela veut dire notamment que deux séquences de conversions implicites comprenant des conversions définies par l'utilisateur ne sont comparables que si elles utilisent la même conversion définie par l'utilisateur.

1.8 Appel de fonctions écrites en C

Pour gérer la surcharge des fonctions, le compilateur C++ produit pour chaque fonction compilée un nom long comprenant le nom original de la fonction et une représentation codée de sa signature ; c'est ce nom long qui est communiqué à l'éditeur de liens.

Or les fonctions produites par un compilateur C n'ont pas de tels noms longs. Pour qu'elles puissent être appelées dans un programme C++ il faut encadrer leur déclaration par un énoncé de la forme :

```
extern "C" {
    déclarations de variables et fonctions obéissant aux règles du langage C
    }
```

Exemple :

```
extern "C" {
    double sqrt(double);
    }
```

D'autre part, dans tous les compilateurs C++, la macro **__cplusplus** est prédéfinie et vaut 1. Cela permet d'écrire des textes qui pourront être donnés à compiler aussi bien au compilateur de C qu'à celui de C++ :

```
#if __cplusplus
extern "C" {
#endif
```

déclarations de variables et fonctions C

```
#if __cplusplus
    }
#endif
```

1.9 Types nouveaux

1.9.1 Le type booléen

Le type **bool** est une alternative « propre » à la manière dont le langage C gère les conditions. Il comporte les deux valeurs **true** et **false** (ces deux mots sont réservés). Naturellement, le résultat d'un opérateur de comparaison ou d'un connecteur logique est de type **bool**. De même, les opérandes d'un connecteur logique, ou la condition d'un **if** ou d'un **while**, sont censés avoir le type **bool**.

La taille d'une valeur de ce type n'est pas précisée, mais il est garanti qu'une telle valeur peut être rangée dans un champ de bits quelle que soit sa largeur (manière de dire que l'information portée par un **bool** tient dans un bit).

Pour rester compatible avec le langage C, des conversions implicites sont définies du type **bool** vers les types numériques (**false** → 0, **true** → 1) et inversement (0 → **false**, toute valeur non nulle → **true**). Mais il est fortement déconseillé d'utiliser ces conversions implicites, qui sont appelées à devenir obsolètes.

Autrement dit, en C++ il ne faut pas remplacer « **if(x != 0)** … » par « **if(x)** … ».

1.9.2 Références

A côté des pointeurs, les références sont une autre manière de manipuler les adresses d'objets placés dans la mémoire (c'est-à-dire, des *lvalue*). Si *T* est un type, le type « référence sur *T* » se note *T***&**. Exemple :

```
int &r = i; // r est une référence sur i
```

Une valeur de type référence est une adresse mais, hormis lors de son initialisation, toute opération effectuée sur la référence agit sur l'objet référencé, non sur l'adresse. Il en découle qu'il est obligatoire d'initialiser une référence lors de sa création ; après, c'est trop tard :

```
r = j;          // ceci ne transforme pas r en une référence sur j
                // mais copie la valeur de j dans i
```

On peut dire qu'une référence est un pointeur auto-déréférencé ; par conséquent :

- une expression de type référence est une *lvalue*,
- les types **void &**, pointeur ou référence sur une référence et tableau de références ne sont pas valides.

Les références permettent de définir un synonyme d'un identificateur, ou un nom pour un objet qui n'en a pas :

```
int &elem = t[i];

elem = x;       // équivaut à: t[i] = x;
```

Les références permettent surtout de réaliser le passage par adresse des paramètres d'une fonction sans utiliser explicitement les pointeurs :

```
void permuter(int &a, int &b)
    {
    int w = a;
    a = b; b = w;
    }
```

Exemple d'appel de cette fonction (c'est le passage des paramètres qui produit l'initialisation, en tant que références, des arguments formels **a** et **b**) :

```
permuter(t[i], u);
```


Fonctions renvoyant des références

Une fonction peut renvoyer une référence comme résultat (ici, c'est l'objet temporaire « résultat de la fonction » qui sera initialisé en tant que référence) :

```
int &tab(int i)
    {
    return pages[i / N][i % N];
    }
```

Utilisations :

```
x = tab(i);
tab(j) = y;         // puisqu'une référence est une lvalue
```

ATTENTION. Lors de l'écriture d'une fonction qui renvoie une référence comme résultat, on doit faire attention à ne pas donner une référence sur un objet local, qui disparaîtra aussitôt après :

```
Matrice &produit(const Matrice &A, const Matrice &B)
    {
    Matrice res;

    calcul des coefficients de la matrice res = A × B

    return res;                    // ERREUR
    }
```

De même, il faut éviter de renvoyer des références « peu stables » :

```
class Pile
    {
    ...
public:
    int &sommet() { return tab[niv - 1]; }
    int depiler() { return tab[--niv]; }
    };

Pile p;
...
p.sommet() = p.depiler() + 1;      // Résultat imprévisible
```

Référence sur une *rvalue*

On peut définir une référence sur une expression qui n'est pas une *lvalue* (une constante, le résultat d'un calcul, etc.), à la condition qu'il s'agisse d'une référence sur un objet **const** :

```
int &r = 100;                      // ERREUR
double &pi = 4 * atan(1);          // ERREUR

const int &r = 100;                // Oui
const double &pi = 4 * atan(1);    // Oui
```

Pour initialiser une telle référence le compilateur crée une *lvalue* temporaire que la référence adresse. Les mêmes dispositions sont prises dans le cas de l'appel d'une fonction avec un paramètre de type référence :

```
void fon1(int &r);
void fon2(const int &r);
int x;

fon1(x);                       // Oui
fon1(2 * x + 3);               // ERREUR
fon2(x);                       // Oui
fon2(2 * x + 3);               // Oui
```

Notez que les types *T*& et **const** *T*& sont « assez différents » pour faire jouer la surcharge des fonctions :

```
void fon(int &r);
void fon(const int &r);

int x;
```

```
fon(x);                         // Oui, appel de fon(int &)
fon(2 * x + 3);                 // Oui, appel de fon(const int &)
```

1.9.3 Enumérations, structures, unions et classes

Le nom d'une énumération, d'une structure, d'une union ou d'une classe, est suffisant pour identifier le type correspondant (il n'y a pas besoin de répéter, comme en C, le mot-clé **enum**, **struct** ou **union**) :

```
enum Couleur { noir, rouge, jaune, vert, blanc };
Couleur c = blanc;
```

Tout se passe comme si à la suite de la déclaration de **enum Couleur** le compilateur avait ajouté la déclaration :

```
typedef enum Couleur Couleur;
```

Il en découle que, contrairement à ce qui se passe en C, les noms des énumérations, structures et unions définis localement masquent les noms définis dans des régions déclaratives englobantes.

1.9.4 Caractères étendus

Le type **wchar_t** représente un ensemble étendu de caractères, pouvant contenir de nombreux jeux de caractères locaux.

Notez qu'en C++ **wchar_t** est un mot réservé, alors qu'en C il n'est qu'un identificateur défini par une déclaration **typedef** (dans le fichier **<stddef.h>**).

1.10 Gestion dynamique de la mémoire

1.10.1 Allocation et restitution de mémoire

A côté des fonctions **malloc** et **free** de la bibliothèque standard de C, en C++ on dispose des opérateurs **new** et **delete**. Pour allouer un unique objet :

```
new type
```

Pour allouer un tableau de *n* objets :

```
new type[n]
```

Dans les deux cas, **new** renvoie une valeur de type « *type* ***** ». Exemples :

```
Machin *ptr = new Machin;

double *tab = new double[n];
```

Si *type* est une classe possédant un constructeur par défaut, celui-ci il sera appelé une fois (premier cas) ou *n* fois (deuxième cas) pour construire l'objet ou les objets alloués. Si *type* est une classe sans constructeur par défaut, une erreur sera signalée par le compilateur.

La valeur de *n* peut être inconnue à la compilation, mais la taille des objets alloués doit être connue. Il en découle que dans le cas des tableaux à plusieurs indices, seule la première dimension peut être non constante :

```
int (*M)[10];
...
acquisition de la valeur de n
...
M = new int[n][10];
```

(**M** pourra être utilisé comme une matrice à **n** lignes et 10 colonnes).

L'opérateur **delete** restitue la mémoire dynamique. Si la valeur de **p** a été obtenue par un appel de **new** on écrit

```
delete p;
```

dans le cas d'un objet qui n'est pas un tableau, et

```
delete [ ] p;
```


si c'est un tableau. Les conséquences d'une utilisation de **delete** là où il aurait fallu utiliser **delete[]**, ou inversement, sont imprévisibles.

A cause de la confusion, héritée de C, entre les types « tableau » et « adresse d'un élément », l'allocation dynamique des tableaux crée parfois des situations confuses :

```
struct Tiroir { ... };
typedef Tiroir Commode[8];

Tiroir  *t = new Tiroir;        // Oui
Commode *c = new Commode;       // ERREUR de type
Tiroir  *c = new Commode;       // Oui
```

On dit parfois « ce qui a été alloué par **new** doit être libéré par **delete** et ce qui a été alloué par **new[]** doit être libéré par **delete[]** ». On voit ici que ce n'est pas toujours aussi simple :

```
delete t;                       // Oui
delete c;                       // ERREUR (c'est un tableau)
delete [] c;                    // Oui
```

1.10.2 Surcharge de new et delete

Le programmeur peut définir son propre système d'allocation de mémoire en redéfinissant les opérateurs **new** et **delete**. Il lui suffit pour cela de définir des fonctions

```
void *operator new(size_t size);
void *operator new[](size_t size);

operator delete(void * ptr);
operator delete [] (void * ptr);
```

Par la suite, des appels comme

```
p = new type;
q = new type[n];
```

sont traduits par

```
p = operator new(sizeof(type));
q = operator new[](n * sizeof(type));
```

L'emploi des quatre versions prédéfinies des opérateurs **new** et **delete** est interdite dans un programme où au moins un de ces opérateurs a été redéfini.

1.10.3 Opérateurs de placement

Les opérateurs de placement prédéfinis « font semblant » d'allouer des objets à des adresses connues :

```
new(adresse) type
new(adresse) type[n]
```

Ces opérateurs se contentent de renvoyer adresse. Aucun **delete** ne doit être fait sur des objets ainsi obtenus.

Plus généralement on appelle opérateur de placement toute fonction **operator new** ou **operator new []** munie de plus de paramètres que l'obligatoire « **size_t** *size* » en première place. Ces opérateurs doivent être définis par le programmeur. Exemple

```
void *operator new [] (size_t size, void *buffer, int id_nbr);
```

Exemple d'appel :

```
int *tab = new(&espace, r++) type[n];
```

La définition de ses propres opérateurs d'allocation ou de placement suppose généralement qu'on renonce à utiliser la version standard de **delete**.

1.10.4 Echec de l'allocation de mémoire

L'échec de **new** produit le lancement d'une exception **bad_alloc**. Attraper cette exception est donc un moyen de savoir si l'allocation s'est mal passée et éventuellement y remédier.

Un autre moyen consiste à installer une fonction qui intercepte de tels échecs :

```
typedef void (*handler)();
handler set_new_handler(handler);
```

Utilisation :

```
void monManipulateur();

handler ancienManipulateur = set_new_handler(monManipulateur);

section de code où les allocations de mémoire sont protégées par monManipulateur

set_new_handler(ancienManipulateur);
```

Si la fonction d'interception se termine normalement, **new** tente à nouveau de faire l'allocation. Cela permet d'essayer diverses conduites avant de jeter l'éponge :

```
void monManipulateur()
    {
    static int numEssai = 0;
    switch(numEssai++)
        {
        case 0:
            compactage du tas
            break;
        case 1:
            expulsion de code inutile
            break;
        case 2:
            utilisation d'un fichier swap
            break;
        case 3:
            cerr << "Memoire insuffisante\n";
            terminate();
        }
    }
```

☺

Chapitre 2. Les classes

2.1 Classes et membres

La définition d'une classe est semblable à celle d'une structure en C :

```
class Point
    {
    typedef unsigned long Couleur;
public:
    void afficher()
        { cout << "(" << x << ", " << y << ") "; }
    Point();
    const int maxx, maxy;
private:
    int x, y;
    Couleur couleur;
    static int nbrPoints;
    };
```

avec des différences :

- le mot-clé **`class`** remplace souvent, mais non nécessairement, le mot-clé **`struct`** (première ligne de l'exemple ci-dessus),
- certains membres peuvent être des fonctions (**`afficher`** et **`Point`**, dans l'exemple ci-dessus),
- le droit d'accès à certains membres peut être restreint : on ne peut accéder à ces membres que depuis certains contextes (ci-dessus, les membres **`x`**, **`y`**, **`nbrPoints`** sont privés),
- certains membres peuvent exister en un unique exemplaire, étant alors partagés par tous les objets d'un même type classe (ci-dessus, **`nbrPoints`**),
- certains membres peuvent être astreints à rester constants (**`maxx`** et **`maxy`** ci-dessus),
- des types imbriqués peuvent être définis à l'intérieur d'une classe (le type **`Couleur`**, ci-dessus).

Dans une classe, les déclarations des membres peuvent se trouver dans un ordre quelconque, même lorsque ces membres se référencent mutuellement. Dans l'exemple précédent, le membre **`afficher`** mentionne les membres **`x`** et **`y`**, dont la définition se trouve après celle de **`afficher`**.

JARGON. Nous appellerons désormais

- *objet* une *lvalue* d'un type classe ou structure,
- *fonction membre* un membre d'une classe qui est une fonction,
- *donnée membre* ou *membre variable*, un membre qui n'est pas une fonction.

2.1.1 Membres publics et privés

Sauf dans quelques situations où un allégement est possible, l'accès aux membres des classes se fait selon la notation habituelle pour les champs des structures en C :

```
objet.membre
adresse->membre
```

Quelle que soit la notation employée, un membre public d'une classe peut être utilisé partout où il est visible ; un membre privé ne peut être utilisé que depuis une fonction membre de la classe (les notions de *membre protégé*, cf. § 4.1.1, et de classes et fonctions *amies*, cf. § 2.9, nuanceront cette affirmation).

Par défaut, les membres des classes sont privés. Les mots clés **`public`** et **`private`** permettent de modifier les droits d'accès des membres :

```
class nom
    {
    les membres déclarés ici sont privés
public:
    les membres déclarés ici sont publics
private:
    les membres déclarés ici sont privés

    etc.

    };
```

Les expressions **`public:`** et **`private:`** peuvent apparaître un nombre quelconque de fois dans une classe. Les membres déclarés après **`private:`** (resp. **`public:`**) sont privés (resp. publics) jusqu'à la fin de la classe, ou jusqu'à la rencontre d'une expression **`public:`** (resp. **`private:`**).

L'accès à un membre privé *m* d'un objet *x* est limité aux fonctions membres de la classe de *x*. Par exemple, avec les déclarations

```
class Point
    {
    ...
public:
    void afficher()
        { cout << '(' << x << ", " << y << ") "; }
    int distance(Point autrePt)
        { return abs(x - autrePt.x) + abs(y - autrePt.y); }
private:
    int x, y;
    ...
    };

Point p, q;
```

dans une fonction qui n'est pas membre ou amie de la classe **`Point`**, les expressions

```
p.x, p.y
```

pourtant syntaxiquement correctes et non ambiguës, constituent des accès illégaux aux membres privés **`x`** et **`y`** de la classe **`Point`**. Alors que l'expression :

```
p.afficher();
```

induit des accès parfaitement légaux aux membres **`p.x`** et **`p.y`**.

Encapsulation au niveau de la classe

On notera que tous les objets d'une classe ont le droit d'accéder aux membres privés de cette classe : deux objets de la même classe ne peuvent rien se cacher. Par exemple, dans l'expression

```
p.distance(q)
```

qui se développe en

```
abs(p.x - q.x) + abs(p.y - q.y)
```


l'objet **p** accède, grâce à la fonction membre **distance**, aux membres privés **x** et **y** de l'objet **q**. On dit que *C++ pratique l'encapsulation au niveau de la classe, non au niveau de l'objet.*

On notera au passage que, contrairement à d'autres langages, en C++, encapsuler n'est pas cacher mais interdire. Les utilisateurs d'une classe voient ses membres privés, mais ne peuvent pas les utiliser.

Structures

Une structure est la même chose qu'une classe mais, par défaut, les membres y sont publics. Sauf pour ce qui touche ce point, tout ce qui sera dit dans la suite à propos des classes s'appliquera donc aux structures.

```
struct nom
    {
    les membres déclarés ici sont publics
private:
    les membres déclarés ici sont privés
public:
    les membres déclarés ici sont publics

    etc.

    };
```

Classes incomplètes

Une classe peut intervenir dans sa propre définition, soit que certains membres variables sont des pointeurs vers la classe, soit comme type de paramètres ou du résultat de certaines fonctions membres :

```
class Point
    {
    ...
    Point *autrePoint;
public:
    int mieuxPlace(Point a, Point b);
    Point symetrique();
    };
```

D'autre part, une classe déclarée et non définie (on appelle cela un *type incomplet*) peut être utilisée dans toute expression ne requérant pas la connaissance de ses membres ou de sa taille :

```
class Point;

Point *pt;              // Oui
extern Point a;         // Oui
pt = &a;                // Oui

Point y;                // ERREUR
pt = new Point;         // ERREUR
pt->afficher();         // ERREUR
```

2.1.2 Fonctions membres

Les fonctions membres d'une classe ont le droit d'accéder à tous les membres, publics et privés, de la classe. Elles peuvent notamment référencer des membres déclarés plus loin dans la classe :

```
class Point
    {
public:
    int X() { return x; }
    int Y() { return y; }
    float distance(Point);

private:
    int x, y;
    };
```

Les fonctions membres d'une classe peuvent être définies à l'intérieur de la classe, comme **X** et **Y** dans l'exemple précédent. Il est alors convenu qu'il s'agit de fonctions en ligne (cf. § 1.6.4). Conséquence : ces fonctions doivent être courtes, en texte et en temps d'exécution.

Si on ne souhaite pas qu'elles soient en ligne, les fonctions membres doivent être uniquement déclarées dans la classe, comme **distance** dans l'exemple précédent. Il faut alors ultérieurement, ou dans un autre fichier, définir ces fonctions ; pour cela on utilise l'opérateur de résolution de portée :

```
classe::identificateur
```

Exemple (une fonction contenant un appel de **sqrt**, fonction relativement « lourde » de la bibliothèque mathématique, ne mérite pas d'être en ligne) :

```
float Point::distance(Point autrePt)
    {
    int dx = x - autrePt.x;
    int dy = y - autrePt.y;
    return sqrt(dx * dx + dy * dy);
    }
```

Accès simplifié à ses propres membres

Qu'elle soit définie dans la classe ou bien au dehors, dans une fonction membre les membres de l'objet à travers lequel la fonction a été appelée sont référencés à l'aide d'une notation simplifiée. Les identificateurs **x** et **y** qui apparaissent seuls dans la fonction précédente renvoient aux valeurs des membres **x** et **y** de l'objet à travers lequel on a appelé la fonction **distance**. Ainsi, si **p** et **q** sont des objets de la classe **Point**, un appel tel que

```
p.distance(q)
```

initialisera **dx** et **dy** comme si on avait écrit

```
dx = p.x - q.y;
dy = p.y - q.y;
```

2.1.3 Le pointeur this

Dans une fonction membre d'une classe *C* on peut utiliser l'identificateur **this**, comme nom d'une variable locale de type « *C* *** const** » (pointeur constant sur un *C*) et pour valeur l'adresse de l'objet à travers lequel la fonction a été appelée. Par exemple, les fonctions **X** et **Y** auraient pu être écrites de manière équivalente (mais cela n'a aucun avantage) :

```
class Point
    {
    ...
public:
    int X() { return this->x; }
    int Y() { return this->y; }
    ...
    };
```

Le pointeur **this** est indispensable lorsqu'un objet doit faire référence à lui même, et non pas à un de ses membres. Par exemple, si on avait disposé d'une fonction **distance** non membre :

```
float distance(Point, Point);       // distance entre deux Points
```

on aurait pu définir notre fonction membre **distance** par :

```
float Point::distance(Point autrePt)
    {
    return ::distance(*this, autrePt);
    }
```

Noter dans la fonction précédente comment l'opérateur de résolution de portée est utilisé pour indiquer que la fonction **distance** appelée est celle de la région déclarative globale :

```
::identificateur
```

Cela est nécessaire car une autre fonction nommée **distance** est définie dans la classe (la classe détermine une région déclarative englobée) qui masque la précédente. Au point d'appel de la fonction **distance**, les deux fonctions dont il est question dans cet exemple ne constituent pas un cas de surcharge de nom de fonction : dans la portée de la classe **Point**, la fonction membre masque la fonction non membre, et une seule fonction est candidate (cf. § 1.7.2).

2.2 Constructeurs

A propos de la construction des objets, voir aussi § 4.2.

Un constructeur d'une classe est une fonction membre spéciale :

- il a le même nom que la classe,
- il n'indique pas de type de retour,
- il ne contient pas d'instruction **`return`**.

Exemple :

```
class Point
    {
public:
    Point(int a, int b)
        { x = a; y = b; }
    ...
private:
    int x, y;
    };
```

Le constructeur initialise l'objet, notamment en donnant des valeurs aux membres pour lesquels cela est utile. Le constructeur n'a pas à s'occuper de trouver l'espace pour l'objet, il est appelé immédiatement après que cet espace ait été obtenu (quelle que soit la sorte d'allocation qui a été faite : dynamique, statique ou automatique, cela ne regarde pas le constructeur).

Une classe peut posséder plusieurs constructeurs, qui doivent alors avoir des signatures différentes (c'est un cas fréquent de surcharge d'un nom de fonction) :

```
class Point
    {
public:
    Point(int a, int b)
        { x = a; y = b; }
    Point(int a)
        { x = y = a; }
    Point()
        { x = y = 0; }
    ...
private:
    int x, y;
    };
```

L'utilisation de paramètres avec des valeurs par défaut peut servir à mettre en facteur certains constructeurs :

```
class Point
    {
    ...
    Point(int a = 0)        // ceci définit Point(int) et Point()
        { x = y = a; }
    ...
    };
```

Un constructeur est toujours appelé lorsqu'un objet est crée, soit explicitement, soit implicitement. Les appels explicites peuvent être écrits sous deux formes :

```
Point a(3, 4);

Point b = Point(5, 6);
```

Malgré les apparences, la deuxième expression ci-dessus exprime une initialisation, non une affectation (en effet, cette expression commence par un type, c'est donc une déclaration ou une définition, ici une définition, et il y a donc création d'un objet nouveau).

Dans le cas d'un constructeur avec un seul paramètre, on peut en outre utiliser une forme qui rappelle l'initialisation des variables de types prédéfinis (cela découle de la possibilité de définir des conversions propres à l'utilisateur, cf. § 3.3.1) :

```
Point e = 7;            // équivaut à: Point e = Point(7)
```

Un objet alloué dynamiquement est également initialisé, au mois implicitement. Dans beaucoup de cas il peut, ou doit, être initialisé explicitement. Cela s'écrit :

```
Point *pt = new Point(1, 2);
```

Les constructeurs peuvent aussi être utilisés pour initialiser des objets temporaires, anonymes. En fait, chaque fois qu'un constructeur est appelé avec succès, un objet est crée, même si cela ne se passe pas à l'occasion de la définition d'une variable. Par exemple, deux objets sans nom (représentant les points [0,0] et [3,4]) sont créés dans l'expression suivante :

```
cout << Point(0, 0).distance(Point(3, 4)) << "\n";
```

Toutes les variables d'un type classe sont initialisées. Les constructeurs des variables globales sont appelés dans l'ordre de déclaration de ces variables, avant l'appel de la fonction principale du programme (généralement main).

REMARQUE. Un objet est considéré construit uniquement lorsque son constructeur a été complètement exécuté. Cette affirmation semble une trivialité, mais devient une question importante au moment de programmer le traitement des exceptions qui pourraient se produire pendant la construction de l'objet.

2.2.1 Constructeur par défaut

Le constructeur par défaut est un constructeur qui peut être appelé sans paramètre (soit il n'a pas de paramètre, soit tous ses paramètres ont des valeurs par défaut). Il joue un rôle remarquable, car il est appelé chaque fois qu'un objet est créé sans qu'il y ait appel explicite d'un constructeur, soit que le programmeur ne le juge pas utile, soit qu'il n'en a pas la possibilité :

```
Point x;                    // équivaut à: Point x = Point();
Point t[10];                // produit 10 appels de Point();
Point *p = new Point[10];   // produit 10 appels de Point();
```

Tout objet doit être initialisé lors de sa création. Si le programmeur n'a écrit aucun constructeur pour une classe *C*, alors le compilateur synthétise un constructeur par défaut pour *C*. Si *C* n'a ni objet membre, ni fonction virtuelle, ni classe de base (c'est le cas de notre classe **`Point`**), alors le constructeur par défaut synthétisé par le compilateur est le constructeur trivial, qui consiste à ne rien faire. Si *C* a des objets membres, le constructeur synthétisé provoque l'appel du constructeur par défaut de chaque objet membre.

Si au moins un constructeur est défini pour une classe, alors aucun constructeur par défaut n'est synthétisé. Par conséquent, ou bien l'un des constructeurs explicitement définis est un constructeur par défaut, ou bien toute création d'un objet devra mentionner des valeurs d'initialisation.

2.2.2 Constructeur par copie

Le constructeur par copie d'une classe *C* est un constructeur dont le premier paramètre est de type *C***&** ou **`const`** *C***&** et dont les autres paramètres, s'ils existent, ont des valeurs par défaut.

Ce constructeur est appelé lorsqu'un objet est créé par clonage d'un objet existant. Cela arrive souvent, notamment chaque fois qu'un objet est passé comme paramètre par valeur à une fonction ou rendu comme résultat d'une fonction.

```
class Point
    {
public:
    Point(Point &p)
        { x = p.x; y = p.y; }       // copie triviale
    ...
    };
```



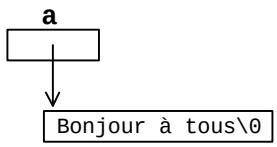
```
Point p, q;
...
r = distance(p, q);                // p et q sont copiés ici
```

Si le programmeur n'a pas défini de constructeur de copie pour une classe *C*, le compilateur synthétise un constructeur par copie qui consiste à recopier chaque membre d'un objet dans le membre correspondant de l'autre. Si aucun membre n'est un objet (et si *C* n'a pas de classe de base) cela revient à faire une copie bit à bit d'un objet sur l'autre, cela s'appelle un constructeur par copie *trivial*. C'est le cas du constructeur **`Point(Point&)`** montré plus haut, qui est inutile : le programmeur a écrit exactement ce qu'il aurait obtenu en ne rien écrivant.

Si la classe contient des membres qui sont des objets, alors le constructeur par copie synthétisé n'est plus trivial, car les copies des membres objets feront appel aux constructeurs par copie de leurs classes respectives.

Objets avec des bouts dehors

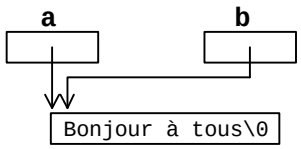
Typiquement, on écrit un constructeur par copie pour une classe lorsque celle-ci possède des membres de type pointeur. L'information représentée par un objet (l'« objet logique ») ne se trouve alors pas entièrement incluse dans l'espace contigu que le compilateur connaît comme étant l'objet (l'« objet technique »), car des morceaux d'information se trouvent à d'autres endroits de la mémoire.



Lors du clonage d'un tel objet, seul le programmeur peut décider de l'intérêt qu'il y a à cloner à leur tour ces informations pointées depuis l'objet, le compilateur ne peut pas le faire à sa place. Exemple :

```
class Chaine
    {
    char *caracteres;
public:
    Chaine(char *);
    ...
    };

Chaine::Chaine(char *s)
    {
    caracteres = new char[strlen(s) + 1];
    strcpy(caracteres, s);
    }

Chaine a("Bonjour a tous"), b = a;
```

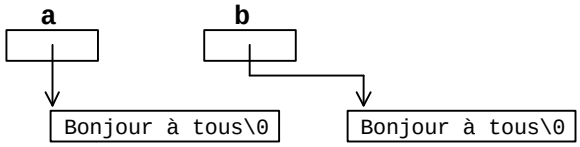
En l'absence de constructeur par copie, le clonage d'un objet **`Chaine`** consiste en la copie bit à bit de l'« objet technique », dont le membre pointeur. Cela donne une copie *superficielle* :




```
class Chaine
    {
    ...
public:
    Chaine(Chaine &);
    ...
    };
```

```
Chaine::Chaine(Chaine &c)
    {
    caracteres = new char[strlen(c.caracteres) + 1];
    strcpy(caracteres, c.caracteres);
    }
```

Avec un constructeur par copie adéquat, le clonage devient une « copie profonde » :



2.3 Destructeurs

A propos de la destruction des objets, voir aussi la section § 4.2

Un destructeur est une fonction membre spéciale. Il a le même nom que la classe, précédé du caractère ~. Il n'a pas de paramètre, ni de type de retour. Il y a donc au plus un constructeur par classe.

Le destructeur d'une classe est appelé automatiquement lorsqu'un objet de la classe est détruit, juste avant que la place occupée par l'objet soit rendue.

```
class Chaine
    {
    ...
public:
    ~Chaine();
    ...
    };
Chaine::~Chaine()
    {
    delete caracteres;
    }
```

Notez que le destructeur n'a pas à s'occuper de restituer l'espace occupé par l'objet. Cette restitution dépend de la mémoire que l'objet occupe (statique, automatique ou dynamique) et ne regarde pas le destructeur. Le travail de ce dernier est généralement la restitution des bouts de mémoire pointés depuis l'objet qui va disparaître ; sans cela, ces portions de mémoire risqueraient de devenir irrécupérables.

Si le programmeur n'a écrit aucun destructeur, le compilateur en synthétise un. Si la classe n'a ni objets membres ni classes de base, alors il s'agit du destructeur trivial qui consiste à ne rien faire. Si la classe a des objets membres, alors le destructeur synthétisé consiste à appeler les destructeurs des membres, dans l'ordre inverse de leur création.

2.4 Construction des objets membres

Lorsque des membres d'une classe sont à leur tour d'un type classe (on dit que la classe a des objets membres) alors la construction d'un objet entraîne la construction de ces membres.

Cela est *toujours* le cas, que le programmeur ait défini ou non des constructeurs explicites, et que ces constructeurs explicitent ou non l'initialisation des objets membres.

Les constructions des objets membres ont lieu

- dans l'ordre de déclaration des membres objets,
- avant l'exécution du corps de l'éventuel constructeur explicite.

Une syntaxe spéciale permet de spécifier des paramètres pour ces constructeurs :

```
classe(paramètres)
        : membre(paramètres), ... membre(paramètres)
    {
    corps du constructeur
    }
```

Exemple :

```
class Segment
    {
    Point origine, extremite;
    int epaisseur;
public:
    Segment(int, int, int, int, int);
    };

Segment::Segment(int x1, int y1, int x2, int y2, int ep)
        : origine(x1, y1), extremite(x2, y2)
    {
    epaisseur = ep;
    }
```

L'obligation d'initialiser explicitement des membres existe également lorsque la classe possède des membres d'un type référence, ou bien des membres constants (cf. § 2.5.1).

REMARQUE. On peut noter que cette syntaxe spéciale pour initialiser les objets membres est indispensable, sans elle on n'y arriverait pas. Deux expériences ratées :

```
Segment::Segment(int x1, int y1, int x2, int y2, int ep)
    {
    origine = Point(x1, y1);           // VERSION ERRONEE
    extremite = Point(x2, y2);         // VERSION ERRONEE
    epaisseur = ep;
    }
```

Le constructeur d'un **Segment** commence ici par l'affectation (non l'initialisation) des deux extrémités, préalablement initialisées par défaut. Par conséquent, cette version n'est correcte que si la classe **Point** a un constructeur par défaut, et elle est en tout cas maladroite, car **origine** et **extrémité** sont initialisés pour rien, deux points temporaires sont construits, affectés aux premiers, puis détruits.

```
Segment::Segment(int x1, int y1, int x2, int y2, int ep)
    {
    Point origine(x1, y1);             // VERSION ERRONEE
    Point extremite(x2, y2);           // VERSION ERRONEE
    epaisseur = ep;
    }
```

Cette version est tout à fait erronée, car elle définit et initialise deux points, nommés **origine** et **extremite**, qui sont des *variables locales* du constructeur, dans lequel elles masquent les membres **origine** et **extremite**, rendant impossible l'initialisation de ces derniers.

2.5 Membres constants

2.5.1 Données membres constantes

Une donnée membre d'une classe peut être qualifiée **const**. Il est alors obligatoire de l'initialiser lors de la construction d'un objet, et sa valeur ne pourra plus être modifiée.

```
class Segment
    {
    Point origine, extremite;
    int epaisseur;
    const int numeroDeSerie;
public:
    Segment(int, int, int, int, int, int);
    };
```

Constructeur, version erronée :

```
Segment::Segment(int x1, int y1, int x2, int y2, int ep, int num)
        : origine(x1, y1), extremite(x2, y2)
    {
    epaisseur = ep;
    numeroDeSerie = num;    // ERREUR (modification d'une constante)
    }
```

Constructeur, version correcte, en utilisant une syntaxe voisine de celle de l'initialisation des objets membres :

```
Segment::Segment(int x1, int y1, int x2, int y2, int ep, int num)
        : origine(x1, y1), extremite(x2, y2), numeroDeSerie(num)
    {
    epaisseur = ep;
    }
```

2.5.2 Fonctions membres constantes

Le mot **const**, placé à la fin du prototype d'une fonction membre, indique que l'état de l'objet à travers lequel on appelle cette fonction n'est pas altéré du fait de l'appel. C'est la manière de dire qu'il s'agit d'une fonction de *consultation* de l'objet, non d'une fonction de *modification*.

Si *C* est la classe en question, à l'intérieur d'une fonction **const** le pointeur **this** est de type **const** *C* *** const** (pointeur constant vers un *C constant*) : l'objet pointé par **this** ne pourra pas être modifié. Essentiellement, cela permet au compilateur d'autoriser certains accès qui, sans cela, auraient été interdits :

```
const Point p(2, 3);

int a = p.X();          // ERREUR: ces fonctions modifient
int b = p.Y();      // peut-être l'objet constant p
```

Version correcte :

```
class Point
    {
public:
    int X() const { return x; }
    int Y() const { return y; }
    ...
    };

const Point p(2, 3);

int a = p.X();          // Oui
int b = p.Y();
```

Les fonctions membres constantes et non constantes peuvent être appelées sur un objet quelconque, mais seules les fonctions membres constantes peuvent être appelées sur un objet constant.

La qualification **const** d'une fonction membre fait partie de sa signature. Ainsi, on peut surcharger une fonction membre non constante par une fonction membre constante ayant, à part cela, le même prototype. La fonction non constante sera appelée sur les objets non constants, la fonction constante sur les objets constants.

```
class Point
    {
public:
    int X() const { return x; }
    int Y() const { return y; }
    int &X() { return x; }
    int &Y() { return y; }
    ...
    };
```

Avec la déclaration précédente, les fonctions **X** et **Y** sont devenues « intelligentes » : sur un objet constant elles ne permettent que la consultation, sur un objet non constant elles permettent la consultation et la modification :

```
Point a(2, 3);
const Point b(4,5);

r = a.X();      // Oui : 'Point::X()' est appelée, rendant int&
a.X() = 0;      // Oui

r = b.X();      // Oui : 'Point::X() const' est appelée, rendant int
b.X() = 0;      // ERREUR
```

2.5.3 Membres mutables

Un membre qualifié **mutable** est une donnée membre qui peut être modifiée même si elle appartient à un objet qualifié **const** ou, ce qui revient au même, s'il est atteint à travers une fonction membre qualifiée **const**.

Un tel statut convient à un membre qui contribue à l'état interne de l'objet, non à la vue qu'en ont les utilisateurs. Vu de l'extérieur, l'objet reste constant même lorsque ce membre change. Exemple : un membre qui compte le nombre de fois qu'un objet a été *consulté* :

```
class Point
    {
public:
    int X() const { nbrConsult++; return x; }
    int Y() const { nbrConsult++; return y; }
    ...
private:
    int x, y;
    mutable int nbrConsult;
    };

const Point p(x, y);

a = p.X();      // ceci incrémente p.nbrConsult
```

2.6 Membres statiques

Alors que chaque objet d'une classe possède sa propre instance de chaque membre ordinaire (ce qui a surtout du sens dans le cas des données membres), les *membres statiques*, signalés par la qualification **static**, sont partagés par tous les objets de la classe.

De chacun il existe un seul exemplaire par classe, quelque soit le nombre des objets. La visibilité et les droits d'accès sur ces membres sont régis par les mêmes règles que les autres membres.

2.6.1 Données membres statiques

```
class Point
    {
public:
    Point(int a, int b)
        { x = a; y = b; nombreDePoints++; dernierPoint = this; }
    ...
    static int nombreDePoints;
```

```
private:
    static Point *dernierPoint;
    int x, y;
    };
```

Chaque objet **Point** possède ses propres exemplaires des membres **x** et **y** mais, quel que soit le nombre de points existants à un moment donné, il existe un seul exemplaire des membres **nombreDePoints** et **dernierPoint**.

Les déclarations de **nombreDePoints** et **dernierPoint** dans la classe sont de simples déclarations ; il faut ensuite créer et initialiser ces variables (pour les membres ordinaires ce travail a lieu lors de la construction des objets). Cela se fait par des déclarations qu'il faut écrire au niveau global, même lorsqu'il s'agit de membres privés :

```
int Point::nombreDePoints = 0;
Point *Point::dernierPoint = Point(0, 0);
```

De l'extérieur, les membres statiques peuvent être accédés à travers un objet :

```
Point p;
...
cout << p.nombreDePoints << "\n";
```

ou bien, puisqu'il y a un seul exemplaire de ces membres, indépendamment de tout objet :

```
cout << Point::nombreDePoints << "\n";
```

2.6.2 Constantes membres statiques

Une donnée membre statique peut être déclarée **const**. Sa portée sera réduite à la classe, elle sera publique ou privée comme les autres membres mais, à part cela, elle pourra s'utiliser comme une constante ordinaire. Par exemple, elle pourra intervenir comme dimension d'un tableau :

```
class Vecteur
    {
    static const int dim = 3;
    double tab[dim];                    // Oui
public:
    ...
    };

const int *ptr = &Vecteur::dim;         // ERREUR: Vecteur::dim
                                        // n'est pas une lvalue
```

Si la constante doit pouvoir être considérée comme une *lvalue* (par exemple : pour en prendre l'adresse) alors il faut *de plus* la définir au niveau global :

```
const int Vecteur::dim;

const int *ptr = &Vecteur::dim;         // Maintenant, oui
```

2.6.3 Fonctions membres statiques

Une fonction membre statique n'est pas attachée à un objet. Autrement dit :

- elle ne dispose pas du pointeur **this**,
- de la classe, elle ne peut référencer que les fonctions et les membres statiques.

```
class Point
    {
    ...
    static nombreDePoints;
    static void montrerDernier()
        { cout << '(' << dernierPoint->X() << ','
                                  << dernierPoint->Y() << ')'; }
private:
    static Point *dernierPoint;
    int x, y;
    };
```

Exemple d'appel :

```
Point a;
...
a.montrerDernier();
```

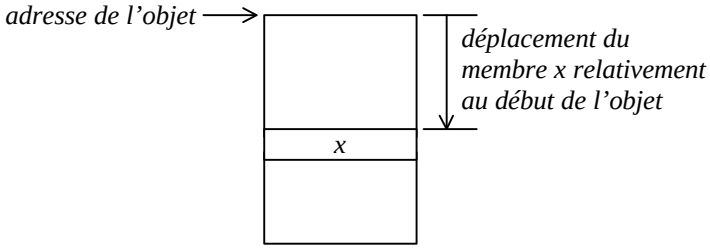
ou bien

```
Point::montrerDernier();
```

La qualification **static** ne fait pas partie de la signature d'une fonction : une fonction membre statique ne peut pas avoir le même nom qu'une fonction membre ordinaire ayant le même prototype.

2.7 Pointeurs vers membres

Un pointeur vers un membre d'une classe représente l'adresse d'un membre d'un objet, exprimée comme un *déplacement* (ou *décalage*) *relatif à l'adresse de l'objet* :



C étant une classe et *T* un type, le type « *pointeur vers un membre de C de type T* » se note « *T C*`::*` ». Exemple :

```
int Point::*ptCoordonnee;
```

L'adresse d'un membre relative au début d'un objet ne dépend pas de l'objet, mais seulement du membre. La classe suffit donc pour obtenir une telle valeur. On utilise pour cela l'opérateur spécial **&C::***membre* :

```
if (horizontalement)
    ptCoordonnee = &Point::x;
else
    ptCoordonnee = &Point::y;
```

L'expression précédente n'est légitime qu'en un contexte où l'on a le droit d'accéder aux membres **x** et **y**. De plus, un pointeur vers membre ne peut pas pointer sur un membre statique, puisqu'un tel membre n'est pas attaché à un objet.

L'accès au membre indiqué par la valeur d'un pointeur vers membre requiert un objet ; il se fait à l'aide de la notation « **.*** », si l'objet n'est pas pointé, ou « **->*** » s'il l'est :

```
Point p(2, 3), *q;
q = new Point(4, 5);
...
p.*ptCoordonnee += delta;

q->*ptCoordonnee += delta;
```

On peut également définir des pointeurs vers des fonctions membres. Exemple :

```
class ObjetGraphique
    {
    ...
public:
    void afficher();
    void eteindre();
    void arriere_plan();
    void avant_plan();
    ...
    };

void pour_chaque(ObjetGraphique *t, int n, void (ObjetGraphique::*f)())
    {
    for (int i = 0; i < n; i++)
        (t[i].*f)();
    }

const int max = 10;
ObjetGraphique figure[max];
int nbr;

void main()
    {
    ...
    pour_chaque(figure, nbr, &ObjetGraphique::eteindre);
    ...
    pour_chaque(figure, nbr, &ObjetGraphique::afficher);
    ...
    }
```

2.8 Types imbriqués

Un type peut être défini à l'intérieur d'une classe. Cela a deux conséquences principales :

- il ne peut être utilisé en dehors de la classe qu'en association avec l'opérateur de résolution de portée (ce qui réduit les conflits de noms),
- il subit les mêmes contrôles d'accès (public, privé…) que les membres de la classe.

```
class Point
    {
    enum Etat { allume, eteint };
public:
    enum Couleur { rouge, bleu, jaune, vert, noir };
    ...
    };

Couleur c1;                         // ERREUR
Point::Couleur c2 = rouge;          // ERREUR
Point::Couleur c3 = Point::rouge;   // Oui
Point::Etat e;                      // ERREUR (c'est un type privé)
```

Un type imbriqué dans une classe peut être utilisé dans le corps d'une fonction membre, même lorsque la déclaration de la fonction est placée avant celle du type. Pinaillons :

```
class Point
    {
    void test1() { Etat c; }        // Oui
    Etat test2() { }                // ERREUR

    enum Etat { allume, eteint };
    Etat test3() { }                // Maintenant, oui
    ...
    };
```

CLASSES IMBRIQUEES. Cas particulier de type défini par l'utilisateur, une classe peut être définie à l'intérieur d'une autre classe. *Cela ne donne aucun droit d'accès particulier d'une classe sur l'autre*. Le seul privilège de la classe imbriquée est de pouvoir mentionner les types et les membres statiques de la classe sans devoir utiliser l'opérateur de résolution de portée.

L'intérêt des types imbriqués est de limiter les conflits entre noms globaux, soit en rendant obligatoire l'utilisation de l'opérateur de résolution de portée :

```
class Segment
    {
public:
    class Point
        {
        int x, y;
        };
private:
    Point *a, *b;
    };

Point p, q;                 // ERREUR
Segment::Point p, q;        // Oui
```

soit en interdisant l'utilisation de la classe imbriquée :

```
class Segment
    {
    class Point
        {
        int x, y;
        };
    Point *a, *b;
    };

Segment::Point p, q;        // ERREUR (Segment::Point est privée)
```

La classe englobée peut être seulement déclarée dans la classe englobante. Il faut alors utiliser l'opérateur de résolution de portée pour la définir ailleurs :

```
class Segment
    {
public:
    class Point;
private:
    Point *a, *b;
    };

class Segment::Point
    {
    int x, y;
    };
```

2.9 Amis

2.9.1 Fonctions amies

Une fonction *amie* d'une classe *C* est une fonction qui, sans être membre de cette classe, a le droit d'accéder à tous ses membres.

Une telle fonction doit se déclarer ou se définir dans la classe qui lui accorde un tel droit d'accès, indifféremment en partie publique ou privée, précédée du mot réservé **`friend`** :

```
class Tableau
    {
    int *tab, nbr;
    friend ostream &operator<<(ostream&, const Tableau&);
public:
    ...
    };

ostream &operator<<(ostream &o, const Tableau &t)
    {
    o << '[';
    for (int i = 0; i < t.nbr; i++)
        o << ' ' << t.tab[i];
    o << " ]";
    return o;
    }
```

Une fonction amie *définie* à l'intérieur d'une classe est en ligne ; cela convient donc aux fonctions courtes et critiques :

```
class Tableau
    {
    ...
public:
    friend int taille(const Tableau &t)
        { return t.nbr * sizeof t.tab[0]; }
    ...
    };
```

emploi (**`tab`** étant de type **`Tableau`**) :

```
taille(tab)
```

ATTENTION. Notez que la qualification **`friend`** fait que, bien que définie à l'intérieur de classe, la fonction **`taille`** n'est pas membre de la classe **`Tableau`** ; en particulier, elle n'est pas attachée à un objet (le pointeur **`this`** n'est pas défini).

Le plus souvent, une fonction amie définie à l'intérieur de la classe peut être remplacée avantageusement par une fonction membre :

```
class Tableau
    {
    ...
public:
    int taille()
        { return nbr * sizeof tab[0]; }
    ...
    };
```

emploi (**`tab`** étant de type **`Tableau`**) :

```
tab.taille()
```

Une fonction membre d'une classe peut être amie d'une autre classe :

```
class Tableau;

class Fenetre
    {
public:
    void afficher(const Tableau &t);
    };

class Tableau
    {
    int *tab, nombre;
    friend void Fenetre::afficher(const Tableau&);
    ...
    };
```

2.9.2 Classes amies

Une classe amie d'une classe *C* est une classe qui a le droit d'accéder à tous les membres de *C*. Une telle classe doit être déclarée ou définie dans la classe *C*, indifféremment en partie privée ou publique :

```
class Maillon
    {
    int info;
    Maillon *suivant;
    Maillon(int i, Maillon *s)          // un seul constructeur, privé
        { info = i; suivant = s; }

    friend class Queue;
    friend ostream &operator<<(ostream&, const Queue&);
    };

class Queue
    {
    Maillon *premier, *dernier;
public:
    Queue()
        { premier = 0; }
    bool vide()
        { return premier == 0; }
    void entree(int i);
    int sortie();

    friend ostream &operator<<(ostream&, const Queue&);
    };

void Queue::entree(int i)
    {
    Maillon *p = new Maillon(i, 0);
    if (premier != 0)
        dernier->suivant = p;
    else
        premier = p;
    dernier = p;
    }

int Queue::sortie()
    {
    if (premier == 0)
        throw "Queue vide";
```

```
    Maillon *p = premier;
    premier = premier->suivant;
    int r = p->info;
    delete p;
    return r;
    }

ostream &operator<<(ostream &o, const Queue &q)
    {
    o << '[';
    for (Maillon *p = q.premier; p != 0; p = p->suivant)
        o << ' ' << p->info;
    o << " ]";
    return o;
    }
```

La relation d'amitié n'est pas transitive (« les amis de mes amis *ne sont pas* mes amis »).

A l'évidence, la notion d'amitié est une entorse aux règles qui régissent les droits d'accès ; elle doit être employée avec une grande modération, et uniquement pour assouplir les relations entre des éléments indissociables d'une même entité logicielle (les classes **`Maillon`** et **`Queue`** sont deux parties inséparables du concept « file d'attente réalisée par une liste chaînée »).

2.10 Autres remarques sur les types

2.10.1 Tableaux

La notion de tableau offerte par C++ est celle de C : une collection de *lvalue*s contiguës et de même type. Si les éléments du tableau sont des objets d'une classe *C*, alors ils seront initialisés lors de la construction du tableau ; cela oblige la classe *C* à posséder un constructeur par défaut.

Exception : dans le cas d'un tableau statique ou automatique (c.-à-d. non dynamique, ni membre d'une classe) on peut donner des valeurs initiales comme en C :

```
Point tab[N] = { Point(2, 3), Point(4,5), Point(6,7) };
```

Si **`N`** est supérieur à 3, le constructeur par défaut de la classe **`Point`** sera appelé pour initialiser les **`N`** – 3 derniers points.

2.10.2 Unions

Les unions peuvent avoir des fonctions membres privées et publiques, y compris des constructeurs et un destructeur. Mais elles ne doivent pas avoir d'objet membre possédant un constructeur [resp. un destructeur, un constructeur par copie, un opérateur d'affectation] non trivial, ni de membre statique.

Unions anonymes.

Une union peut être anonyme, elle ne définit pas alors un type, mais un objet sans nom. Les noms de ses membres doivent être différents des autres noms définis dans la région déclarative courante ; ces membres pourront être accédés directement :

```
static union {
        unsigned long l;
        char c[4];
        };

l = -1;
c[2] = 0;
cout << hex << l;          // Afiche : FF00FFFF
```

Une union anonyme ne peut pas avoir des membres privés, ni des fonctions membres. Une union anonyme globale doit être qualifiée **`static`** (cela la rend inaccessible aux autres modules).

Si elle est utilisée pour déclarer un objet ou un pointeur, une union n'est plus anonyme :

```
static union {
        unsigned long l;
        char c[4];
        } x;

l = -1;             // ERREUR ('l' inconnu)
x.l = -1;           // Oui
```

2.10.3 Classes agrégats

Une classe agrégat est une « structure au sens du langage C », c'est-à-dire une classe n'ayant que des membres publics, des constructeurs triviaux et aucune classe de base. Les objets d'une telle classe peuvent être initialisés comme les structures de C :

```
struct Point
    {
    int x, y;
    };

enum Couleur { rouge, vert, bleu, jaune, noir };

struct Segment
    {
    Point o, e;
    Couleur c;
    };

Segment s = { { 1, 2 }, { 3, 4 }, rouge };
```

☺☹

Chapitre 3. Les opérateurs et les conversions

3.1 Surcharge des opérateurs

3.1.1 Principe

C++ permet de définir ou modifier la sémantique de l'application des opérateurs du langage à des objets, soit pour étendre aux objets des opérateurs qui n'étaient initialement définis que sur les types standard, soit pour changer la sémantique prédéfinie d'opérateurs pouvant s'appliquer aux objets.

Cela s'appelle *surcharger* un opérateur. Tous les opérateurs de C++ peuvent être surchargés, sauf :

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

Surcharger un opérateur revient à définir une fonction ; tout ce qui a déjà été dit à propos de la surcharge des fonctions s'appliquera donc à la surcharge des opérateurs.

Il n'est pas possible d'inventer de nouveaux opérateurs ; seuls les opérateurs déjà connus du compilateur peuvent être surchargés.

Il n'est pas possible de modifier la sémantique d'un opérateur sur des valeurs de types prédéfinis ; un opérateur surchargé doit avoir au moins un opérande d'un type classe. Une fois surchargés, les opérateurs gardent leur pluralité, leur priorité et leur associativité. En revanche, ils perdent leur éventuelle commutativité et leurs éventuels liens sémantiques avec d'autres opérateurs. Par exemple, les surcharges de ++ ou <= n'auront aucun lien prédéterminé avec celles de + ou <.

Pour surcharger un opérateur @ il faut définir une fonction nommée **`operator`**@. Elle doit avoir au moins un paramètre d'un type classe. Elle peut être une fonction membre d'une classe ou bien une fonction indépendante, membre d'aucune classe.

3.1.2 Surcharge par une fonction membre

Si la fonction **`operator`**@ est membre d'une classe, elle doit comporter un paramètre de moins que la pluralité de l'opérateur : le premier opérande est l'objet à travers lequel la fonction est appelée . Sauf quelques exceptions,

obj_1 @ obj_2 équivaut à obj_1**`.operator`**@(obj_2)

Exemple :

```
class Point
    {
public:
    Point(int = 0, int = 0);
    int X() const { return x; }
    int Y() const { return y; }    ...
    Point operator+(Point);
    ...
private:
    int x, y;
    };

Point Point::operator+(Point q)
    {
    return Point(x + q.x, y + q.y);
    }

Point p, q, r;
...
r = p + q;              // compris comme: r = p.operator+(q);
```

3.1.3 Surcharge par une fonction non membre

Si la fonction **operator**@ n'est pas membre d'une classe, alors elle doit comporter un nombre de paramètres égal à la pluralité de l'opérateur. Ainsi :

obj_1 @ obj_2 équivaut à **operator**@(obj_1, obj_2)

Exemple :

```
Point operator+(Point p, Point q)
    {
    return Point(p.X() + q.X(), p.Y() + q.Y());
    }

Point p, q, r;
...
r = p + q;              // compris comme: r = operator+(p, q);
```

A cause des conversions implicites, la surcharge d'un opérateur binaire *symétrique* par une fonction non membre est en général préférable, car les deux opérandes y sont traités symétriquement. Exemple :

```
Point p, q, r;
int x, y;
```

Surcharge par une fonction membre :

```
r = p + q;              // Oui: r = p.operator+(q);
r = p + y;              // Oui: r = p.operator+(Point(y));
r = x + q;              // ERREUR (x n'est pas un objet)
```

Surcharge par une fonction non membre :

```
r = p + q;              // Oui: r = operator+(p, q);
r = p + y;              // Oui: r = operator+(p, Point(y));
r = x + q;              // Oui: r = operator+(Point(p), q);
```

La surcharge d'un opérateur binaire par une fonction non membre est obligatoire lorsque le premier opérande est d'un type standard ou d'un type classe prédéfini (c.-à-d. une classe que le programmeur ne peut plus étendre).

Exemple canonique : l'opérateur **<<** utilisé pour injecteur des données dans un flux de sortie. D'une part, l'auteur de la classe **ostream** a défini des surcharges, par des fonctions membres, pour les types connus de lui, c'est-à-dire les types standard :


```
class ostream
    {
    ...
public:
    ...
    ostream &operator<<(int);
    ostream &operator<<(unsigned int);
    ostream &operator<<(long);
    ostream &operator<<(unsigned long);
    ostream &operator<<(double);
    ostream &operator<<(long double);
    etc.
    ...
    };
```

D'autre part, l'utilisateur qui souhaite étendre aux objets d'une classe qu'il vient de définir ne peut plus ajouter des membres à la classe **`ostream`**. Il doit donc passer par une fonction non membre :

```
ostream &operator<<(ostream &o, const Point p)
    {
    return o << '(' << p.X() << ',' << p.Y() << ')';
    }
```

Parfois (ce n'est pas le cas ici) l'écriture de l'opérateur non membre sera plus simple si on en fait une fonction amie des classes de ses opérandes objets :

```
class Point
    {
    ...
    friend ostream &operator<<(ostream&, const Point);
    };

ostream &operator<<(ostream &o, const Point p)
    {                                              // possibilité d'accès
    return o << '(' << p.x << ',' << p.y << ')';   // aux membres privés
    }
```

En contrepartie, la surcharge d'un opérateur binaire par une fonction membre est préférable lorsqu'on ne souhaite pas que les deux opérandes jouent des rôles symétriques, comme dans le cas de l'affectation (voir § 3.2.1).

3.2 Quelques opérateurs remarquables

3.2.1 Affectation

L'affectation entre objets est une opération prédéfinie qui peut être surchargée. Si le programmeur ne le fait pas, tout se passe comme si le compilateur avait synthétisé une opération d'affectation consistant en la recopie membre à membre des objets. S'il s'agit d'une classe sans objets membres, sans classe de base et sans fonction virtuelle, cela donne l'opérateur d'affectation trivial, consistant en la copie bit à bit d'une portion de mémoire sur une autre (c'est la copie des structures du langage C).

Les raisons qui poussent à surcharger l'opérateur d'affectation pour une classe sont les mêmes que celles qui poussent à écrire un constructeur par copie (cf. § 2.2.2). Très souvent, c'est que la classe possède des « bouts dehors » (c'est-à-dire des membres de type pointeur) et qu'on ne peut pas se contenter d'une copie superficielle.

L'opérateur d'affectation doit être surchargé par une fonction membre (dans une affectation on ne souhaite évidemment pas que les opérandes jouent des rôles symétriques) :

```
class Chaine
    {
    char *caracteres;
public:
    Chaine(char * = "");
    ~Chaine()
        { delete caracteres ; }
    Chaine &operator=(const Chaine&);
    ...
    };

Chaine::Chaine(char *s)
    {
    caracteres = new char[strlen(s) + 1];
    strcpy(caracteres, s);
    }

Chaine &Chaine::operator=(const Chaine &c)
    {
    if (this != &c)
        {
        delete caracteres;
        caracteres = new char[strlen(c.caracteres) + 1];
        strcpy(caracteres, c.caracteres);
        }
    return *this;
    }
```

REMARQUE. Une construction consiste à donner une première valeur à un objet qui n'existait pas l'instant précédent, alors qu'une affectation consiste à donner une nouvelle valeur à un objet qui en avait déjà une. C'est la raison pour laquelle la surcharge de l'opérateur d'affectation se compose souvent, comme ici, du code du destructeur suivi du code du constructeur par copie.

3.2.2 Operateurs *new* et *delete*

Cette section peut être sauté en première lecture.

Ces opérateurs peuvent être surchargés pour une classe, indépendamment de leur éventuelle surcharge au niveau global (cf. § 1.10.2). Cette surcharge se fait par une fonction membre qui doit être implémentée comme une fonction statique (elle ne dispose donc pas du pointeur **`this`** et n'accède pas aux membres non statiques de l'objet).

La surcharge pour une classe *C* d' l'opérateur **`new`** doit être une fonction membre de prototype

```
void *C::operator new(size_t);
```

Cette fonction est appelée pour allouer des objets *C*, avec pour paramètre effectif la taille de l'objet à allouer (attention, cette taille n'est pas nécessairement **`sizeof(`***C***`)`**, pensez à l'allocation d'objets de classes dérivées), immédiatement avant l'appel du constructeur de l'objet. Le résultat renvoyé doit être l'adresse de la zone de mémoire alloué.

La surcharge pour une classe *C* de **`delete`** doit être une fonction membre de prototype

```
void C::operator delete(void *);
```

Cette fonction est appelée lors de l'évaluation d'une expression **`delete`** concernant un pointeur sur un objet de *C*, immédiatement après l'appel du destructeur de *C*. Exemple purement démonstratif (qui ne sert à rien) :

```
class Point
    {
public:
    Point(int a = 0, int b = 0)
        { x = a; y = b; }
    void *operator new(size_t);
    void operator delete(void *);
    ...
    };

void *Point::operator new(size_t t)
    {
    Point *r = ::new Point;
    cout << "new pour un Point\n";
    return r;
    }

void Point::operator delete(void *p)
    {
    cout << "delete pour un Point\n";
    ::delete static_cast<Point *>(p);      // l'oubli de :: tue!
    }
```

Ces opérations s'utilisent de manière transparente pour le programmeur :

```
Point *p;
...
p = new Point(2, 3);
...
delete p;
```

L'opérateur **new** peut comporter des paramètres supplémentaires, il devient alors un opérateur de placement (cf. § 1.10.3). Exemple :

```
const int MAX = 100;
Point ptable[MAX], *pniveau = ptable;

void *Point::operator new(size_t t, bool dynamique)
    {
    return dynamique ? ::new Point : pniveau++;
    }
```

La fonction **operator delete** peut avoir un paramètre supplémentaire, de type **size_t**. Lors de l'appel, ce paramètre contiendra la taille de l'espace à libérer ; tout cela est transparent pour l'utilisateur :

```
void Point::operator delete(void *p, size_t t)
    {
    espaceRendu += t;
    ::delete (Point *) p;
    }
```

Pour l'allocation ou la restitution d'un tableau, ce sont les fonctions **void ***C**::operator new[](size_t)** et **void** C**::operator delete[](void *)** qui sont appelées :

```
void *Point::operator new[](size_t t)
    {
    return ::new Point[t / sizeof(Point)];     // Ceci craint!
    }
```

3.2.3 Indexation

L'opérateur **[]** doit nécessairement être surchargé par une fonction membre :

```
class Vecteur
    {
    double *tab;
    int nbr;
public:
    Vecteur(int n)
        { tab = new double[nbr = n]; }
    double &operator[](int i)
        { assert(0 <= i && i < nbr); return tab[i]; }
    };
```

Emploi :

```
int n, i;
...
vecteur v(10);
...
V[i] = 0;
...
cout << v[i];
```

Il est parfois commode de donner deux surcharges qui ne différent que par la qualification **const** de l'une des deux. Cela permet de définir un accès indexé aux objets constants distinct de l'accès indexé aux objets non constants :

```
class Vecteur
    {
    ...
public:
    ...
    double &operator[](int i)
        { assert(0 <= i && i < nbr); return tab[i]; }
    double operator[](int i) const
        { assert(0 <= i && i < nbr); return tab[i]; }
    };

void uneFonction(Vecteur t, const Vecteur k)
    {
    t[0] = 0;
    k[0] = 0;   // ERREUR (l'operande gauche doit être une lvalue)

    cout << t[0] << '\n';
    cout << k[0] << '\n';
    }
```

Le type de l'indice n'a pas à être nécessairement entier : on peut définir des tableaux indexés par des objets de n'importe quel type. L'opérateur est binaire, on ne peut pas avoir des tableaux à plusieurs indices ; en revanche, on peut avoir des tableaux à un indice dont les éléments sont à leur tour des tableaux.

Exemple, les matrices **NL** × **NC** :

```
const int NL = ...;
const int NC = ...;
```

```
class Matrice
    {
    class Ligne
        {
        double col[NC];
    public:
        double &operator[](int i)
            { return col[i]; }
        } lin[NL];
public:
    Ligne &operator[](int i)
        { return lin[i]; }
    };

Matrice m;

for (int i = 0; i < NL; i++)
    for (int j = 0; j < NC; j++)
        m[i][j] = 100 * (i + 1) + j;
```

Dans ces conditions, l'expression **`x = m[i][j]`** se lit comme :

```
x = m.operator[](i).operator[](j);
x = m.lin[i].operator[](j);
x = m.lin[i].col[j];
```

3.2.4 Appel de fonction

L'appel de fonction avec *n* arguments peut être surchargé par une ou plusieurs fonctions membres, chacune de la forme **`operator()(`**T_1**`,`** … T_n**`)`**, où T_1 … T_n représentent des types.

La surcharge de l'opérateur d'appel de fonction permet une écriture très élégante de l'opération « extraction d'une sous-chaîne d'une chaîne » :

```
class Chaine
    {
    char *cars;
public:
    Chaine(char *s = "", bool avecCopie = true)
        { cars = avecCopie ? strcpy(new char[strlen(s) + 1], s) : s; }
    Chaine operator()(unsigned int, unsigned int);
    ...
    friend ostream &operator<<(ostream &o, const Chaine &c)
        { return o << c.cars; }
    };

Chaine Chaine::operator()(unsigned int debut, unsigned int fin)
    {
    if (debut <= fin && fin < strlen(cars))
        {
        char *p = new char[fin - debut + 2];
        memmove(p, &cars[debut], fin - debut + 1);
        p[fin - debut + 1] = '\0';
        return Chaine(p, false);
        }
    else
        return Chaine("");
    }
```

Essai :

```
Chaine s("Marquise, vos beaux yeux me font mourir d'amour");
cout << s(14, 23) << '\n';
```

affichage obtenu :

```
beaux yeux
```

La surcharge de l'opérateur d'appel de fonction permet également l'écriture de puissantes expressions en rapport avec la représentation des fonctions dans le cadre des *conteneurs*, des *itérateurs* et des *algorithmes* de la bibliothèque standard. A ce sujet, voir § 9.1.2.

3.2.5 L'opérateur ->

L'opérateur **->** doit être surchargé par une fonction *membre* **operator->()**. Il faut savoir que, contrairement à la règle générale, l'expression

```
objet->membre
```

est traduite par :

```
(objet.operator->())->membre
```

Par conséquent, l'opérateur **operator->()** doit renvoyer

- soit un pointeur sur une classe ou une structure,
- soit un objet ou une référence vers un objet pour lequel l'opérateur **->** a été surchargé à son tour.

On notera que l'opérateur **->** surchargé s'applique à un objet, non à un pointeur. Cela semble bizarre, mais découle nécessairement du fait que les opérateurs ne peuvent pas être surchargés pour les types prédéfinis (les pointeurs sont des types prédéfinis).

Exemple : une *TableDePoints* est un tableau de points avec une notion d'« élément courant ». L'opérateur **->** permet de demander à une telle table la réalisation de services qu'elle délègue à l'élément courant :

```
class Point
    {
    int x, y;
public:
    int &X() { return x; }
    int &Y() { return y; }
    ...
    };

class TableauDePoints
    {
    Point *tab;
    int nombre, pointCourant;
public:
    TableauDePoints(int n)
        { tab = new Point[nombre = n]; }

    void debut()    { pointCourant = 0; }
    void suivant()  { pointCourant++; }
    bool fini()     { return pointCourant >= nombre; }

    Point *operator->()
        { return &tab[pointCourant]; }
    };
```

Emploi :

```
TableauDePoints table(10);

...

for (table.debut(); ! table.fini(); table.suivant())
    cout << "[ " << table->X() << ", " << table->Y() << " ]\n";
```

3.2.6 Opérateurs ++ et – –

Pour une classe *C*, l'opérateur **++** *préfixé* peut être surchargé

- par une fonction membre *C*`::operator++()`, ou bien
- par une fonction non membre `operator++(`*C*`)` ou `operator++(` *C*`&)`.

De même, l'opérateur **++** *postfixé* peut être surchargé

- par une fonction membre *C*`::operator++(int)`, ou bien
- par une fonction non membre `operator++(`*C*`, int)` ou `operator++(`*C*`&, int)`.

Dans tous les cas la qualification `const` peut être présente. Explication analogue pour `--`.

Exemple (bidon) :

```
class Point
    {
public:
    Point(int a = 0, int b = 0)
        { x = a; y = b; }
    Point operator++()      { x++; return *this; }
    Point operator++(int)   { y++; return *this; }
    ...
    friend ostream &operator<<(ostream &o, const Point &p)
        { return o << '[' << p.x << ',' << p.y << "]\n"; }
private:
    int x, y;
    };
```

Essai :

```
Point p;

cout << p++ << ++p << p++ << ++p << p++ << ++p << p++ << ++p;
```

affichage obtenu :

```
(4,4)(4,3)(3,3)(3,2)(2,2)(2,1)(1,1)(1,0)
```

(Pourquoi en marche arrière ?).

3.2.7 Cas des énumérations

Les opérateurs peuvent être surchargés pour les énumérations, puisque ces dernières ne sont pas des types prédéfinis. Exemple canonique :

```
enum Jour { lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche };

Jour operator++(Jour &j, int)
    {
    Jour j0 = j;
    j = Jour((j + 1) % (dimanche + 1));
    return j0;
    }

void creerMois(Jour mois[], Jour premier, int nbrJours)
    {
    for (int i = 0; i < nbrJours; i++)
        mois[i] = premier++;
    }

void main()
    {
    Jour janvier99[31];

    creerMois(janvier99, vendredi, 31);
    ...
    }
```

3.3 Conversions définies par l'utilisateur

Une conversion définie par l'utilisateur fait nécessairement intervenir une classe *C*. Deux sens sont possibles :

- d'un type quelconque vers *C*, en munissant *C* de constructeurs de conversion,
- de *C* vers un type quelconque, en surchargeant pour *C* les opérateurs de conversion.

Une fois définie, une conversion de l'utilisateur est mise en œuvre par le compilateur aussi bien lors de conversions explicites (opérateurs *cast*) que dans des conversions implicites imposées par le contexte (passage d'arguments à des fonctions, récupération de résultat, affectation, etc.)

3.3.1 Constructeurs de conversion

Un constructeur de conversion est un constructeur qui peut être appelé avec un seul argument :

```
class Point
    {
    ...
    Point(int a)
        { x = y = a; }
    ...
    };
```

Utilisations explicites :

```
Point p = Point(2), q = 3;
```

Utilisation implicite (avec l'exemple du § 3.1.2) :

```
r = p + 5;      // compris comme: r = operator+(p, Point(5));
```

Attention, les conversions implicites peuvent être dangereuses ou dénuées de sens :

```
class Tableau
    {
    int *tab, nombre;
public:
    Tableau(int n)
        { tab = new int(nombre = n); }
    ...
    };

void uneFonction(Tableau t);

Tableau tab(10);        // Oui

uneFonction(5);         // Oui. Mais insensé: uneFonction(Tableau(5)) !
```

Interdiction des conversions implicites

On peut interdire l'utilisation d'un constructeur à un argument pour effectuer des conversions implicites en lui donnant la qualification **`explicit`** :

```
class Tableau
    {
    int *tab, nombre;
public:
    explicit Tableau(int n)
        { tab = new int(nombre = n); }
    ...
    };

void uneFonction(Tableau t);

Tableau tab(10);        // Oui

uneFonction(5);         // ERREUR
```


3.3.2 Opérateurs de conversion

C étant une classe et *T* un type, prédéfini ou non, on définit la conversion de *C* vers *T* par une fonction membre

```
C::operator T()
```

Exemple :

```
class Point
    {
    ...
public:
    Point(int a)                    // conversion int    Point
        { x = y = a; }
    operator int ()                 // conversion Point    int
        { return abs(x) + abs(y); }
    ...
    }
```

3.3.3 Le cas de l'indirection

Une classe peut définir une surcharge de l'opérateur (unaire) ***** ; c'est particulièrement approprié si on peut voir ses objets comme des « pointeurs généralisés » (des clés d'un dictionnaire, des alias d'un système de fichier, etc.). Cependant, il est souvent plus satisfaisant et plus utile de définir non pas une surcharge de *, mais un opérateur de conversion vers un type pointeur.

En effet, si *obj* est un objet d'une classe dans laquelle on a défini *une seule* conversion vers un type pointeur, alors le compilateur s'en sert pour

- interpréter l'expression **obj*, si l'opérateur * n'est pas surchargé lui aussi, et
- interpréter l'expression *obj[i]*, si l'opérateur [ ], n'est pas surchargé de son côté.

L'exemple suivant est peu utile, mais montre la bonne volonté du compilateur :

```
class Message
    {
    char *str;
public:
    Message(char *s)
        { strcpy(str = new char[strlen(s) + 1], s); }
    operator char *() { return str; }
    };

void main()
    {
    Message x("Bonjour");

    cout << x[2] << x[1] << *x << '\n';
    }
```

Ce programme affiche

```
noB
```

☺☺

Chapitre 4. L'héritage

4.1 L'héritage

Le mécanisme de l'*héritage* consiste en la définition d'une classe par *réunion* d'une ou plusieurs classes préexistantes, dites *classes de base directes*, et d'un ensemble de membres spécifiques de la classe nouvellement définie, appelée *classe dérivée*. La syntaxe est :

```
class classe : dérivation classe, dérivation classe, … dérivation classe
    {
    déclarations et définitions de membres spécifiques
    }
```

où *dérivation* représente l'un des mots clés **`private`**, **`protected`** ou **`public`**. Exemple :

```
class Pile : private Tableau
    {
    int niveau;
public:
    void empiler(int);
    int depiler();
    ...
    };
```

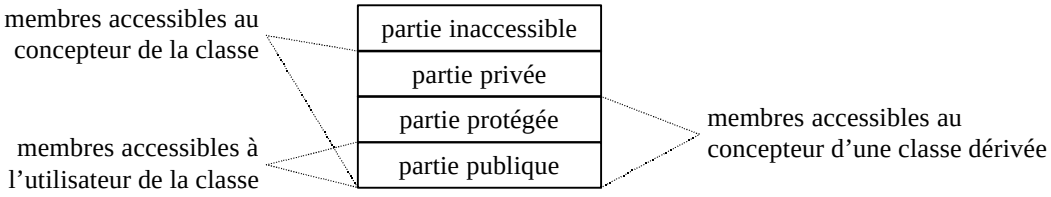
L'héritage est *simple* s'il y a une seule classe de base directe, il est *multiple* sinon.

Etant donnée une classe *C*, une *classe de base* de *C* est soit une classe de base directe de *C*, soit une classe de base directe d'une classe de base de *C*. Une *classe de base accessible* est une classe de base dont les membres publics sont accessibles par les utilisateurs de *C* ; cela signifie que, dans le graphe d'héritage, le chemin qui relie *C* à cette classe est entièrement fait de dérivations publiques (cf. § 4.1.2).

Les unions sont exclues de l'héritage : une union ne peut ni être ni avoir une classe de base.

4.1.1 Membres protégés

En plus des membres publics et privés, une classe peut avoir des membres *protégés*. Annoncés par le mot clé **`protected`**, ils sont « un peu moins » privés que les membres privés, car ils ne sont pas accessibles aux utilisateurs de la classe, mais ils le sont aux concepteurs des classes dérivées :



Exemple :

```
class Tableau
    {
    int *tab;
protected:
    int maxTab;     // on estime que les auteurs des éventuelles
                    // classes dérivées auront à accéder à ce membre
public:
    Tableau(int n)
        { tab = new int(maxTab = n); }
    int &elem(int i)
        { assert(0 <= i && i < maxTab); return tab[i]; }
    };
```

4.1.2 Héritage privé, protégé et public

En choisissant le mot-clé qui introduit la dérivation (**`private`**, **`protected`** ou **`public`**), le programmeur détermine l'accessibilité des membres hérités dans la classe dérivée.

On notera que :

- les objets de la classe dérivée contiennent toujours *tous les membres* de toutes les classes de base,
- il n'est jamais possible d'utiliser la dérivation pour élargir l'accessibilité d'un membre.

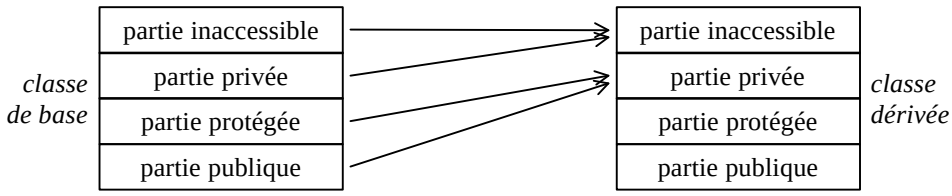
Conséquence des deux points ci-dessus : les objets des classes dérivées peuvent comporter des membres inaccessibles. Typiquement : les membres privés de la classe de base sont présents, mais tout à fait inaccessibles, dans les objets de la classe dérivée.

Héritage privé

Syntaxe :

`class` *classe_dérivée* **`: private`** *classe_de_base*

Le mot clé **`private`** est facultatif (pour les classes, l'héritage privé est l'héritage par défaut). C'est la forme la plus restrictive d'héritage :



Héritage privé

Pour ce qu'en voit le monde extérieur, dans l'héritage privé l'interface (ensemble des membres publics) de la classe de base disparaît lors de la dérivation. Cela veut dire généralement que la classe de base sert à réaliser l'implémentation de la classe dérivée, mais qu'on s'oblige à écrire entièrement une nouvelle interface pour la classe dérivée.

Exemple canonique : la définition d'une classe **`Pile`** comme dérivée de la classe **`Tableau`** déjà introduite. Il s'agit d'héritage privé, car la classe tableau ne fournit que l'implémentation des piles, non leur comportement.

```
class Tableau
    {
    int *table;
protected:
    int nombre;
public:                                 // Comportement des tableaux
    Tableau(int n)
        { table = new int[nombre = n]; }
    int &operator[](int i)
        { assert(0 <= i && i < nombre); return table[i]; }
    ...
    };
```



```
class Pile : private Tableau
    {
    int niveau;
public:                                  // Comportement des piles
    Pile(int max) : Tableau(max)
        { niveau = 0; }
    void empiler(int x)
        { assert(niveau <= nombre); (*this)[niveau++] = x; }
    int depiler()
        { assert(niveau > 0); return (*this)[--niveau]; }
    ...
    };
```

Emploi d'un tableau :

```
Tableau t(dim);

t[i] = x;
...
cout << t[i];
```
etc.

Emploi d'une pile :

```
Pile p(maxPile);

p[i] = x;               // ERREUR: operator[](int) n'est pas accessible
p.empiler(x);           // Oui
...
cout << t[i];           // ERREUR: operator[](int) n'est pas accessible
cout << p.depiler();    // Oui
```
etc.

A retenir : si *D* dérive de manière privée de *B* alors, vu de l'extérieur, un *D* ne peut pas être tenu pour une sorte de *B* (ce qu'on peut demander à un *B*, on ne peut pas le demander à un *D*).

REMARQUE. La déclaration **using** permet de changer le statut dans la classe dérivée d'un identificateur *non privé* de la classe de base :

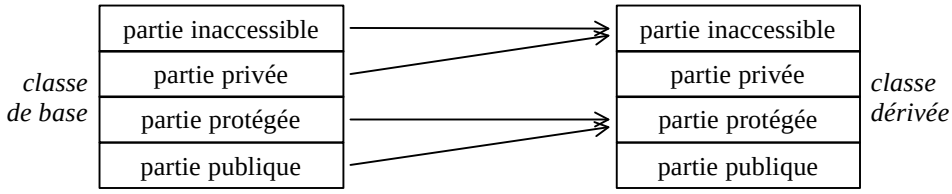
```
class B
    {
public:
    int b1, b2;
    ...
    };

class D : private B
    {
public:
    int d;
    using B::b2;
    ...
    };

void main()
    {
    D unD;
    int x, y, z;

    x = unD.d;          // Oui,     d est public dans D
    y = unD.b1;         // ERREUR:  b1 est privé dans D
    z = unD.b2;         // Oui,     b2 a été rendu public dans D
    }
```

Héritage protégé

C'est un peu la même chose que l'héritage privé (c.-à-d. : la classe de base fournit l'implémentation de la classe dérivée, non son interface) mais on considère ici que les détails de l'implémentation de la classe dérivée doivent être accessibles aux concepteurs d'éventuelles classes dérivées de la classe dérivée.

Syntaxe :

```
class classe_dérivée : protected classe_de_base
```

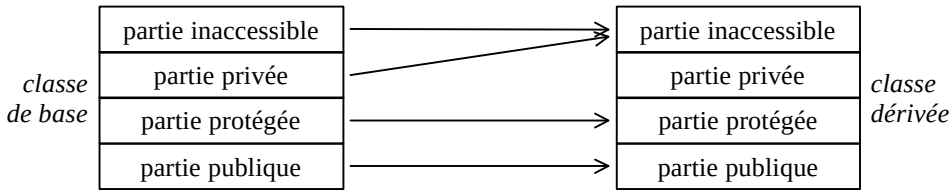


Héritage protégé

Héritage public

Syntaxe :

```
class classe_dérivée : public classe_de_base
```



Héritage public

Dans l'héritage public il y a conservation de l'interface : tous les éléments du comportement de la classe de basse font partie du comportement de la classe dérivée.

C'est la forme d'héritage la plus fréquemment utilisée, car la plus utile et la plus facile à comprendre. Dire que *D* dérive publiquement de *B* c'est dire que « vu de l'extérieur, tout objet *D* est une sorte d'objet *B* » ou, si l'on préfère, qu'un *D* peut répondre à toutes les requêtes qu'on peut soumettre à un *B*.

La plupart des exemples d'héritage examinés dans la suite de ce document seront des cas de dérivation publique.

4.1.3 Redéfinition des membres

Il est fréquent qu'un membre d'une classe de base soit redéfini dans une classe dérivée.

Si la classe *D* dérive publiquement de *B*, tout *D* peut être vu comme un *B* amélioré (augmenté d'autres classes de base, ou de membres spécifiques) ; il est donc naturel qu'un *D* réponde à toutes les requêtes qu'on peut soumettre à un B, et qu'il y réponde *de façon améliorée*. C'est le principe de la redéfinition : la classe dérivée donne des versions enrichies de fonctions de la classe de base. Ces fonctions accèdent notamment aux membres que la classe dérivée a de plus que la classe de base.

On notera qu'il ne s'agit pas ici de *surcharge* des fonctions ni de quelque autre mécanisme complexe, mais simplement de *masquage* : l'identificateur défini ou déclaré dans la classe dérivée masque l'identificateur défini ou déclaré dans la classe de base. Cela ne concerne que les noms, il n'est nullement requis que l'identificateur masquant désigne la même sorte de chose que l'identificateur masqué.

L'opérateur de résolution de portée permet toutefois d'accéder au nom masqué :

```
class Point
    {
    int x, y;
public:
    Point(int a = 0, int b = 0)
        { x = a; y = b; }
    void afficher();
    };
```



```
class Pixel : public Point
    {
    int couleur;
public:
    Pixel(int a = 0, int b = 0, int c = 0)
        : Point(a, b) { couleur = c; }
    void afficher();
    };

void Point::afficher()
    {
    cout << '(' << x << ',' << y << ')';
    }

void Pixel::afficher()
    {
    cout << '[';
    Point::afficher();      // afficher un Pixel en tant que Point
    cout << ';' << couleur << ']';
    }
```

4.2 L'héritage, la création et la destruction des objets

4.2.1 Création et destruction des objets dérivés

Lors de la construction d'un objet, ses parties héritées sont initialisées selon les constructeurs des classes de base. S'il n'y a pas d'indication, il s'agit des constructeurs par défaut. Si des arguments sont requis, il faut le signaler dans le constructeur de l'objet selon une syntaxe analogue à celle de l'initialisation des objets membres :

```
classe(paramètres)
        : classeDeBase(paramètres), ... classeDeBase(paramètres)
    {
    corps du constructeur
    }
```

De la même manière, lors de la destruction d'un objet, les destructeurs des classes de base sont *toujours* appelés (mais là il n'y a rien à écrire, car le destructeur n'a pas de paramètre).

Exemple :

```
class Pile : private Tableau
    {
    int niv;
public:
    Pile(int);
    ~Pile();

    Pile &operator << (int x)  // empiler
        { assert(niv < maxTab); elem(niv++) = x; return *this; }
    int operator -- ()         // depiler
        { assert(niv > 0); return elem(--niv); }
    };

Pile::Pile(int n)
        : Tableau(n)           // ce constructeur commence par un
    {                          // appel (ici explicite) de Tableau(int)
    niv = 0;
    }

Pile::~Pile()
    {
    if (niv > 0)
        cerr << "\nAttention: destruction d'une pile non vide\n";
                               // ce destructeur se termine par un
    }                          // appel (toujours implicite) de ~Tableau()
```

4.2.2 Création et destruction des objets, le cas général

Nous sommes enfin équipés pour donner les versions complètes des réponses à deux questions générales. La première : quelles fonctions sont appelées, et dans quel ordre, lors de la création et la destruction des objets ?

1. Sauf pour les objets qui sont les valeurs de variables globales, le constructeur d'un objet est appelé immédiatement après l'obtention de l'espace pour l'objet.

On peut considérer que l'espace pour les variables globales existe dès que le programme est chargé. Les constructeurs de ces variables sont appelés, dans l'ordre des déclarations de ces variables, juste avant l'activation de la fonction principale du programme (en principe **`main`**).

2. L'activation d'un constructeur commence par les appels des constructeurs des sous-objets hérités de chacune de ses classes de base directes, *dans l'ordre de la déclaration des classes de base*.

3. Viennent ensuite les appels des constructeurs de chacun des objets membres, *dans l'ordre des déclarations des données membres*.

4. Enfin, les instructions qui composent le corps du constructeur sont exécutées.

Par défaut, les constructeurs dont il est question aux points 2 et 3 sont les constructeurs par défaut des classes de base et des classes des données membres. Si des arguments doivent être précisés, par exemple parce que certaines de ces classes n'ont pas de constructeur par défaut, cela se fait selon la syntaxe :

```
classe(parametres)
        : nomDeMembreOuDeClasseDeBase(parametres),
                        ... nomDeMembreOuDeClasseDeBase(paramètres)
    {
    corps du constructeur
    }
```

ATTENTION. Bien noter que l'ordre d'exécution des constructeurs des classes de base et des membres ne correspond pas à l'ordre selon lequel leurs appels ont été écrits dans le constructeur de la classe. Comme dit aux points 2 et 3 , les appels des constructeurs se font toujours dans l'ordre des déclarations.

Les objets sont détruits dans l'ordre inverse de leur construction. Par conséquent :

1. L'activation du destructeur d'un objet commence par l'exécution des instructions qui en composent le corps.

2. Elle continue par l'appel du destructeur de chaque objet membre, dans l'ordre inverse des déclarations des données membres.

3. Est fait ensuite l'appel du destructeur de chaque classe sous-objet hérité, dans l'ordre inverse de la déclaration des classes de base.

4. Enfin, l'espace occupé par l'objet est libéré.

Quand les objets sont-ils détruits ?

Les objets qui sont des valeurs de variables globales sont détruits après la terminaison de la fonction principale (en principe **`main`**), dans l'ordre inverse de la déclaration des variables globales.

Les objets qui ont été dynamiquement alloués ne sont détruits que lors d'un appel de **`delete`** les concernant.

Les objets qui sont des valeurs de variables automatiques sont détruits lorsque le contrôle quitte le bloc auquel elles sont rattachées ou, au plus tard, lorsque la fonction contenant leur définition se termine.

OBJETS TEMPORAIRES. Les objets temporaires ont les vies les plus courtes possibles. En particulier, les objets temporaires créés lors de l'évaluation d'une expression sont détruits avant l'exécution de l'instruction suivante.

Exception, les objets temporaires créés pour initialiser une référence : ils persistent jusqu'à la destruction de la référence (on considère que le résultat d'une fonction est une référence, détruite à la fin de l'instruction contenant l'appel de la fonction).

Attention, les pointeurs ne sont pas traités avec autant de soin, ce qui peut entraîner des erreurs.

4.2.3 Membres synthétisés, le cas général

La deuxième question que nous pouvons nous poser ici est : de quels outils une classe est pourvue, lorsque le programmeur ne s'est pas occupé de cette question, relativement à la création, le clonage et la destruction des objets ?

1. Si le programmeur n'a écrit *aucun* constructeur (quelques autres conditions sont requises, voir ci-dessous), le compilateur synthétise un constructeur avec un corps vide. La construction d'un objet se réduit donc alors à l'appel du constructeur par défaut de chaque classe de base, suivi de l'appel du constructeur par défaut de chaque objet membre.

2. Si le programmeur n'a pas écrit de constructeur par copie, le compilateur synthétise un constructeur par copie appelé « copie membre à membre ». Son exécution consiste en l'appel du constructeur de copie de chaque classe de base, suivi de l'appel du constructeur par copie de chaque objet membre, suivi de la copie « bit à bit » des autres membres.

3. Si le programmeur n'a pas écrit de surcharge de l'opérateur d'affectation, le compilateur synthétise un tel opérateur, par une fonction qui consiste en la copie des parties héritées selon l'opération d'affectation des classes de base, suivie de la copie des objets membres selon l'opération d'affectation de leurs classes respectives, suivie de la copie bit à bit des autres données membres.

4. Si le programmeur n'a pas écrit le destructeur, le compilateur synthétise un destructeur avec un corps vide. La destruction d'un objet se réduit donc à l'appel du destructeur de chaque objet membre, suivi de l'appel du destructeur de chaque classe de base.

Inhibition de la synthèse

Outre le cas où le programmeur a écrit des constructeurs pour la classe, la synthèse par le compilateur du constructeur par défaut est suspendue lorsqu'elle est impossible, c'est-à-dire

- lorsque la classe possède une constante membre, ou
- lorsque la classe possède un membre (non statique) de type référence, ou
- lorsque le constructeur par défaut d'une classe de base n'est pas accessible, ou
- lorsque le constructeur par défaut d'un objet membre n'est pas accessible.

De même, lorsque un membre ou une classe de base possèdent un destructeur par défaut inaccessible, la synthèse du destructeur devient impossible ; or dans ce cas l'écriture d'un destructeur explicite est également impossible. On obtient donc un objet indestructible, qui ne pourra être que dynamiquement alloué et jamais libéré (un objet ayant un destructeur privé est également indestructible).

4.3 Polymorphisme

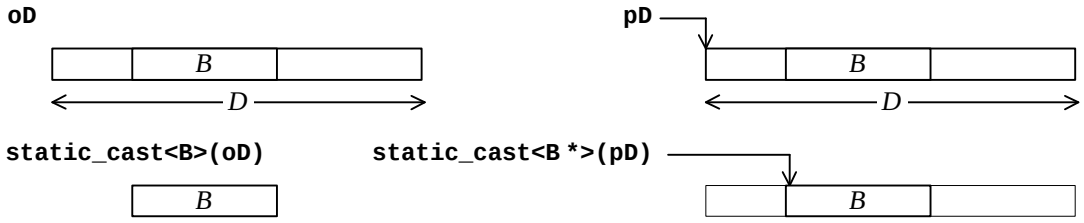
4.3.1 Conversion standard vers une classe de base

Si la classe *D* dérive *publiquement* de la classe *B*, tous les services offerts par un *B* sont offerts aussi par un *D*. Par conséquent, là où un *B* est prévu, on doit pouvoir mettre un *D*. C'est la raison pour laquelle la conversion d'une classe vers une de ses classes de base accessible est définie, et a le statut de *conversion standard*.

On peut voir cette conversion comme l'appel d'une fonction membre de *D* qui serait publique, protégée ou privée selon le mode dont *D* dérive de *B*. Ainsi, la conversion d'une classe dérivée vers une classe de base privée ou protégée existe mais n'est pas accessible ailleurs que depuis l'intérieur de la classe dérivée. Le polymorphisme est intéressant surtout dans le cas de la dérivation publique ; sauf avis contraire, dans cette section nous supposeront être dans ce cas.

Selon qu'elle s'applique à des objets ou à des pointeurs, la conversion vers la classe de base recouvre deux réalités très différentes :

- convertir un objet *D* vers le type *B* c'est lui enlever tous les membres qui ne font pas partie de *B* (les membres spécifiques et ceux hérités d'autres classes) : il y a perte effective d'information ;
- au contraire, convertir un *D* * en un *B* * ne fait perdre aucune information. Pointé à travers un *B* *, l'objet n'offre que les services de la classe *B*, mais il ne cesse pas d'être un *D* avec tous ses membres.



conversion de **oD**, *un objet D, en un objet B*

conversion de **pD**, *un pointeur sur D, en un pointeur sur B (voir aussi § 4.3.2)*

Exemple :

```
class Point
    {
public:
    int x, y;
    Point(int = 0, int = 0);
    };

class Pixel : public Point
    {
public:
    int couleur;
    Pixel(int = 0, int = 0, int = 0);
    };

void travail1(Point);
void travail2(Point *);

void main()
    {
    Pixel pix(1, 2, 3);
    ...
    travail1(pix);
    travail2(&pix);
    ...
    }

void travail1(Point pt)
    {
```

La valeur de **pt** *était peut-être un* **Pixel** *au départ, mais elle est devenue un* **Point** *lors de l'appel. Dans cette fonction,* **pt** *n'a pas de couleur. Aucune conversion (autre que définie par l'utilisateur) ne peut permettre de refaire un* **Pixel** *à partir de* **pt**.

```
    ...
    }

void travail2(Point *pt)
    {
```

Il n'est rien arrivé à la valeur de l'objet pointé par **pt**. *Si on peut garantir que cet objet est un* **Pixel**, *la ligne suivante a un sens (elle est placée sous la responsabilité du programmeur) :*

```
    cout << ((Pixel *) pt)->couleur;
    ...
    }
```

N.B. Pour ce qui est dit ici, les références (pointeurs gérés de manière interne par le compilateur) doivent être considérées comme des pointeurs. Ainsi :

```
void travail3(Point &);

void main()
    {
    Pixel pix(1, 2, 3);
    ...
    travail3(pix);
    ...
    }
```

```
void travail3(Point &rf)
    {
```
Il n'est rien arrivé à la valeur de l'objet référencé par **rf**. *Si on peut garantir que cet objet est un* **Pixel**, *la ligne suivante a un sens (elle est placée sous la responsabilité du programmeur) :*
```
    cout << ((Pixel &) rf).couleur;
    ...
    }
```

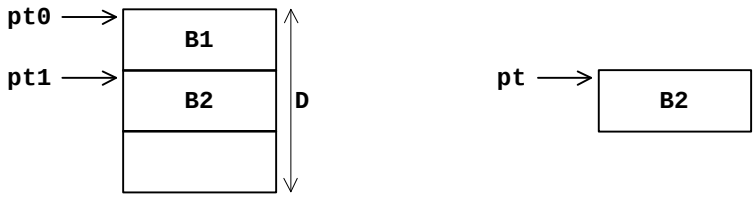
4.3.2 Implantation de la « généralisation » des pointeurs

En C, la conversion d'un T_1 ***** en un T_2 ***** est une opération sans garantie et toujours *sans travail* : la valeur du pointeur ne subit aucune modification.

En C++, la conversion (standard ou forcée par le programmeur) d'un T_1 ***** en un T_2 *****, lorsque T_2 est une classe de base accessible de T_1, est une opération légitime et utile qui, parfois, *implique la modification de la valeur du pointeur*. Ce point surprenant découle de l'héritage multiple :

```
class B1
    { ... };
class B2
    { ... };
class D : public B1, public B2
    { ... };
B2 *pt  = new B2;
D  *pt0 = new D;
B2 *pt1;
pt1 = pt0;
```

Dans cet exemple, la valeur obtenue dans **pt1** doit être celle de **pt0** augmentée de la taille de **B1**, afin que dans la suite des événements, les accès aux membres de **B2** à partir de **pt1** puissent être traités exactement comme ceux qui seront faits à partir de **pt** :



4.3.3 Type statique, type dynamique, généralisation

Considérons la situation suivante :

```
class D : public B
    { ... };
D  objD;
B  objB = objD;
B *ptrB = & objD;
B &refB = objD;
```

Le type de **objB** ne soulève aucune question (c'est un **B**) ; en revanche, il y a des subtilités à dire à propos des types de **ptrB** et **refB**. On dit que

- le type statique de ***ptrB** (ce que **ptB** pointe) et de **refB** est **B** ;
- le type dynamique de ***ptrB** et de **refB** est **D**.

Le type statique d'une expression découle d'une analyse du texte du programme ; il est connu à la compilation. Le type dynamique, au contraire, est déterminé par la valeur courante de l'expression, il peut changer durant l'exécution du programme.

Grâce aux conversions implicites de pointeurs et références vers des pointeurs et références sur des classes de base, *tout objet peut être momentanément tenu pour plus général qu'il n'est*. Cela permet les traitements collectifs :

```
class Figure
    { ... }

class Triangle : public Figure
    { ... }

class Ellipse : public Figure
    { ... }

Figure *image[N];

image[0] = new Triangle(...);
image[1] = new Figure(...);
image[2] = new Ellipse(...);
...
for (int i = 0; i < N ; i++)
    image[i]->dessiner();
```

Si on suppose que la fonction **`dessiner`** est redéfinie dans les classes **`Triangle`**, **`Ellipse`**, etc., il est intéressant de se demander quelle est la fonction effectivement appelée dans l'expression **`image[i]->dessiner()`** écrite ci-dessus.

Par défaut, la détermination du membre accédé à travers un pointeur ou une référence se fait durant la compilation, c'est-à-dire d'après le type statique du pointeur ou de la référence. Ainsi, dans l'expression précédente, c'est la fonction **`dessiner`** de la classe **`Figure`** qui est appelée. Cette réalité décevante obéit à des considérations d'efficacité et de compatibilité avec le langage C ; elle semble limiter considérablement l'intérêt des traitements collectifs.

La section suivante expose un comportement beaucoup plus intéressant.

4.3.4 Liaison dynamique et fonctions virtuelles

Si elle est appelée à travers un pointeur ou une référence, la sélection d'une *fonction virtuelle* se fait d'après le type dynamique (c.-à-d., le type à l'exécution, non à la compilation) de l'objet pointé ou référencé.

On obtient une fonction virtuelle en faisant précéder sa déclaration du mot-clé **`virtual`** :

```
class Figure
    {
    ...
public:
    virtual void dessiner() { ... }
    ...
    };

class Ellipse : public Figure
    {
    ...
public:
    void dessiner() { implémentation du dessin d'une ellipse }
    ...
    };
```

Maintenant, l'appel **`image[i]->dessiner()`** active la fonction **`dessiner`** définie dans la classe de l'objet effectivement pointé par **`image[i]`**. On notera que cet appel est légal parce que la fonction dessiner a été déclarée une première fois dans la classe qui correspond au type statique de **`image[i]`**.

Une fonction virtuelle reste virtuelle dans les classes dérivées : **`dessiner()`** est virtuelle dans **`Ellipse`**, car elle l'est dans **`Figure`** (mais on aurait pu indiquer ce fait en la qualifiant aussi **`virtual`** dans **`Ellipse`**).

Contraintes

En compétition avec le masquage, la redéfinition des fonctions virtuelles subit des contraintes fortes, sur les paramètres et sur le résultat :

1. Tout d'abord, la signature de la redéfinition doit être la même que celle de la première définition ; sinon, non seulement la deuxième définition n'est pas liée à la première, mais en plus elle la masque (et la fonction virtuelle devient inaccessible) :

```
class B
    {
public:
    virtual void f(int) { cout << "B::f(int)\n"; }
    };

class D : public B
    {
public:
    virtual void f(float) { cout << "D::f(float)\n"; }
    };
```

Essais :

```
B *pB = new D;
pB->f(0.5);
```

cela écrit **`B::f(int)`** car, à cause de la différence de signature, **`D::f`** n'est pas une redéfinition de **`B::f`**. Autre essai :

```
D *pD = new D;
pD->f(1);
```

cela écrit **`D::f(float)`** car **`B::f`** et **`C::f`** ne sont pas en compétition pour la surcharge (**`D::f`** masque **`B::f`**).

2. La contrainte sur le résultat découle de l'utilisation de la fonction. Supposons que **`f`** est une fonction virtuelle rendant un résultat de type T_1, et qu'une redéfinition de **`f`** rend un résultat de type T_2. Pour que l'expression

```
r = p->f();
```

soit cohérente, il faut que le résultat de **`p->f()`** puisse être converti dans le type de **`r`**, c'est-à-dire que :

- soit $T_1 = T_2$,
- soit T_1 est un pointeur [resp. une référence] sur un objet d'une classe C_1, T_2 est un pointeur [resp. une référence] sur un objet d'une classe C_2 et C_1 est une *classe de base accessible* de C_2.

On peut énoncer ce deuxième cas, plus simplement, en disant : si une fonction virtuelle *f* rend l'adresse d'un *T*, alors toute redéfinition de *f* doit rendre l'adresse d'une sorte de *T*.

Si la contrainte sur la signature est satisfaite, le compilateur identifie un cas de redéfinition d'une fonction virtuelle ; si alors la contrainte sur le résultat n'est pas satisfaite, il annonce une erreur.

La magie des fonctions virtuelles, un exemple canonique

Les fonctions virtuelles permettent de programmer des traitements généraux alors que des détails sont encore inconnus :

```
class Figure
    {
    Couleur couleur;
public:
    virtual void dessiner() { ... }
    void effacer();
    ...
    };
```

```
void Figure::effacer()
    {
    couleur = couleurDuFond;          // trait invisible
    dessiner();
    couleur = couleurDuTrait;         // trait visible
    };
```

Un objet **Figure** n'est pas plus capable de s'effacer qu'il ne l'est de se dessiner, mais toute classe dérivée qui définira une version effective de **dessiner** disposera aussitôt d'une bonne version effective d'**effacer**, héritée de **Figure**. Il est important de noter que quand **effacer** est compilée la fonction **dessiner** qui sera effectivement appelée n'existe pas encore ; pourtant, il ne sera pas nécessaire de recompiler ultérieurement **effacer** pour la faire fonctionner.

Quand déclarer une fonction virtuelle ?

Le comportement d'une fonction virtuelle n'est différent de celui d'une fonction ordinaire que si la fonction virtuelle est appelée pour un objet d'une classe T_1 à travers un pointeur ou une référence sur une classe T_2.

Déclarer virtuelles des fonctions qui n'ont pas besoin de l'être n'a pour conséquence qu'un petit encombrement supplémentaire de la classe (une entrée dans la **vtable**) et un peu de temps supplémentaire lors de l'appel (une indirection) ; ce n'est pas bien cher. Si on veut malgré tout ne déclarer des fonctions virtuelles que lorsque c'est indispensable, on doit examiner les points suivants :

- la classe en question possédera-t-elle des classes dérivées ?
- les objets de la classe dérivée seront-ils accédés à travers des pointeurs ou des références ?
- la fonction en question sera-t-elle être redéfinie dans les classes dérivées ?

4.3.5 Implémentation des fonctions virtuelles

Une classe possédant une fonction virtuelle (spécifique ou héritée) s'appelle une *classe polymorphe*.

Chaque objet d'une classe polymorphe possède, en plus des membres explicitement déclarés et hérités, un pointeur **vptr** vers une table nommée **vtable** qui contient les adresses des fonctions virtuelles de la classe. L'appel d'une fonction virtuelle se fait via un élément de cette table.

1. Commençons par l'héritage simple. Exemple, avec les déclarations :

```
class A
    {
    int ma;
public:
    virtual void f();
    virtual void g();
    virtual void h();
    ...
    };

A *pa;
```

un appel comme

```
pa->g()
```

qui, en l'absence de fonctions virtuelles, aurait été traduit par (le passage de **pa** comme paramètre effectif caché permet l'initialisation de l'argument formel **this**) :

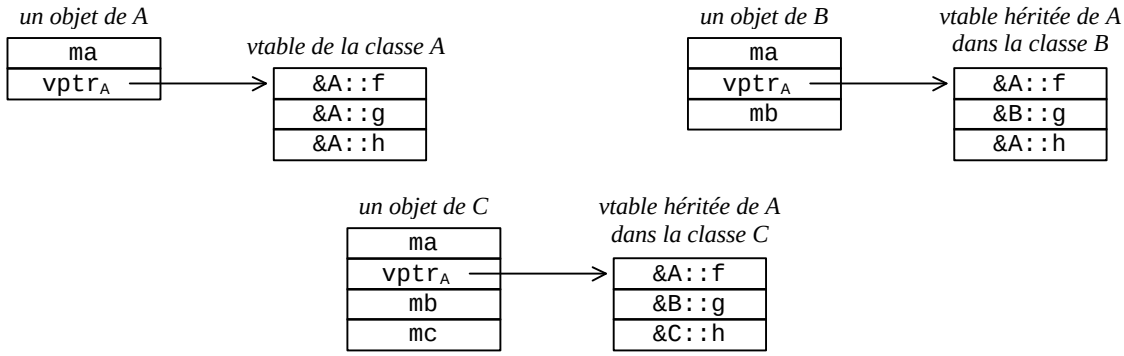
```
A::g(pa)
```

sera traduit ici en quelque chose comme :

```
(*pa->vptr[1])(pa)
```

Quand une classe est polymorphe ses classes dérivées le sont aussi. La création d'un objet d'une telle classe implique l'initialisation du pointeur **`vptr`** avec l'adresse de la **`vtable`** de la classe de l'objet :

```
class B : public A
    {
    int mb;
public:
    virtual void g();
    };

class C : public B
    {
    int mc;
public:
    virtual void h();
    };
```



Comme le montre la figure, l'expression produite durant la compilation

```
(*pa->vptr[1])(pa)
```

active une fonction ou une autre selon le type effectif de l'objet pointé par **`pa`**.

2. En présence d'héritage multiple, l'implémentation des fonctions virtuelles se complique passablement. Considérons les classes suivantes :

```
class B1
    {
    int b1;
public:
    virtual void f();
    };

class B2
    {
    int b2;
public:
    virtual void f();
    };

class D : public B1, public B2
    {
    int d;
public:
    void f();
    };
```

et les déclarations

```
D *pD = new D;
B1 *pB1 = pD;
B2 *pB2 = pD;
```

Puisque **`pD`**, **`pB1`** et **`pB2`** pointent tous un objet de **`D`**, les appels

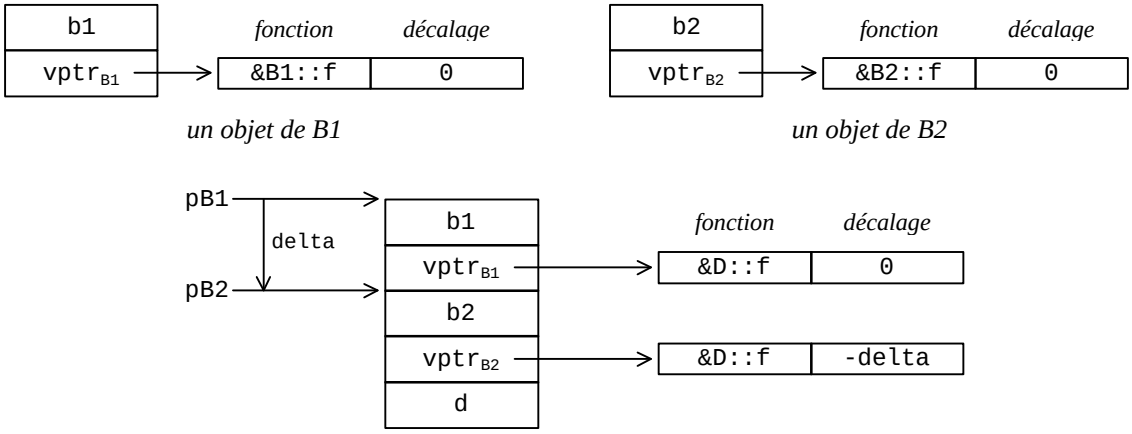
```
pD->f();
pB1->f();
pB2->f();
```

doivent être des appels de la même fonction **D::f** sur le même objet ***pD**. Si le compilateur suivait le schéma expliqué pour l'héritage simple, ces appels seraient traduits par des expressions respectivement analogues à :

```
(*pD->vptrD[0])(pD)
(*pB1->vptrB1[0])(pB1)
(*pB2->vptrB2[0])(pB2)
```

et, dans la fonction **D::f**, le pointeur **this** aurait **pB1** pour valeur lors du second appel et **pB2** lors du troisième. Cela ne peut pas fonctionner correctement, car les valeurs de **pB1** et **pB2** ne sont pas égales (cf. § 4.3.2).

Ce problème ne peut pas être traité par le compilateur, puisque celui-ci ne peut pas distinguer les endroits où la fonction est appelée à travers un pointeur résultant d'une conversion qui a fait un décalage du pointeur des endroits où elle est appelée à travers un pointeur sans conversion ou résultant d'une conversion sans décalage. On complique donc la table des fonctions virtuelles, en associant à chacune l'adresse de la fonction à appeler et le décalage à appliquer à l'adresse passée à cette fonction comme argument, pour avoir la bonne valeur du pointeur **this** :



Un appel tel que

```
pB2->f();
```

sera maintenant traduit en quelque chose comme :

```
(*pB2->vptrB2[0].fonction)(pB2 + pB2->vptrB2[0].decalage);
```

4.3.6 Appel d'une fonction virtuelle et contrôle du compilateur

L'appel d'une fonction virtuelle comporte donc le choix, selon des critères dynamiques, d'une fonction parmi plusieurs fonctions possibles. Mais cet appel est contrôlé statiquement par le compilateur ; ce qui influe sur :

1. Les droits d'accès :

```
class B
    {
public:
    virtual void f();
    };

class D : public B
    {
    void f();
    }

D d;
B *pB = &d;
D *pD = &d;
```

```
pB->f();        // Oui, appel de D::f()
pD->f();        // ERREUR : pas le droit d'accès
```

2. Les arguments par défaut :

```
class B
    {
public:
    virtual void f(int = 0);
    };

class D : public B
    {
public:
    void f(int);
    }

D d;
B *pB = &d;
D *pD = &d;

pB->f();        // Oui, appel de D::f(0)
pD->f();        // ERREUR : il faut un argument
```

3. La résolution de la surcharge :

```
class B
    {
public:
    virtual void f(int);
    };

class D : public B
    {
public:
    virtual void f(int);
    virtual void f(char);
    };

D d;
B *pB = &d;
D *pD = &d;

pB->f('A');     // appelle D::f(int)
pD->f('A');     // appelle D::f(char)
```

4.3.7 Constructeurs, destructeurs et opérateurs virtuels

Pas de constructeurs virtuels

Un objet n'a pas de type tant qu'il n'est pas entièrement construit. Deux conséquences :

- un constructeur ne peut être virtuel,
- l'utilisation d'une fonction virtuelle à l'intérieur d'un constructeur est très délicate.

Le programme suivant affiche deux fois **`fruit`** (son programmeur espérait probablement autre chose) :

```
class Fruit
    {
public:
    char *quoi;
    Fruit() { initialiser(); }

    virtual void initialiser() { quoi = "fruit"; }
    };
```

```
class Pomme : public Fruit
    {
public:
    virtual void initialiser() { quoi = "pomme"; }
    };

void main()
    {
    Fruit f;
    Pomme p;
    cout << f.quoi << ' ' << p.quoi << '\n';
    }
```

Création et clonage d'objets dont le type n'est pas complètement connu

Comment créer un objet de même type, ou un clone d'un objet donné, sans que le type précis de ce dernier soit connu à la compilation ? Les fonctions virtuelles fournissent une solution :

```
class Fruit
    {
    Fruit();
    Fruit(Fruit&);
public:
    virtual Fruit *nouveau()   { return new Fruit; }
    virtual Fruit *clone()     { return new Fruit(*this); }
    ...
    };

class Pomme : public Fruit
    {
    Pomme();
    Pomme(Pomme&);
public:
    Pomme *nouveau()   { return new Pomme; }
    Pomme *clone()     { return new Pomme(*this); }
    ...
    };
```

Avec ces déclarations et

```
Fruit *f, *g, *h;
```

les expressions

```
g = f->nouveau();
h = f->clone();
```

affectent à **g** et à **h** des pointeurs sur des objets dont le type dynamique est celui de **f**.

Destructeurs virtuels

Le destructeur d'une classe peut être virtuel, même si ce destructeur ne fait rien, et cela est même une obligation si la classe est la racine d'une hiérarchie de classes polymorphes. Exemple :

```
class Article
    {
    int numero;
public:
    Article(int r)     { numero = r; }
    virtual ~Article()  { }
    ...
    };
```

```
class ArticleNomme : public Article
    {
    char *nom;
public:
    ArticleNomme(int r, char *d) : Article(r)
        { strcpy(nom = new char[strlen(d) + 1], d); }
    ~ArticleNomme()
        { delete [] nom; }
    };
```

Utilisation :

```
Article *p = new ArticleNomme(10101, "Cocotte minute SEB");
...
delete p;
```

Le destructeur de la classe **Article** ne fait rien, mais ayant été déclaré virtuel, l'expression **delete p** provoque en fait l'appel de **~ArticleNomme()**, ce qui libère l'espace occupé par le nom de l'article.

Notez que **~Article()** sera quand même appelé, puisque le destructeur d'un objet se termine toujours par les appels des destructeurs des classes de base (cela aurait été utile, si **~Article()** n'avait pas été vide).

Affectation virtuelle

La question se complique sérieusement en présence de certains opérateurs, comme l'affectation :

```
class Article
    {
    int ref;
    float prix;
public:
    Article(int r, float p)
        { ref = r; prix = p; }
    ...
    };

class ArticleFrais : public Article
    {
    long date;
public:
    ArticleFrais(int r, float p, long d) : Article(r, p)
        { date = d; }
    ...
    };
```

Utilisation :

```
Article *p, *q;
...
q = new ArticleFrais(100250, 99.50, 1299);
...
*p = *q;
```

L'affectation ci-dessus n'a pas l'effet voulu, car p étant de type statique **Article ***, la valeur du membre **date** de l'**ArticleFrais** pointé par **q** n'est pas copiée (qu'est-ce qui permet de penser qu'il y a, dans ce que pointe ***p** *avant l'affectation*, assez de place pour y copier le membre **date** ?)

On doit donc surcharger l'opérateur d'affectation. Attention, la première idée n'est pas bonne :

```
class Article
    {
    ...
public:
    virtual Article &operator=(const Article &y)
        { ref = y.ref; prix = y.prix; return *this; }
    ...
    };
```

```
class ArticleFrais : public Article
    {
    ...
public:                                                    // VERSION ERRONEE
    virtual ArticleFrais &operator=(const ArticleFrais &y)
        { date = y.date; Article::operator=(y); return *this; }
    ...
    };
```

Cela ne marche pas car, le paramètre formel n'étant pas de même type, le membre **operator=** de la classe **ArticleFrais** ne redéfinit pas le membre de même nom de la classe **Article** mais, au contraire, il le masque.

La solution est passablement alambiquée et peu efficace (l'opérateur **dynamic_cast** est expliqué au § 4.5.1) :

```
class ArticleFrais : public Article
    {
    ...
public:
    virtual ArticleFrais &operator=(const Article &y)
        {
        Article::operator=(y);
        try
            {
            const ArticleFrais &z = dynamic_cast<const ArticleFrais &>(y);
            date = z.date;
            }
        catch (bad_cast)
            {                   // Ou bien signaler une erreur
            }                   // (car l'affectation sera incomplète)
        return *this;
        }
    };
```

On notera que l'affectation

```
*p = *q;
```

est traduite par

```
(*p).operator=(*q);
```

Ainsi, c'est le type dynamique de ***p** qui détermine lequel, des opérateurs **Article::operator=** et **ArticleFrais::operator=** est effectivement appelé. Or, le type dynamique de ***p** est le type de ce que **p** pointe *avant l'affectation*. Le service obtenu est donc bien celui qu'il fallait : si l'**Article** pointé par **q** est un **ArticleFrais**, celui-ci sera ou bien complètement copié (par **ArticleFrais::operator=**) ou bien tronqué (par **Article::operator=**) selon que dans ce que **p** pointe il a assez d'espace pour recevoir tous les membres de ***q** ou bien seulement ceux qui font un **Article**.

4.4 Fonctions virtuelles pures et classes abstraites

Souvent une fonction virtuelle est déclarée une première fois dans une classe placée, dans la hiérarchie d'héritage, à un niveau tellement général que la fonction ne peut pas recevoir une implémentation vraisemblable ou utile.

On peut en faire une *fonction virtuelle pure* en écrivant « **= 0** » à la fin de son prototype :

```
class Figure
    {
    ...
public:
    virtual void dessiner() = 0;
    ...
    };
```


D'une part, cela « officialise » le fait que la fonction ne peut pas, à ce niveau, posséder une implémentation. D'autre part, cela crée, pour le programmeur, l'*obligation* de définir cette implémentation dans une classe dérivée ultérieure.

Une fonction virtuelle pure reste virtuelle pure dans les classes dérivées, aussi longtemps qu'elle ne fait pas l'objet d'une redéfinition.

Une classe contenant des fonctions virtuelles pures s'appelle une *classe abstraite*. Tenter de créer des objets d'une classe abstraite est une erreur, signalée par le compilateur :

```
class Cfigure
    {
    ...
public:
    virtual void dessiner() = 0;
    ...
    };

class CRectangle : public Cfigure
    {
    int x1, y1, x2, y2;
    ...
public:
    void dessiner()
        {
        trait(x1, y1, x2, y1);
        trait(x2, y1, x2, y2);
        trait(x2, y2, x1, y2);
        trait(x1, y2, x1, y1);
        }
    ...
    };
```

Utilisation :

```
CFigure f;          // ERREUR:  On ne peut pas créer des instances
                    //          de la classe abstraite Figure
CFigure *pt;        // Oui
pt = new CFigure;   // ERREUR:  On ne peut pas créer des instances
                    //          de la classe abstraite Figure
CRectangle r;       // Oui
```

Dans un sens plus large, une *classe abstraite* est une classe qui ne doit pas avoir des objets. C'est généralement une classe insuffisamment détaillée pour représenter un objet réel. De telles classes sont très utiles, par exemple pour « mettre en facteur » les éléments communs à d'autres classes (qui, elles, ont des objets).

Il y a donc deux manières principales de définir de telles classes sans objets :

- définir une classe contenant au moins une fonction virtuelle pure,
- définir une classe (sans amis) dont tous les constructeurs sont protégés.

4.5 Identification dynamique du type

Le coût du polymorphisme, en temps, est celui d'une indirection[1] supplémentaire pour chaque appel de fonction virtuelle. En espace, le coût est constant : chaque objet d'une classe polymorphe a un encombrement supplémentaire égal à la taille d'un pointeur, quel que soit le nombre de fonctions virtuelles de la classe. D'où l'idée de profiter de l'existence de ce pointeur pour ajouter au langage, sans coût supplémentaire, une gestion des types dynamiques qu'autrement les programmeurs devraient écrire eux-mêmes (probablement à l'aide de fonctions virtuelles).

Fondamentalement, ce mécanisme se compose des deux opérateurs **`dynamic_cast`** et **`typeid`**.

---

[1] *Indirection*, on dit aussi *déréférence* : obtention d'un accès à l'objet pointé à partir d'un accès au pointeur.

4.5.1 L'opérateur *dynamic_cast*

Syntaxe

```
dynamic_cast<type>(expression)
```

Il s'agit d'effectuer la conversion d'une expression d'un type pointeur vers un autre type pointeur, les types pointés étant deux classes *polymorphes* dont l'une, *B*, est une classe de base accessible de l'autre, *D*. S'il s'agit de généralisation (conversion dans le sens $D\ \texttt{*} \rightarrow B\ \texttt{*}$), cet opérateur ne fait rien de plus que la conversion standard. Le cas intéressant est $B\ \texttt{*} \rightarrow D\ \texttt{*}$, conversion qui n'a de sens que si l'objet pointé par l'expression de type *B* * est en réalité une sorte de *D*.

L'opérateur **dynamic_cast** fait ce travail de manière sûre et portable, et rend l'adresse de l'objet lorsque c'est possible, 0 dans le cas contraire :

```
class Animal
    {
public:
    ...
    virtual ~Animal()       // au moins une fonction virtuelle
        { ... }             // (il faut des classes polymorphes)
    };

class Mammifere : public Animal
    { ... };

class Chien : public Mammifere
    { ... };

class EpagneulBreton: public Chien
    { ... };

class Chat : public Animal
    { ... };

class Reverbere
    { ... };

Chien medor;
Animal *ptr = &medor;

Mammifere *p0 = ptr;                // ERREUR à la compilation

Mammifere *p1 = dynamic_cast<Mammifere *>(ptr);
                                    // Oui, p1 reçoit une bonne adresse

EpagneulBreton *p2 = dynamic_cast<EpagneulBreton *>(ptr);
                                    // Oui, mais p2 reçoit 0

Chat *p3 = dynamic_cast<Chat *>(ptr);
                                    // Oui, p3 reçoit 0

Reverbere *p4 = dynamic_cast<Reverbere *>(ptr);
                                    // Oui, p4 reçoit 0
```

L'opérateur **dynamic_cast** s'applique également aux références. L'explication est la même, sauf qu'en cas d'impossibilité d'effectuer la conversion, cet opérateur lance l'exception **bad_cast**.

```
Chien Medor;
Animal &ref = Medor;

Mammifere &r1 = dynamic_cast<Mammifere &>(ref);
                            // Oui, r1 référence le Mammifere qu'est Medor

EpagneulBreton &r2 = dynamic_cast<EpagneulBreton &>(ref);
                            // L'exception bad_cast est lancée;
                            // non attrapée, elle tuera le programme
```

4.5.2 L'opérateur *typeid*

Syntaxes :


```
typeid(type)
```

ou

```
typeid(expression)
```

Le résultat est une valeur de type **const type_info&**, où **type_info** est une classe de la bibliothèque standard comportant au moins les membres suivants :

```
class type_info
    {
public:
    const char *name() const;
    int operator==(const type_info&) const;
    int operator!=(const type_info&) const;
    int before(const type_info&) const;
    ...
    };
```

On a la possibilité de comparer les types dynamiques et d'en obtenir l'écriture des noms :

```
Chien Medor;
Animal * ptr = & Medor;

cout << typeid(ptr).name() << '\n';
cout << typeid(*ptr).name() << '\n';

cout << "L'animal pointé par ptr "
     << ( typeid(*ptr) == typeid(Chien) ? "est" : "n'est pas" )
     << " un chien\n";
```

affichage obtenu :

```
class Animal *
class Chien
L'animal pointé par ptr est un chien
```

La fonction **before** rend 0 ou 1. Elle implémente une relation d'ordre total entre les types, qui n'a pas de lien avec la relation d'héritage et qui peut varier d'un système à un autre, voire d'une exécution à une autre..

4.6 L'héritage multiple

4.6.1 L'héritage multiple et ses ambiguïtés

Puisque C++ pratique l'héritage multiple, une classe peut hériter, depuis des classes de base directes, plusieurs membres ayant le même nom. Une telle situation n'est contrôlée ni par le mécanisme de la surcharge, ni par celui du masquage et redéfinition des membres : les identificateurs hérités, tous visibles, créeront des ambiguïtés et les accès ne pourront se faire qu'en utilisant l'opérateur de résolution de portée.

```
class Lumiere
    {
public:
    float intensite;
    ...
    };

class Son
    {
public:
    float intensite;
    ...
    };

class Spectacle : public Lumiere, public Son
    {
    ...
    };

Spectacle s;
```

```
s.intensite = 0;                // ERREUR: ambiguïté
s.Lumiere::intensite = 0;       // Oui
s.Son::intensite = 0;           // Oui
```

Si le conflit de noms est le résultat d'un malheureux hasard, on peut améliorer la situation en « habillant » par des fonctions les noms problématiques (avec des fonctions en ligne on ne perd aucune efficacité) :

```
class Spectacle : public Lumiere, public Son
    {
public:
    float &luminosite() { return Lumiere::intensite; }
    float &volume()     { return Son::intensite; }
    ...
    };

s.luminosite() = 0;             // Oui
s.volume() = 0;                 // Oui
```

Dans d'autres cas le conflit est inévitable car il ne s'agit pas de membres homonymes sans rapport, mais des mêmes membres, hérités par des chemins différents.

Exemple : les deux classes suivantes décrivent les *ordinateurs* et les *téléphones* :

```
class Ordinateur
    {
public:
    void afficherCaracteristiques() const
        { ... }
    ...
    };

class Telephone
    {
public:
    void appeler(const char *numero)
        { ... }
    ...
    };
```

Un objet **`Portable`** est un appareil qui fonctionne sur batterie :

```
class Portable
    {
    int chargeBatterie;
public:
    Portable(int chargeInit)
        { chargeBatterie = chargeInit; }
    void rechargerBatterie()
        { chargeBatterie = 100; }
    ...
    };
```

Bien entendu, il y a des ordinateurs et des téléphones qui fonctionnent sur batterie :

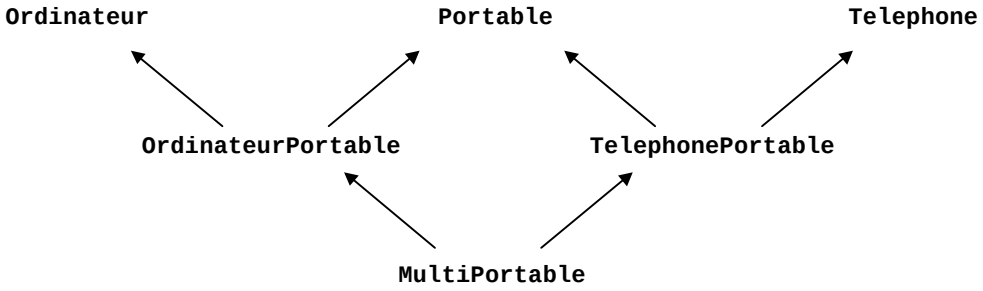
```
class OrdinateurPortable : public Ordinateur, public Portable
    {
public:
    OrdinateurPortable()
            : Portable(0)   // la batterie des ordinateurs est livrée vide
        { ... }
    ...
    };
```

```
class TelephonePortable : public Telephone, public Portable
    {
public:
    TelephonePortable()
            : Portable(100) // la batterie des téléphones est livrée pleine
        { ... }
    ...
    };
```

Enfin, il existe des ordinateurs portables qui permettent de téléphoner :

```
class MultiPortable : public OrdinateurPortable, public TelephonePortable
    {
    ...
    };
```

Le graphe d'héritage (avec l'héritage multiple ce n'est plus un arbre) n'est plus élémentaire :



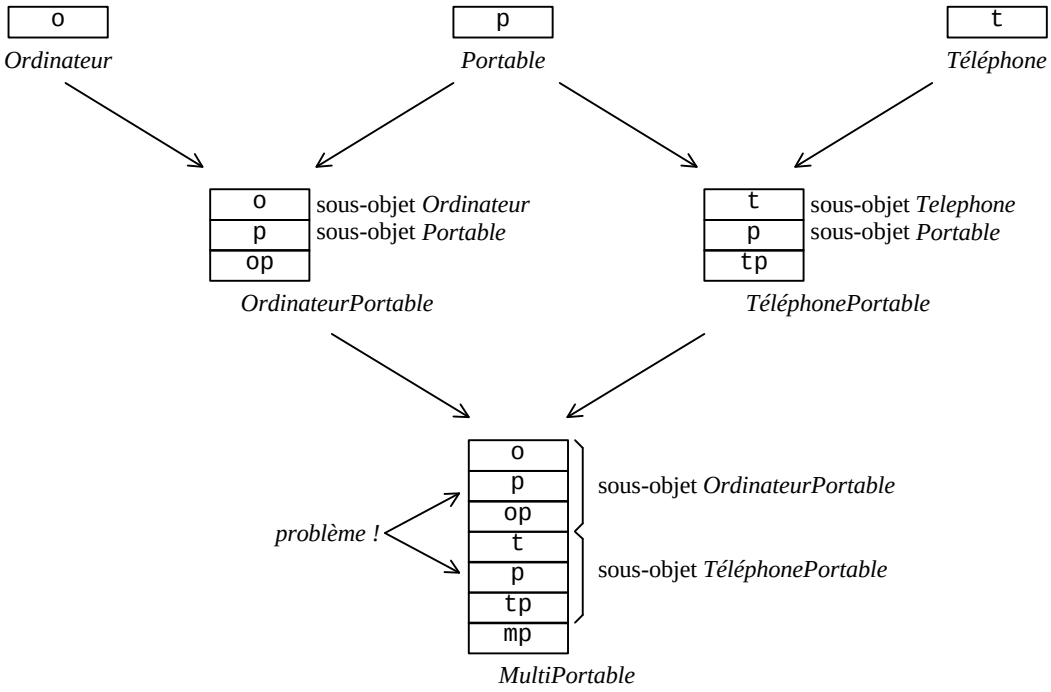
Cette situation n'est pas saine : les membres de **`Portable`** se retrouvent en double dans **`MultiPortable`**, ce qui pose un problème de cohérence et de gestion de l'espace (un **`MultiPortable`** aura deux batteries !).

Par exemple, nous pouvons observer les tailles des objets :

```
cout << "Ordinateur         : " << sizeof(Ordinateur)         << '\n';
cout << "Telephone          : " << sizeof(Telephone)          << '\n';
cout << "Portable           : " << sizeof(Portable)           << '\n';
cout << "OrdinateurPortable : " << sizeof(OrdinateurPortable) << '\n';
cout << "TelephonePortable  : " << sizeof(TelephonePortable)  << '\n';
cout << "MultiPortable      : " << sizeof(MultiPortable)      << '\n';
```

Si on suppose que dans chaque classe on a déclaré pour 100 octets de membres spécifiques, on obtient :

```
Ordinateur          : 100
Telephone           : 100
Portable            : 100
OrdinateurPortable  : 300
TelephonePortable   : 300
MultiPortable       : 700
```



Héritage multiple (sans dérivation virtuelle)

Il faut noter que, même en acceptant de gaspiller de l'espace (c.-à-d. en conservant dans un **`MultiPortable`** deux sous-objets **`Portable`** dont l'un ne servirait pas) on ne résoudrait pas le problème de cohérence posé par cette situation. Par exemple, la conversion standard d'un **`MultiPortable`** en un **`Portable`** est devenue ambiguë :

```
MultiPortable *pmp = new MultiPortable;
Portable *pp = pmp;                    // ERREUR: ambiguïté
```

On peut résoudre le problème en explicitant *par quel chemin* on veut faire la conversion :

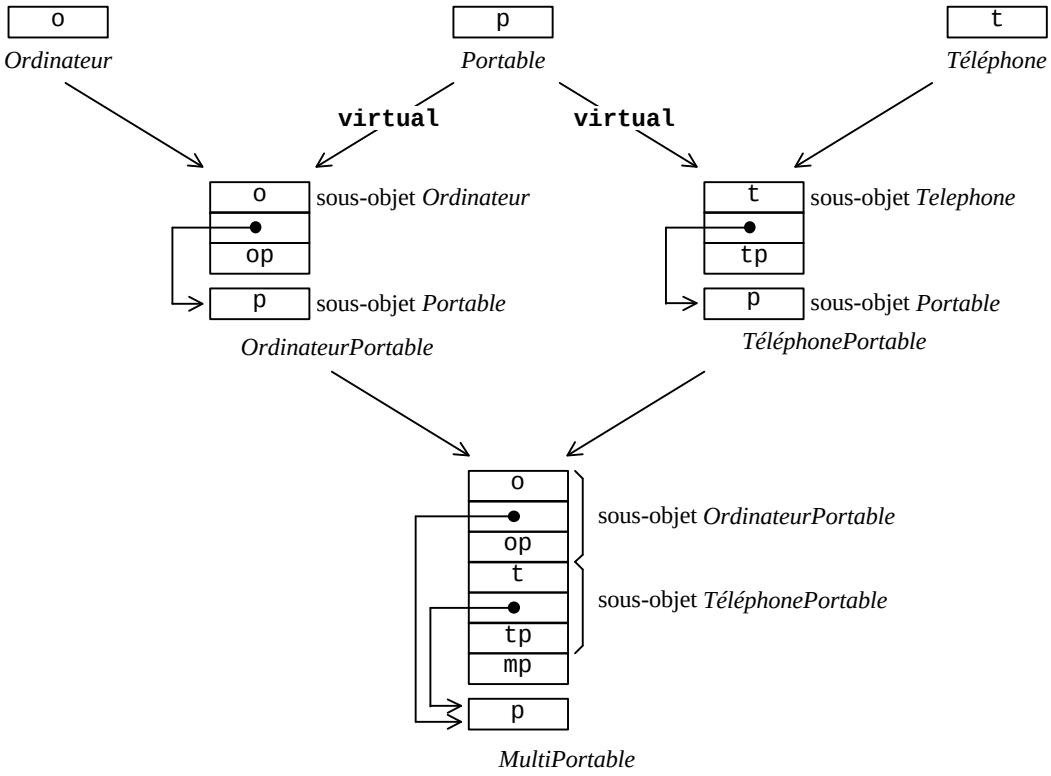
```
MultiPortable *pmp = new MultiPortable;
OrdinateurPortable *pop = pmp;         // OK, un seul OrdinateurPortable
                                       // dans un MultiPortable
Portable *pp = pop;                    // OK, un seul Portable
                                       // dans un OrdinateurPortable
```

mais c'est évidemment pénible et peu fiable.

4.6.2 Classes de base virtuelles

La bonne manière de régler le problème précédent est d'indiquer que **`Portable`** est une *classe de base virtuelle* de **`OrdinateurPortable`** et de **`TelephonePortable`**. Les membres doublement hérités ne se retrouvent alors plus qu'en un seul exemplaire :

```
class Ordinateur
    { ... };
class Telephone
    { ... };
class Portable
    { ... };
class OrdinateurPortable : public Ordinateur, virtual public Portable
    { ... };
class TelephonePortable : public Telephone, virtual public Portable
    { ... };
class MultiPortable : public OrdinateurPortable, public TelephonePortable
    { ... };
```



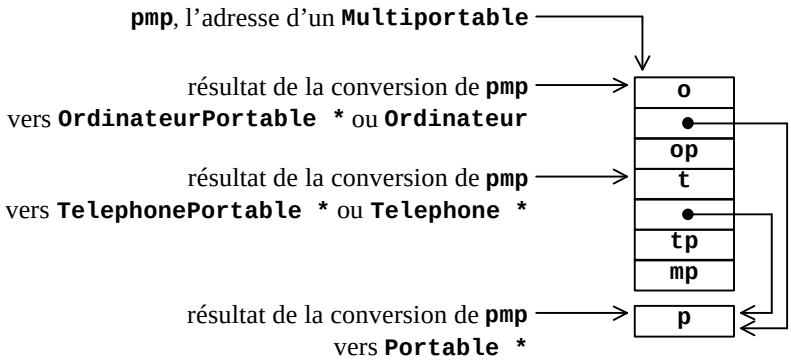
Héritage multiple et dérivation virtuelle

L'affichage obtenu maintenant est :

```
Ordinateur          : 100
Telephone           : 100
Portable            : 100
OrdinateurPortable  : 304
TelephonePortable   : 304
MultiPortable       : 608
```

Avec ce schéma, la conversion d'un *MultiPortable* vers un *Portable* n'est plus ambiguë :

```
MultiPortable *pmp = new MultiPortable;
Portable *pp = pmp;                    // OK, un seul Portable
                                       // dans un MultiPortable
```



4.6.3 Héritage virtuel et constructeurs

Il reste un problème, celui de la construction d'un sous-objet hérité (virtuellement) à travers plusieurs chemins du graphe d'héritage. Ici, cela donne : lors de la construction d'un **`Multiportable`**, faut-il construire le sous-objet **`Portable`** en considérant qu'un **`Multiportable`** est une sorte d'**`OrdinateurPortable`** ou bien en considérant que c'est une sorte de **`TelephonePortable`** ? Ou, plus simplement : un **`Multiportable`** est-il livré avec la batterie vide ou avec la batterie pleine ?

La stratégie adoptée est la suivante : pour éviter de telles ambiguïtés, les sous-objets provenant de classes de base virtuelles *indirectes* (c.-à-d. éloignées de plus d'un niveau) sont initialisés d'office avec les constructeur par défaut correspondants.

De plus, la syntaxe de l'initialisation des classes de base directes peut alors être utilisée pour initialiser ces sous-objets lors de la construction d'objets de la classe dérivée.

Version erronée :

```
class Ordinateur    { ... };

class Telephone     { ... };

class Portable
    {
    int chargeBatterie;
public:
    Portable(int chargeInit)
        { chargeBatterie = chargeInit; }
    ...
    };

class OrdinateurPortable : public Ordinateur, public Portable
    {
public:
    OrdinateurPortable()
        : Portable(0) { }
    ...
    };

class TelephonePortable : public Telephone, public Portable
    {
public:
    TelephonePortable()
        : Portable(100) { }
    ...
    };

class MultiPortable : public OrdinateurPortable, public TelephonePortable
    {
public:
    MultiPortable()
        : OrdinateurPortable(), TelephonePortable() { }
    };

MultiPortable mp;       // ERREUR, constructeur par défaut non trouvé
```

Version correcte, où un **`MultiPortable`** est créé avec une batterie remplie aux trois quarts (75%) :

```
class Ordinateur    { ... };

class Telephone     { ... };

class Portable
    {
    int chargeBatterie;
public:
    Portable(int chargeInit = 0)
        { chargeBatterie = chargeInit; }
    ...
    };

class OrdinateurPortable : public Ordinateur, public Portable
    {
public:
    OrdinateurPortable()
        : Portable(0) { }
    ...
    };
```



```
class TelephonePortable : public Telephone, public Portable
    {
public:
    TelephonePortable()
        : Portable(100) { }
    ...
    };

class MultiPortable : public OrdinateurPortable, public TelephonePortable
    {
public:
    MultiPortable()
        : OrdinateurPortable(), TelephonePortable(), Portable(75) { }
    };

MultiPortable mp;
```

☺ ☹ ☹

Chapitre 5. Les modèles

Un modèle définit une famille de fonctions ou de classes paramétrée par une liste d'identificateurs qui représentent des valeurs et des types. Les valeurs sont indiquées par leurs types respectifs, les types par le mot réservé **`class`**. Le tout est préfixé par le mot réservé **`template`**. Exemple :

```
template<class T, class Q, int N>
    ici figure la déclaration ou la définition d'une fonction ou d'une classe
    où T et Q apparaissent comme types et N comme constante
```

Le mot **`class`** ne signifie pas que *T* et *Q* sont nécessairement des classes, mais des types. Depuis la norme ISO on peut également utiliser le mot réservé **`typename`** (cf. § 5.2.2), qui est un terme plus heureux.

La production effective d'un élément de la famille que définit un modèle s'appelle l'*instanciation* du modèle. Lors de l'instanciation, les paramètres qui représentent des types doivent recevoir pour valeur des types ; les autres paramètres doivent recevoir des expressions ayant une valeur connue durant la compilation.

5.1 Modèles de fonctions

L'instanciation d'un modèle de fonction

```
template<par1, ... park>
    type nom ( argfor1, ... argforn ) ...
```

s'obtient par l'appel d'une des fonctions définies par le modèle, écrit sous la forme :

```
nom<arg1, ... argk>(argeff1, ... argeffn)
```

où les derniers arguments du modèle, arg_i … arg_k (éventuellement tous) sont facultatifs, si leurs valeurs peuvent être déduites sans ambiguïté de $argeff_1$, … $argeff_n$.

Les erreurs faites sur les modèles ont tendance à n'apparaître qu'au moment de l'instanciation car, tant qu'il n'est pas instancié, un modèle ne subit qu'une compilation très superficielle.

Exemple :

```
template<class T>
    T min(T tab[], int n)
        {
        T min = tab[0];
        for (int i = 1; i < n; i++)
            if (tab[i] < min) min = tab[i];
        return min;
        }

int t[] = { 10, 5, 8, 14, 20, 3, 19, 7 };
```

Instanciation du modèle :

```
cout << min<int>(t, 8);                          // T = int
cout << min<char>("BKEFYFFLKRNFAJDQKXJD", 20);   // T = char
```

ou bien, puisque l'argument du modèle peut se déduire de l'appel :

```
cout << min(t, 8);                              // T = int
cout << min("BKEFYFFLKRNFAJDQKXJD", 20);        // T = char
```

Un modèle de fonction peut coexister avec une ou plusieurs *spécialisations*. Par exemple, voici une spécialisation du modèle **min** qu'on ne pourrait pas obtenir par instanciation de ce dernier :

```
char *min(char *tab[], int n)
    {
    char *min = tab[0];
    for (int i = 1; i < n; i++)
        if (strcmp(tab[i], min) < 0) min = tab[i];
    return min;
    }
```

Lors d'un appel effectif, les fonctions ordinaires (spécialisations de modèles) sont préférées aux instances de modèles et les instances de modèles plus spécialisés sont préférées aux instances de modèles moins spécialisés. Par exemple, compliquons l'exemple précédent en nous donnant

- un modèle avec deux paramètres-types :

```
template<class T1, class T2>
    int imin(T1 ta[], T2 tb[], int n)
        {
        cout << "imin(T1 [], T2 [], int)";
        int m = 0;
        for (int i = 1; i < n; i++)
            if (ta[i] < ta[m] || ta[i] == ta[m] && tb[i] < tb[m])
                m = i;
        return m;
        }
```

- un modèle qui en est une spécialisation partielle :

```
template<class T>
    int imin(char *ta[], T tb[], int n)
        {
        cout << "imin(char * [], T2 [], int)";
        int m = 0, r;
        for (int i = 1; i < n; i++)
            {
            r = strcmp(ta[i], ta[m]);
            if (r < 0 || r == 0 && tb[i] < tb[m])
                m = i;
            }
        return m;
        }
```

- une fonction qui est une spécialisation complète du modèle :

```
int imin(char *ta[], char *tb[], int n)
    {
    cout << "imin(char * [], char * [], int)";
    int m = 0, r;
    for (int i = 1; i < n; i++)
        {
        r = strcmp(ta[i], ta[m]);
        if (r < 0 || r == 0 && strcmp(tb[i], tb[m]) < 0)
            m = i;
        }
    return m;
    }
```

Essayons tout cela :

```
int ti1[5] = { 10,  5, 15,  5, 18 };
int ti2[5] = { 20, 10, 20,  8, 20 };
```



```
char *ts1[5] = { "Andre", "Denis", "Charles", "Andre", "Bernard" };
char *ts2[5] = { "Duchamp", "Dubois", "Dumont", "Dupont", "Durand" };

void main()
    {
    cout << " : " << imin(ti1, ti2, 5) << '\n';
    cout << " : " << imin(ts1, ti2, 5) << '\n';
    cout << " : " << imin(ts1, ts2, 5) << '\n';
    }
```

Affichage obtenu :

```
imin(T1 [], T2 [], int) : 1
imin(char * [], T2 [], int) : 3
imin(char * [], char * [], int) : 0
```

5.2 Modèles de classes

5.2.1 Syntaxe

Un modèle de classe est un type paramétré, par d'autres types et des valeurs connues lors de la compilation.

Par exemple, définissons un modèle pour représentant des tableaux dans lesquels la valeur de l'indice est vérifiée lors de chaque accès[2]. Le type des éléments de ce tables n'est pas connu, c'est un paramètre du modèle :

```
template<class TElement>
    class TableSure
        {
        TElement *tab;
        int nbr;
    public:
        TableSure(int n)
            { tab = new TElement[nbr = n]; }
        ~TableSure()
            { delete [] tab; }
        TElement &operator[](int i)
            { assert(0 <= i && i < nbr); return tab[i]; }
        TElement operator[](int i) const
            { assert(0 <= i && i < nbr); return tab[i]; }
        };
```

Utilisations :

```
TableSure<char> message(80);
TableSure<Point> ligne(100);

message[0] = ' ';
ligne[i] = Point(j, k);
```

Les fonctions membres d'un modèle de classe sont des modèles de fonctions avec les mêmes paramètres que le modèle de classe. Cela est implicite pour les déclarations et définitions écrites à l'intérieur du modèle de classe, mais doit être explicité pour les définitions écrites dehors. Voici une autre manière d'écrire notre classe :

[2] Sauf si le programme est compilé avec la pseudo-constante **`NDEBUG`** définie. Dans ce cas, la pseudo-fonction **`assert`** devenant vide, il n'y a pas de contrôle et les accès à de telles tables sont aussi efficaces que les accès aux tableaux ordinaires.

```
template<class TElement>
    class TableSure
        {
        TElement *tab;
        int nbr;
    public:
        TableSure(int);
        ~TableSure();
        TElement &operator[](int);
        TElement operator[](int) const;
        };

template<class TElement>
    TableSure<TElement>::TableSure(int n)
        { tab = new TElement[nbr = n]; }

template<class TElement>
    TableSure<TElement>::~TableSure()
        { delete [] tab; }

template<class TElement>
    TElement &TableSure<TElement>::operator[](int i)
        { assert(0 <= i && i < nbr); return tab[i]; }

template<class TElement>
    TElement TableSure<TElement>::operator[](int i) const
        { assert(0 <= i && i < nbr); return tab[i]; }
```

Les paramètres des modèles de classes peuvent avoir des valeurs par défaut :

```
template<class TElement = int, int N = 100>
class TableFix
    {
    TElement tab[N];
public:
    TElement &operator[](int i)
        { assert(0 <= i && i < N); return tab[i]; }
    TElement operator[](int i) const
        { assert(0 <= i && i < N); return tab[i]; }
    };

TableFix<float, 10> t1;     // Oui, table de 10 float
TableFix<float> t2;         // Oui, table de 100 float
TableFix<> t3;              // Oui, table de 100 int
TableFix t4;                // ERREUR
```

Un modèle peut être déclaré et non défini. Comme une classe il pourra alors être utilisé dans toute situation ne nécessitant pas de connaître sa taille ou ses détails internes :

```
template<class TElement = int, int N = 100> class TableFix;

TableFix<float> *ptr;          // Oui
TableFix<float> table1;        // ERREUR
extern TableFix<float> table2; // Oui (pour la compilation)
```

5.2.2 Qualification *typename*

Introduit par la norme ISO, le mot réservé **typename** a été récemment ajouté au langage C++. Il s'utilise en rapport avec les modèles, de deux manières.

Première utilisation : **typename** est un terme plus heureux que **class** pour signaler, parmi les paramètres des modèles, ceux qui sont des types. Ainsi, l'expression

```
template<class T, class Q, int N>
```
définition du modèle (T et Q apparaissent comme types, N comme constante)

peut s'écrire de manière équivalente :

```
template<typename T, typename Q, int N>
```
définition du modèle (T et Q apparaissent comme types, N comme constante)

Deuxième utilisation : **typename** permet de déclarer qu'un identificateur est le nom d'un type, dans un contexte où cela n'est pas certain. En dehors des modèles il n'y a pas d'ambiguïté, car les noms des types sont connus. Dans les modèles, la situation est parfois confuse, pour le compilateur et pour le programmeur. Pour la clarifier, on convient que les identificateurs apparaissant dans un modèle sont supposés ne pas désigner des types sauf si

- ils ont été explicitement déclarés comme types, ou
- ils sont qualifiés par le mot réservé **typename**.

Dans l'exemple suivant, la qualification **typename** devant **Paire::type_cle** est nécessaire, car rien ne permet d'assurer que l'identificateur **type_cle** (identificateur appartenant à la portée du type effectif qui aura été mis à la place de **Paire** lors de l'instanciation du modèle) est le nom d'un type[3] :

```
template<class Paire>
    class ListeAssociative
        {
        Paire::type_cle clemin, clemax;                    // ERREUR
        typename Paire::type_cle cle_min, cle_max;     // Oui

        /*...*/

        };

struct Membre
    {
    typedef long type_cle;
    typedef char *type_information;

    type_cle cle;
    type_cle info;
    };

...
ListeAssociative<FicheMembre> a;
...
```

5.2.3 Instanciation

L'instanciation d'un modèle de classe est faite par le compilateur lorsque cela est nécessaire :

```
template<class A, class B>
    class pair
        {
    public:
        A first;
        B second;
        pair(A a, B b) : first(a), second(b) { }
        ...
        };

pair<int, char> x(65, 'a');         // intanciation nécessaire

pair<int, char> *ptr;               // instanciation non nécessaire ici
ptr = new pair<int, char>(66, 'b'); // maintenant oui
```

Puisque les arguments-types des modèles de fonctions sont déduits automatiquement lorsque cela est possible, l'utilisation d'un modèle de fonction peut alléger les expressions (autres que des déclarations) instanciant des modèles :

[3] Le compilateur C/C++ analyse une déclaration « complexe » en commençant par y reconnaître l'élément qui désigne un type « simple ».

```
template<class A, class B>
    pair<A, B> make_pair(A a, B b)
        {
        return pair<A, B>(a, b);
        }
```

maintenant, l'expression

```
make_pair(65, 'a')
```

construit et renvoie l'objet

```
pair<int, char>(65,'a')
```

Exemple :

```
void affichage(pair<int, int> &p)
    {
    cout << '[' << p.car << ',' << p.cdr << "]\n";
    }

affichage(make_pair(65, 'a'));
```

On peut pousser plus loin ce mécanisme, et imaginer *affichage* ainsi écrite :

```
template<class A, class B>
    void affichage(pair<A, B> &p)
        {
        cout << '[' << p.car << ',' << p.cdr << "]\n";
        }

affichage(make_pair(65, 'a'));
```

Maintenant, l'expression **`affichage(make_pair(65, 'a'))`** provoque l'instanciation du modèle de classe **`pair`** et des deux modèles de fonction **`make_pair`** et **`affichage`**.

Un argument d'un modèle peut être à son tour instance d'un modèle. Exemple :

```
template<class T, int N>
    class Table
        {
        T tab[N];
    public:
        T &operator[](int i)
            { assert(0 <= i && i < N); return tab[i]; }
        };
```

Ce modèle permet de déclarer et utiliser des matrices de **`NL`** ligne et **`NC`** colonnes :

```
const int NL = 5;
const int NC = 8;

Table<Table<double, NC>, NL> t;        // Matrice NL × NC

for (int i = 0; i < NL; i++)
    for (int j = 0; j < NC; j++)
        t[i][j] = i + j / 10.0;
```

Initialisation des membres statiques des modèles de classes.

Chaque instance d'un modèle de classe a ses propres membres statiques. S'ils sont de même type, comme dans :

```
template<class T> class Machin
    {
    ...
    static int nombreObjets;
    };
```

alors on peut les initialiser « tous d'un coup » par une expression générique, comme :

```
template<class T> int Machin<T>::nombreObjets = 0;
```

Si les membres statiques des instances du modèle ne sont pas forcément de même type, comme dans :


```
template<class T> class Machin
    {
    ...
    static T elementNeutre;
    };
```

alors une expression comme

```
template<class T> T Machin<T>::elementNeutre = 0;
```

n'initialise que les membres statiques des instances pour lesquelles **0** est convertible vers le type **T** :

```
Machin<int> i;          // Oui
Machin<float>  f;       // Oui
Machin<char *> s;       // Oui
Machin<Point>  p;       // ERREUR
```

Il faut donc ajouter une spécialisation de l'initialisation du membre statique :

```
template<class T> T Machin<T>::elementNeutre = 0;

Point Machin<Point>::elementNeutre = Point(0, 0);
```

5.2.4 Spécialisation

Comme les modèles de fonction, les modèles de classe peuvent posséder des spécialisations :

```
template<class T>
    class Vecteur
        {
        T *tab;
        int nbr;
    public:
        Vecteur(int n)
            { tab = new T[nbr = n]; }
        T &operator[](int i)
            { return tab[i]; }
        int encombrement()
            { return nbr * sizeof(T); }
        void hi() { }                   // pour le test
        };

class Vecteur<char *>
    {
    char **tab;
    int nbr;
public:
    Vecteur(int n)
        { tab = new char *[nbr = n]; }
    char * &operator[](int i)
        { return tab[i]; }
    int encombrement();
    void ho() { }                       // pour le test
    };

int Vecteur<char *>::encombrement()
    {
    int r = 0;
    for (int i = 0; i < nbr; i++)
        r += strlen(tab[i]) + 1;
    return r;
    }
```

```
Vecteur<float>  tabNombres(10);
Vecteur<char *> tabChaines(10);

tabNombres.hi();           // Oui
tabNombres.ho();           // ERREUR: membre inexistant
tabChaines.hi();           // ERREUR: membre inexistant
tabChaines.ho();           // Oui
```

REMARQUES

1. La classe **`Vecteur<char *>`** doit être ainsi notée pour mettre en évidence qu'il s'agit d'une spécialisation de **`Vecteur<T>`**. Une déclaration telle que

```
class Vecteur
    {
    char **tab;
    ...
    };
```

serait refusée par le compilateur.

2. En toute rigueur, la syntaxe complète pour déclarer une spécialisation d'une fonction ou d'une classe requiert de la faire précéder la déclaration de la spécialisation par l'expression **`template<>`** :

```
template<> class Vecteur<char *>
    {
    ...
    };
```

Dans la plupart des cas, cependant, l'expression **`template<>`** se révèle facultative.

5.2.5 Modèles de classe et héritage

Tous les cas de figure sont utiles et acceptés par le compilateur :

1. Une classe peut dériver d'une instance de modèle :

```
class PileEntiers : private TableSure<int>
    {
    int top;
public:
    PileEntiers(int max)
            : TableSure<int>(max)
        { top = 0; }
    bool vide()
        { return top == 0; }
    void empiler(int x)
        { (*this)[top++] = x; }
    int depiler()
        { return (*this)[--top]; }
    };
```

essai :

```
void main()
    {
    PileEntiers pi(100);

    for (int i = 0; i < 10; i++)
        pi.empiler(1001 + i);
    while ( ! pi.vide() )
        cout << pi.depiler() << ' ';
    }
```


2. Un modèle peut dériver d'un autre :

```
template<class TElement>
    class Pile : private TableSure<TElement>
        {
        int top;
    public:
        Pile(int max)
                : TableSure<TElement>(max)
            { top = 0; }
        bool vide()
            { return top == 0; }
        void empiler(TElement x)
            { (*this)[top++] = x; }
        TElement depiler()
            { return (*this)[--top]; }
        };
```

essai :

```
void main()
    {
    Pile<char> pc(100);

    for (int i = 0; i < 26; i++)
        pc.empiler('A' + i);
    while ( ! pc.vide() )
        cout << pc.depiler();
    }
```

3. Un modèle peut dériver d'une classe. Par exemple, la classe **`ListeBasique`** implémente les opérations sur les listes chaînées : insertion d'un élément, extraction d'un élément, etc. Pour rester indépendante du type des objets manipulés, elle suppose que ces derniers sont donnés par leurs adresses, qu'elle manipule comme des pointeurs génériques **`void *`**. Cette démarche est très pratique, mais dangereuse, car elle contourne complètement le système des types :

```
class ListeBasique
    {
    class Maillon
        {
        void *info;
        Maillon *suiv;
        Maillon(void *i, Maillon *s)
            { info = i; suiv = s; }
        friend class ListeBasique;
        };
    Maillon *premier, *dernier;
public:
    ListeBasique()  { premier = 0; }
    bool vide()     { return premier == 0; }
    void entrer(void *);
    void *sortir();
    };

void ListeBasique::entrer(void *i)
    {
    Maillon *p = new Maillon(i, 0);
    if (premier == 0)
        premier = p;
    else
        dernier->suiv = p;
    dernier = p;
    }
```

```
void *ListeBasique::sortir()
    {
    if (premier == 0)
        throw "Extraction depuis une liste vide";
    Maillon *p = premier;
    premier = premier->suiv;
    void *r = p->info;
    delete p;
    return r;
    }
```

Le modèle **`Liste`** crée une sorte de « carrosserie » autour de la classe **`ListeBasique`** permettant de réinstaller les contrôles de type :

```
template<class TElement> class Liste : public ListeBasique
    {
public:
    void entrer(TElement &x)
        { ListeBasique::entrer(&x); }
    TElement &sortir()
        { return *static_cast<TElement *>(ListeBasique::sortir()); }
    };
```

La fonction **`Liste<T>::entrer`** semble ne rien faire. En réalité, elle garantit que seuls des objets de type **`T`** entrent dans une **`Liste<T>`** ; en particulier, elle justifie le changement de type qui est fait dans la fonction **`Liste<T>::sortir`**.

Essayons tout cela, avec une classe **`Entier`** purement démonstrative :

```
class Entier
    {
    int valeur;
public:
    Entier(int v)
        { valeur = v; }
    friend ostream &operator<<(ostream &o, const Entier e)
        { return o << e.valeur; }
    };

Entier tab[5] = { Entier(1), Entier(2), Entier(3), Entier(4), Entier(5) };

void main()
    {
    Liste<Entier> l;
    for (int i = 0; i < 5; i++)
        l.entrer(tab[i]);
    while ( ! l.vide() )
        cout << l.sortir() << ' ';
    ...
    }
```

Autre essai :

```
void main()
    {
    ...
    for (int i = 0; i < 5; i++)
        l.entrer( * new Entier(100 + i));
    while ( ! l.vide() )
        {
        Entier &r = l.sortir();
        cout << r << ' ';
        delete &r;
        }
    }
```


5.2.6 L'instanciation en pratique

La visibilité d'un modèle se réduit au fichier dans lequel il est rencontré par le compilateur ; si un modèle est utilisé dans plusieurs fichiers, il faut l'écrire dans un fichier en-tête.

Si un modèle de fonction, ou un modèle de classe comportant des fonctions écrites en dehors de la classe, est utilisé avec les mêmes types arguments dans plusieurs fichiers sources, alors les mêmes fonctions seront instanciées en plusieurs exemplaires, lors des compilations séparées de fichiers distincts. Reconnaissant qu'il s'agit d'instances identiques d'un même modèle, l'éditeur de liens, au lieu de signaler l'erreur « multiple définition d'une fonction », fait en sorte qu'une seule instance soit incorporée à l'exécutable final.

Exemple (de principe) : la confection d'un exécutable à partir des deux sources suivants produit une erreur à l'édition de liens, car la fonction **coucou** est définie deux fois.

Fichier **travail.cpp** :

```
void coucou(int)
    { cout << "Bonsoir\n"; }

void travail()
    { coucou(0); }
```

Fichier **principal.cpp** :

```
void coucou(int)
    { cout << "Bonjour\n"; }

void main()
    {
    void travail();

    coucou(0);
    travail();
    }
```

Mais il n'y a pas d'erreur dans la configuration suivante : fichier **travail.cpp** :

```
template<class T>
void coucou(T)
    { cout << "Bonsoir\n"; }

void travail()
    { coucou(0); }
```

Fichier **principal.cpp** :

```
template<class T>
void coucou(T)
    { cout << "Bonjour\n"; }

void main()
    {
    void travail();

    coucou(0);
    travail();
    }
```

Une seule fonction **coucou** est incorporée à l'exécutable produit. Selon le compilateur utilisé on obtiendra l'affichage soit de deux fois **Bonjour**, soit de deux fois **Bonsoir**.

☺☹☺

Chapitre 6. Les exceptions

6.1 Exceptions

6.1.1 Principe et syntaxe

L'auteur d'une bibliothèque de fonctions peut détecter les erreurs à l'exécution, mais ne sait pas ce qu'il faut faire ensuite. L'utilisateur de la bibliothèque sait comment réagir aux erreurs, mais ne peut pas les détecter. Le mécanisme des exceptions établit une forme de communication entre les fonctions « profondes » d'une bibliothèque et les fonctions « hautes » qui les utilisent. Les éléments clés en sont les suivants :

- la fonction qui détecte un événement exceptionnel construit une exception et la « lance » (**`throw`**) « vers » la fonction qui l'a appelée ;
- l'exception est un nombre, une chaîne ou, mieux, un objet d'une classe spécialement définie dans ce but, souvent une classe dérivée de la classe **`exception`**, composée des diverses informations utiles à la caractérisation de l'événement en question ;
- une fois lancée, l'exception traverse la fonction qui l'a lancée, sa fonction appelante, la fonction appelante de la fonction appelante, etc., jusqu'à atteindre une fonction active (c.-à-d. une fonction commencée et non encore terminée) qui a prévu « d'attraper » (**`catch`**) ce type d'exception ;
- lors du lancement d'une exception, la fonction qui l'a lancée et les fonctions que l'exception traverse sont immédiatement terminées ; malgré son caractère prématuré, cette terminaison prend le temps de détruire les objets locaux de chacune des fonctions ainsi avortées ;
- si une exception arrive à traverser toutes les fonctions actives (c'est-à-dire si aucune de ces fonctions n'a prévu de l'attraper) alors elle produit la terminaison du programme.

Une fonction indique qu'elle s'intéresse aux exceptions qui peuvent survenir durant l'exécution d'une certaine suite d'instructions par une expression qui d'une part identifie cette suite et qui d'autre part associe un traitement spécifique à chacun des types d'exceptions que la fonction intercepte. Cela prend la forme d'un bloc **`try`** suivi d'un groupe de gestionnaires d'interception ou « *gestionnaires* **`catch`** » :

```
try
    {
    instructions susceptibles de provoquer, soit directement soit dans des
    fonctions appelées, le lancement d'une exception
    }
catch(déclarationParamètre1)
    {
    instructions pour traiter toute exception correspondant au type de paramètre1
    }
```

```
catch(déclarationParamètre2)
    {
    instructions pour traiter toute exception, non attrapée par le gestionnaire
    précédent, correspondant au type de paramètre2
    }
    ...
catch(...)
    {
    instructions pour traiter les exceptions non attrapées par les gestionnaires précédents
    }
```

Un bloc **`try`** doit être immédiatement suivi par au moins un gestionnaire **`catch`**. Un gestionnaire **`catch`** ne peut se trouver qu'immédiatement après un bloc **`try`** ou immédiatement après un autre gestionnaire **`catch`**.

Un programme lance une exception par l'instruction

```
throw expression;
```

La valeur de *expression* est supposée décrire l'exception produite. Elle est propagée aux fonctions actives, jusqu'à trouver un gestionnaire **`catch`** dont le paramètre indique un type compatible avec celui de l'expression. Les fonctions sont explorées de la plus récemment appelée vers la plus anciennement appelée (c.-à-d. du sommet vers la base de la pile d'exécution) ; à l'intérieur d'une fonction, les gestionnaires **`catch`** sont explorés dans l'ordre où ils sont écrits. Le gestionnaire

```
catch(...)
```

attrape *toutes* les exceptions. Il n'est donc pas toujours présent et, quand il l'est, il est le dernier de son groupe.

Dès l'entrée dans le gestionnaire l'exception est considérée traitée (il n'y a plus de « balle en l'air »). A la fin de l'exécution d'un gestionnaire, le contrôle est passé à l'instruction qui suit le dernier gestionnaire de son groupe. Les instructions restant à exécuter dans la fonction qui a lancé l'exception et dans les fonctions entre celle-là et celle qui contient le gestionnaire qui a attrapé l'exception sont abandonnées ; les destructeurs des objets locaux de tous ces blocs sont appelés. Exemple simple :

```
struct PileInt
    {
    int *pile, sommet, taille;
    };
void empiler(PileInt &p, int x)
    {
    if (p.sommet > p.taille)
        throw "empilement dans une pile pleine";
    p.pile[p.sommet++] = x;
    }
int depiler(PileInt &p)
    {
    if (p.sommet <= 0)
        throw "dépilement depuis une pile vide";
    return p.pile[--p.sommet];
    }
```



```
void main()
    {
    PileInt p;
    ...
    try
        {
        while (n-- > 0)
            depiler(p, x);
        }
    catch(char *message)
        {
        cout << message << '\n';
        exit(1);
        }
    }
```

6.1.2 Sélection du gestionnaire d'interception

Le gestionnaire sélectionné lorsqu'une exception est lancée est le premier gestionnaire dont le type est compatible avec l'expression, rencontré en descendant la pile des appels de fonctions,.

Un gestionnaire de type T_1, **`const`** T_1, T_1**`&`** ou **`const`** T_1**`&`** est compatible avec une expression de type T_2 si et seulement si :

- T_1 et T_2 sont égaux, ou
- T_1 est une classe de base accessible de T_2, ou
- T_1 et T_2 sont des pointeurs et T_2 peut être converti en T_1 par une conversion standard.

Notez que les conversions standard sur des types qui ne sont pas des pointeurs ne sont pas tentées.

Si aucun gestionnaire adéquat n'est trouvé, ou si une exception est lancée en dehors de tout bloc **`try`**, l'exception termine sa carrière en appelant la fonction **`terminate()`** qui achève le programme.

A l'intérieur d'un gestionnaire **`catch`** on peut trouver une instruction

```
throw;
```

elle indique que l'exception, qui était tenue pour traitée en entrant dans le gestionnaire, est relancée comme si elle n'avait pas été attrapée. En général, cela veut dire que le gestionnaire en question a commencé le travail d'interception de l'exception, un autre gestionnaire placé au-dessus le terminera.

6.1.3 Déclaration des exceptions susceptibles d'être lancées par une fonction

Une fonction peut indiquer la liste des types des exceptions qui peuvent provenir d'elle :

```
void empiler(PileInt &p, int x) throw(char *)
    {
    ...
    }
```

Plus généralement, d'une fonction dont l'en-tête se présente ainsi

```
type nom(paramètres) throw(T1, T2*)
    ...
```

ne peuvent provenir que des exceptions dont le type est T_1 ou un type ayant T_1 pour classe de base accessible ou T_2 * ou un type pointeur vers une classe ayant T_2 pour classe de base accessible. L'expression

```
type nom(paramètres) throw()
    ...
```

indique que d'une telle fonction ne peut venir aucune exception. De telles déclarations ne font pas partie de la signature : deux fonctions dont les en-têtes ne différent qu'en cela ne sont pas assez différentes pour faire jouer le mécanisme de la surcharge.

Pour les exceptions explicitement lancées par la fonction, cette indication permet une vérification de la part du compilateur, avec l'émission d'éventuels messages de mise en garde.

Lorsqu'une fonction reçoit une exception (venant d'une fonction plus profonde) qu'elle n'attrape pas et qui ne figure pas dans sa liste **`throw(`**T_1**`,`** … T_n**`)`** la fonction prédéfinie **`unexpected()`** est appelée. Par défaut la fonction **`unexpected()`** se réduit à l'appel de la fonction **`terminate()`**.

A propos de **`unexpected`** et **`terminate`**, voir aussi § 6.1.5 et § 8.2.6.

6.1.4 La classe exception

Une exception est une valeur quelconque, d'un type standard ou bien d'un type classe défini à cet effet. Cependant, les exceptions lancées par les fonctions de la bibliothèque standard sont toutes des objets de classes dérivées d'une classe définie spécialement à cet effet, la classe **`exception`**. Cette dernière comporte au minimum les membres suivants : construction par défaut (l'expression **`throw`** construit l'exception), construction par copie (l'expression catch est traitée comme un appel de fonction et commence donc, si besoin, par la copie de l'exception), affectation (la bibliothèque standard affecte parfois les exceptions), destruction et une fonction **`what`** qui renvoie une chaîne de caractères « décrivant » l'exception :

```
class exception
    {
public:
    exception() throw();
    exception(const exception &e) throw();
    exception &operator=(const exception &e) throw();
    virtual ~exception() throw();

    virtual const char *what() const throw();
    };
```

On notera qu'aucune exception ne peut être lancée durant l'exécution d'un membre de la classe exception.

Bien que la question concerne plus la bibliothèque standard que le langage lui-même, signalons les principales classes dérivées directement ou indirectement de **`exception`** :

`exception`

`bad_exception`	Exception lancée par la version par défaut de la fonction **`unexpected`** (cf. § 6.1.3). Signale l'émission par une fonction d'une exception qui n'est pas annoncée dans sa liste **`throw`**.
`bad_cast`	Cette exception signale l'exécution d'une expression **`dynamic_cast`** invalide (cf. § 4.5.1et, pour un exemple, § 4.3.7).
`bad_typeid`	Indique la présence d'un pointeur **`p`** nul dans une expression **`typeid(*p)`** (cf. § 4.5.2).
`logic_error`	Ces exceptions signalent des erreurs provenant de la structure logique interne du programme ; en théorie, on aurait pu les prévoir en lisant attentivement le texte du programme :
`domain_error`	erreur de domaine,
`invalid_argument`	argument invalide,
`length_error`	tentative de création d'un objet de taille supérieure à une certaine « taille maximum autorisée » (le sens précis dépend du contexte),
`out_of_range`	arguments en dehors de ses limites.
`bad_alloc`	Cette exception correspond à l'échec d'une allocation de mémoire.
`runtime_error`	Exceptions signalant des erreurs, autres que des erreurs d'allocation de mémoire, qui ne peuvent être détectées que durant l'exécution du programme :
`range_error`	erreur de rang,

overflow_error	débordement arithmétique (par le haut),
underflow_error	débordement arithmétique (par le bas).

REMARQUE. Les classes précédentes sont des *emplacements réservés* pour des exceptions que les fonctions de la bibliothèque standard (actuelle ou future) peuvent éventuellement lancer. Rien ne dit que c'est actuellement le cas ; en particulier, on a bien du mal à construire des exemples faisant intervenir des exceptions **logic_error** ou **runtime_error**.

6.1.5 Les fonctions *terminate* et *unexpected*

Le comportement des fonctions **terminate** et **unexpected** peut être modifié par le programmeur, à l'aide des fonctions prédéfinies **set_terminate** et **set_unexpected** :

```
typedef void (*handler)();

handler set_terminate(handler);
handler set_unexpected(handler);
```

Exemple :

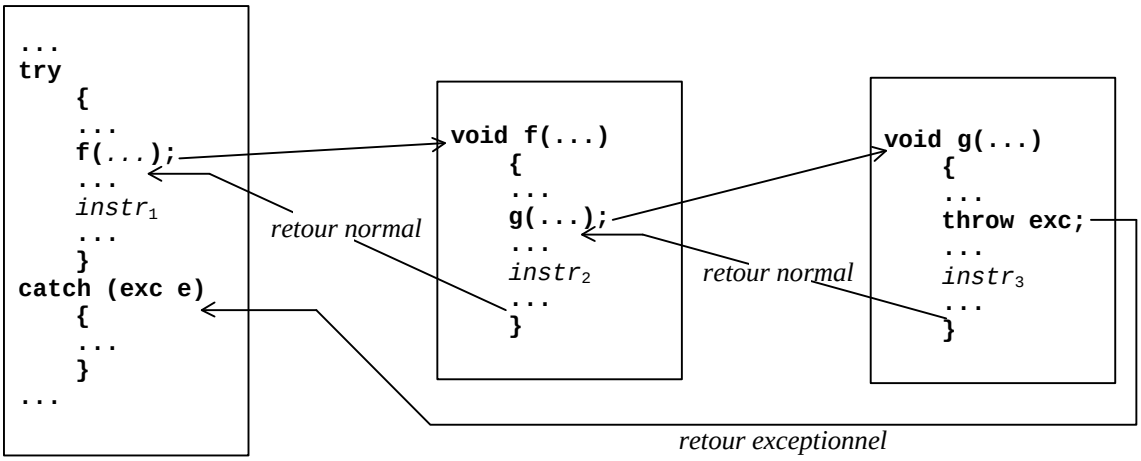
```
void exceptionResiduelle()
    {
    MessageBox("Quelque chose de très impossible s'est un peu produit",
        "PROGRAMME ARRETE");
    exit(0xFF);
    }

unTraitement()
    {
    handler previous = set_terminate(exceptionResiduelle);
    ...
    set_terminate(previous);
    }
```

La fonction installée pour **terminate** doit terminer le programme ; elle ne doit pas retourner à l'appelant ni lancer une exception. Celle installée pour **unexpected** ne doit pas retourner à l'appelant mais peut lancer une exception.

A propos de **unexpected** et **terminate**, voir aussi § 8.2.6.

6.2 Exceptions et libération des ressources

Lors du lancement d'une exception, les instructions qui, dans l'ordre de l'exécution, auraient été placées entre le lancement d'une exception et son interception sont abandonnées.



Techniquement, cela se traduit par la terminaison immédiate des fonctions qui ont été activées plus récemment que celle qui contient la construction **`try...catch...`** et ne sont pas encore terminées au moment de l'évaluation du **`throw`** : la pile d'exécution est brutalement décapitée, perdant les espaces locaux de toutes ces fonctions plus récentes..

C++ garantit cependant que les destructeurs des objets locaux de ces fonctions seront appelés. Cela est un service important qu'il faut penser à utiliser, notamment pour libérer proprement des ressources qui sans cela resteraient indûment allouées.

Exemple (de ce qu'il ne faut pas faire) :

```
void traitement() throw(certaines exceptions)
    {
    Objet *ptr = new Objet();

    Instructions pouvant provoquer le lancement d'une exception

    delete ptr;
    }
```

Si une exception est lancée durant l'exécution de la fonction **`traitement`** l'objet alloué au début de cette fonction ne sera pas libéré car il est représenté par un pointeur (les pointeurs ne sont pas des objets).

Première solution, pénible, prévoir la chose :

```
void traitement() throw(certaines exceptions)
    {
    Objet *ptr = new Objet();

    try
        {
        Instructions pouvant provoquer le lancement d'une exception
        }

    catch(...)
        {
        delete ptr;
        throw;
        }
    delete ptr;
    }
```

6.2.1 Allocateurs

Une solution plus fine consiste à faire en sorte que les ressources en question soient membres d'objets locaux. La destruction de ces derniers, normale ou à l'occasion du lancement d'une exception, garantit alors la libération des ressources. Cela peut se faire élégamment avec un modèle de classe :

```
template<class T>
    class Allocateur
        {
        T *adresse;
                                              // ces deux membres sont
        Allocateur(const Allocateur&) {}      // privés: un allocateur
        void operator=(const Allocateur&) {}  // ne se copie pas !
    public:
        Allocateur(size_t n = 1)
            { adresse = new T[n]; }
        ~Allocateur()
            { delete [] adresse; }
        operator T *() const                  // la conversion
            { return adresse; }               // allocateur -> pointeur
        };
```

```
void traitement() throw(certaines exceptions)
    {
    Allocateur<Objet> zone;
    Objet *ptr = zone;
    Instructions pouvant provoquer le lancement d'une exception
    Pas de delete dans cette fonction (dans laquelle il n'y a pas non plus de new apparent)
    }
```

Maintenant la terminaison, normale ou exceptionnelle, de **traitement** provoquera la destruction de ses objets locaux, dont **zone** ; ainsi, l'espace dynamiquement alloué pointé par **ptr** sera libéré.

Un tel allocateur s'utilise aussi simplement dans le cas d'un tableau :

```
void traitement(int nbr) throw(certaines exceptions)
    {
    Allocateur<Objet> zone(nbr);
    Objet *tab = zone;
    Instructions pouvant provoquer le lancement d'une exception ; par exemple :
    for (int i = 0; i < nbr; i++)
        tab[i] = Objet(...);
    Pas de delete dans le cas d'un tableau non plus.
    }
```

Une autre application de cette même technique est la libération des ressources d'objets à moitié construits (c'est-à-dire d'objets dont la construction n'a pas abouti) :

```
class Machin
    {
    ObjetA *tabA;
    ObjetB *tabB;
public:
    Machin(int na, int nb)
        {
        tabA = new ObjetA[na];
        tabB = new ObjetB[nb];
        autres initialisations
        }
    ~Machin()
        {
        delete [] tabB;
        delete [] tabA;
        }
    };
```

Le destructeur de la classe **Machin** libère bien les ressources allouées à un objet mais si une exception est lancée pendant la construction d'un **Machin** (par exemple, pour signaler un échec du second **new**) le destructeur ne sera pas appelé, car un objet n'a un type défini que lorsque son constructeur a été complètement exécuté.

En revanche, la destruction des objets membres et des sous objets hérités déjà construits est assurée lorsqu'une exception est lancée durant la construction d'un objet. D'où une meilleure manière d'écrire notre classe

```
class Machin
    {
    Allocateur<ObjetA> tabA;
    Allocateur<ObjetB> tabB;
public:
    Machin(int na, int nb)
            : tabA(na), tabB(nb)
        { autres initialisations éventuelles }
    ~Machin()
        { }     // rien à écrire: les objets membres tabA et tabB
                // se chargent de la libération de l'espace dynamique
    };
```

REMARQUE 1. On notera que si **`unMachin`** est un objet de la classe **`Machin`**, la notation **`unMachin.tabA[i]`** reste correcte (au droit d'accès près) et désigne bien un **`ObjetA`**. En effet :

`unMachin.tabA[i] =`

compte tenu du fait que l'opérateur d'indexation n'est pas surchargé pour la classe **`Machin`**, se lit :

`= *((`*`un type pointeur`**`) unMachin.tabA + i) =`**

ce qui, puisqu'une seule conversion vers un type pointeur est définie dans **`Allocateur`**, se lit

`= *((T *) unMachin.tabA + i) =`

défini, dans le modèle **`Allocateur`**, comme

`= *(unMachin.tabA.adresse + i) =`

soit finalement

`= unMachin.tabA.adresse[i]`

REMARQUE 2. D'autres manières d'obtenir une gestion efficace des ressources pointées et notammen d'en garantir la libération sont les *auto-pointeurs* (cf. § 9.1.3) et les pointeurs intelligents, étudiés en exercice.

☺☺☹

Chapitre 7. Les espaces de noms

Dernier élément apporté au langage C++, le but des *espaces de noms* est d'éviter les conflits entre identificateurs globaux qui peuvent apparaître lorsqu'on utilise ensemble des bibliothèques de classes et fonctions développées séparément.

7.1 Visibilité des identificateurs

Identificateurs, portées, régions déclaratives

Déclarer un *identificateur* c'est annoncer l'existence d'un couple (*identificateur*, *entité*) associant un identificateur à une entité d'un programme (une constante, un type, une variable, une fonction, etc.), le premier devenant par là le moyen de faire référence à la seconde.

Chaque couple (*identificateur*, *entité*) possède une *portée*, qui est la plus grande partie du programme dans laquelle l'identificateur peut être utilisé, seul, pour nommer l'entité correspondante. Par « seul » on veut dire ici que l'identificateur ne fait pas l'objet d'une qualification particulière, c'est-à-dire qu'il n'apparaît pas immédiatement après **.** (l'opérateur « membre d'objet »), **->** (l'opérateur « membre d'objet pointé ») ou **::** (l'opérateur de résolution de portée).

Chaque couple (*identificateur*, *entité*) est attaché à une certaine région du texte du programme, appelée une *région déclarative*, qui correspond à une construction syntaxique du langage (un fichier, une classe, une fonction, un bloc **{...}**, etc.).

La région déclarative *R* d'un couple[4] (*identificateur*, *entité*) contient toujours sa portée ; en général elle est strictement plus grande, car la portée est

- la portion du texte du programme qui s'étend depuis la fin de la déclaration jusqu'à la fin de *R*…
- …privée des éventuelles portées de couples (*identificateur*, *entité'*), comportant le même identificateur, déclarés dans des régions déclaratives incluses dans *R*. On dit alors que la déclaration attachée à la région déclarative englobée *masque*, dans sa portée, la déclaration attachée à la région déclarative englobante.

Résolution de portée

Diverses constructions permettent de faire référence à l'entité d'un couple (*identificateur*, *entité*) depuis un point extérieur à sa portée :

- lorsque *entité* est un membre d'une certaine *Classe*, l'association avec un objet de cette classe :

```
Classe obj;
           obj.identificateur
```

- lorsque *entité* est un membre d'une *Classe*, l'association avec l'adresse d'un objet de cette classe :

```
Classe *ptr;
           ptr->identificateur
```

[4] On dit souvent « la portée d'un identificateur ». C'est un abus de langage que nous tâcherons d'éviter ici, car il rend le discours extrêmement confus.

- lorsque *entité* est un membre d'une classe, ou d'une union, ou d'une énumération, *Type*, l'utilisation de l'opérateur de résolution de portée :

  Type`::`*identificateur*

Rappelons qu'en toute circonstance, l'expression

`::`*identificateur*

désigne un identificateur global.

REMARQUE. Préfixer un nom de classe par le mot réservé **`class`** permet aussi d'y accéder quand son nom est masqué par autre chose qu'un nom de classe :

```
class C { ... }

void test()
    {
    int C;

    C x1;               // ERREUR : C n'est pas un type
    class C x2;         // Oui
    ::C x3;             // Oui

    ...
    }
```

7.2 Espaces de noms

7.2.1 Notion et syntaxe

Un *espace de noms* est une région déclarative qui prend la forme suivante :

```
namespace identificateur
    {
    ...
    }
```

On dit que les identificateurs déclarés dans un espace de noms en sont les *membres*. A l'intérieur d'un espace de noms, un membre est visible depuis sa déclaration jusqu'à la fin de l'espace, sauf dans les éventuelles portées de déclarations du même identificateur faites dans des régions déclaratives englobées.

Un espace de noms doit être défini :

- au niveau global, ou bien
- immédiatement imbriqué dans un autre espace de noms.

Sans qualification, un membre d'un espace de noms n'est pas visible en dehors de l'espace. Pour accéder à l'entité associée à un tel identificateur, il faut utiliser l'opérateur de résolution de portée :

nom_espace_de_noms`::`*identificateur*

Bien entendu, il faut que *nom_espace_de_noms* soit visible à l'endroit où l'expression précédente ; si ce n'est pas le cas, par exemple parce que cet identificateur est à son tour membre d'un espace de noms, alors il faudra utiliser pour lui aussi l'opérateur de résolution de portée.

Exemple :

```
namespace nombres
    {
    double zero = 0;
```

```
namespace complexes
    {
    struct Complexe
        {
        double re, im;
        autres membres de la classe Complexe
        };

    Complexe zero;

    void initialisation()
        {
        zero.re = nombres::zero;
        zero.im = nombres::zero;
        }

    autres membres de l'espace de noms complexes
    }

void validation()
    {
    if (zero != complexes::zero.re || zero != complexes::zero.im)
        throw "complexes::zero a été corrompu";
    }
}

void main()
    {
//  cout << zero << "\n";              // ERREUR: zero inconnu ici
    cout << nombres::zero << "\n";
//  cout << complexes::zero << "\n";   // ERREUR: complexes inconnu ici
    cout << nombres::complexes::zero << "\n";
    ...
    }
```

Un membre d'un espace de noms peut être défini à l'extérieur de l'espace : le corps de la définition est alors considéré comme s'il était place à l'intérieur de l'espace de noms :

```
namespace nombres
    {
    double zero = 0;
    ...
    void validation();
    }

void nombres::validation()
    {
    if (zero != complexes::zero.first || zero != complexes::zero.second)
        throw "complexes::zero a été corrompu";
    }
```

Un espace de noms est défini de manière incrémentale. L'espace final est la concaténation des définitions partielles portant le même nom, définies dans une même région déclarative. Exemple :

```
namespace nombres
    {
    double zero = 0, un = 1;
```

```
    namespace complexes
        {
        struct Complexe
            {
            double re, im;
            ...
            };
        Complexe zero, i;
        void initialisation();
        }
    }
namespace nombres
    {
    void complexes::initialisation()
        {
        zero.re = zero.im = i.re = nombres::zero;
        i.im = un;
        }
    void validation();
    }
void nombres::validation()
    {
    if (zero != complexes::zero.re || zero != complexes::zero.im)
        throw "complexes::zero a été corrompu";
    }
```

Un identificateur étant écrit en utilisant l'opérateur de résolution de portée,

- s'il commence par « **::** » la résolution se fait dans la région déclarative globale,
- sinon, la résolution se fait en remontant à partir de la région déclarative courante.

Exemple : une fois corrigé de son erreur, le programme suivant écrit « 1 2 3 4 5 5 6 7 » :

```
char i[1], j[2];
namespace A
    {
    char j[3], k[4];
    namespace B
        {
        char k[5], l[6];
        namespace C
            {
            char l[7];
            void k()
                {
                cout << sizeof i << ' ';
                cout << sizeof ::j << ' ';
                cout << sizeof j << ' ';
                cout << sizeof A::k << ' ';
                cout << sizeof B::k << ' ';
                cout << sizeof A::B::k << ' ';
                cout << sizeof A::l << ' ';   ERREUR: l n'est pas dans A
                cout << sizeof B::l << ' ';
                cout << sizeof l << ' ';
                }
            }
        }
    }
```

```
void main()
    {
    A::B::C::k();
    }
```

7.2.2 Déclaration et directive *using*

La déclaration

```
using région::identificateur;
```

introduit l'*identificateur* indiqué, membre de la *région* indiquée, dans la région déclarative courante, où il pourra être utilisé sans qualification particulière.

Hormis le cas de la surcharge des fonctions, l'identificateur introduit par une déclaration **using** ne doit pas être déjà défini, ni défini ultérieurement, dans la région courante, dans laquelle il masque alors tout identificateur déclaré dans une région englobante.

La directive

```
using namespace espace_de_noms;
```

introduit *tous* les identificateurs de l'espace de noms indiqué dans la région courante, où ils pourront être utilisés sans qualification particulière. Hormis le cas de la surcharge des fonctions, si certains de ces identificateurs étaient déjà définis dans la région courante, alors il se produit des ambiguïtés que le compilateur refusera.

Cette directive est transitive : les identificateurs qui sont dans *espace_de_noms* par suite d'une autre directive **using** sont introduits comme les autres.

Si un identificateur introduit est un nom de fonction et qu'une fonction de même nom est déjà défini dans la région courante, ou bien est défini plus loin dans la région courante, alors le mécanisme de la surcharge joue :

```
void f(int);

namespace A
    {
    void f(char);
    }

using namespace A;

void main()
    {
    f(1);       // appel de f(int)
    f('A');     // appel de A::f(char)
    }
```

Cette possibilité est regrettable. Le but de **namespace** est précisément d'éviter les surcharges involontaires. Or, la possibilité d'introduire la totalité des identificateurs d'un espace de noms, éventuellement volumineux et confus, réinstalle le problème qu'il s'agissait de corriger. La « individuelle » de **using**, qui introduit les identificateurs un par un, est préférable. Mais tellement pénible !

7.2.3 Alias

La directive

```
namespace alias = espace_de_noms;
```

introduit un nouveau nom pour un espace de noms donné. Cela permet par exemple au programmeur de donner des noms courts à des espaces ayant des noms à rallonges :

```
namespace OWL = BorlandObjectsWindowsCompatibleLibrairies;

void main()
    {
    OWL::main();
    }
```

ou de paramétrer l'espace de noms qui est effectivement utilisé dans une compilation :

```
namespace window =
#ifdef __BORLANDC__
    OWL
#else
    MFC
#endif
;
```

7.2.4 Espaces anonymes

Un espace anonyme est défini de la manière suivante :

```
namespace
    {
    déclarations
    }
```

L'effet de cette déclaration est le même que si on avant écrit le couple

```
namespace nom_spécifique
    {
    déclarations
    }

using nom_spécifique;
```

où *nom_spécifique* est un identificateur *unique pour chaque unité de compilation*. Les identificateurs déclarés dans un espace anonyme sont donc globaux, mais ne peuvent pas être utilisés depuis une autre unité de compilation. Par exemple la déclaration

```
namespace
    {
    int n;
    void f();
    }
```

équivaut à

```
static int n;
static void f();
```

L'utilisation d'espaces anonymes est préférable à celle du qualifieur **`static`**, qui a d'autres significations dans d'autres contextes et dont l'utilisation comme dans l'exemple ci-dessus est appelée à devenir obsolète.

☺☺☺

Chapitre 8. La bibliothèque standard (début)

8.1 Structure générale

On appelle bibliothèque standard de C++ un ensemble de bibliothèques, maintenant bien défini par la norme ISO, intégrant notamment deux illustres ancêtres : la *IO Stream Library* (bibliothèque de flux d'entrée-sortie) et la *STL* ou *Standard Template Library* (bibliothèque standard de modèles). Au total, la bibliothèque standard de *C++* se compose des dix *catégories* suivantes :

- support du langage (§ 8.2),
- diagnostics (§ 8.3),
- utilitaires généraux (§ 9.1),
- chaînes de caractères (§ 8.4),
- particularités locales (§ 8.5),
- composants numériques (§ 8.6),
- flux d'entrée sortie (§ 8.7),
- conteneurs (§ 9.2),
- itérateurs (§ 9.3),
- algorithmes (§ 9.4).

Les déclarations requises pour utiliser la bibliothèque standard C++ se trouvent dans 32 fichiers en-tête :

<algorithm>	**<bitset>**	**<complex>**	**<deque>**	**<exception>**
<fstream>	**<functional>**	**<iomanip>**	**<ios>**	**<iosfwd>**
<iostream>	**<istream>**	**<iterator>**	**<limits>**	**<list>**
<locale>	**<map>**	**<memory>**	**<new>**	**<numeric>**
<ostream>	**<queue>**	**<set>**	**<sstream>**	**<stack>**
<stdexcept>	**<streambuf>**	**<string>**	**<typeinfo>**	**<utility>**
<valarray>	**<vector>**			

Les noms ci-dessus peuvent ne pas être des noms de fichier corrects, ou ne pas nommer des fichiers existants sur votre système ; la manière de les transformer en de vrais noms de fichier dépend de l'implémentation.

Les noms de *tous les éléments de la bibliothèque standard*, sauf les macros et les opérateurs **`new`** et **`delete`**, sont membres de l'espace de noms **`std`** ou d'espaces de noms imbriqués dans **`std`**.

Dans les petits programmes, lorsque le risque de collisions de noms est minime, on insère dans l'espace global les identificateurs de la bibliothèque standard, en faisant suivre les directives d'inclusion des fichiers en-tête d'une directive **`using namespace`** comme ceci :

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

Les éléments de la bibliothèque standard de C appartiennent aussi à la bibliothèque standard de C++ à travers les fichiers en-tête suivants, qui renvoient de manière évidente aux fichiers correspondants de la bibliothèque C :

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

Les fichiers en-tête de la bibliothèque C (**assert.h**, **ctype.h**, etc.) peuvent être utilisés aussi en C++. Notez que les éléments définis dans les fichiers « **c***N* » appartiennent à l'espace de noms **std**, tandis que ceux définis dans les fichiers « *N***.h** » appartiennent à l'espace de noms global.

8.2 Support du langage

Cette bibliothèque apporte des éléments requis par la définition du langage C++ lui-même. Elle est composée des fichiers en-tête suivants :

<exception>	**<limits>**	**<new>**	**<typeinfo>**	
<cfloat>	**<climits>**	**<csetjmp>**	**<csignal>**	**<cstdarg>**
<cstddef>	**<cstdlib>**	**<cstdlib>**	**<ctime>**	

8.2.1 Types dépendant de l'implémentation

<cstddef>

Ce fichier est pratiquement le même que **stddef.h**. Il définit notamment :

- **size_t**, le type entier (parmi **int**, **unsigned int**, **long**, etc.) le mieux adapté à l'expression de la taille des objets ; c'est le type du résultat de l'opérateur **sizeof**, de l'argument de la fonction **malloc**, etc.
- **ptrdiff_t**, le type entier le mieux adapté à l'expression du résultat de la soustraction de deux pointeurs.

Notez qu'en C++, contrairement à C, ce fichier ne contient pas la définition du type **wchar_t** : en C++ cet identificateur est un mot réservé, non un type défini par une formule *typedef*.

8.2.2 Propriétés de l'implémentation

<limits> **<climits>** **<cfloat>**

Les fichiers **<climits>** et **<cfloat>** sont analogues à leurs correspondants de la bibliothèque de C (**limits.h** et **float.h**). Ils introduisent un ensemble de macros exprimant les bornes des types numériques : **INT_MIN** (plus petite valeur du type **int**), **INT_MAX** (plus grande valeur du type **int**), **UINT_MAX** (plus grande valeur du type **unsigned int**), etc.

Le fichier **<limits>** remplit le même rôle, mais davantage dans un « esprit C++ ». Il définit le modèle de classe **numeric_limits** dont les membres, tous statiques, correspondent aux diverses questions que l'on peut se poser à propos d'un type numérique :

```
template<class T>
class numeric_limits
    {
public:
            // la plus petite valeur (finie) du type
    static T min() throw();

            // la plus grande valeur (finie) du type
    static T max() throw();

            // nombre de chiffres (dans la base de la représentation interne) qui peuvent être
            // représentés sans erreur (pour un flottant : nombre de chiffres de la mantisse)
    static const int digits = 0;
```



```
        // nombre de chiffres, en base 10,  qui peuvent être représentés sans erreur
    static const int digits10 = 0;

        // le type comprend-t-il des valeurs positives et négatives ?
    static const bool is_signed = false;

        // le type définit-il un sous-ensemble de Z ?
    static const bool is_integer = false;

        // la représentation interne des valeurs est-elle exacte
        // (exemple :  entiers, rationnels, nombres à virgule fixe, etc.)
    static const bool is_exact = false;

        // types entiers: la base de la représentation interne
        // flottants: la base de la représentation de l'exposant
    static const int radix = 0;

        // la différence entre 1 et la plus petite valeur représentable supérieure à 1
    static T epsilon() throw();

        // la plus grande erreur d'arrondi possible
    static T round_error() throw();

        // cela doit être le plus petit exposant, mais  la norme dit « minimum negative
        // integer such that radix raised to the power of one less than that integer is a
        // normalized floating point number » Si cela vous branche…
    static const int min_exponent = 0;

        // comme le précédent, avec 10 à la place de radix
    static const int min_exponent10 = 0;

        // le plus grand entier e tel que radix^e est représentable
        // (radix est la base de la représentation interne)
    static const int max_exponent = 0;

        // le plus grand entier e tel que 10^e est représentable
    static const int max_exponent10 = 0;

        // le type a-t-il le moyen de représenter l'infini ?
    static const bool has_infinity = false;

        // le type a-t-il une représentation de NaN
        // (« Not a Number », un résultat qui n'est pas un nombre) ?
    static const bool has_signaling_NaN = false;

    etc.
    };
```

On trouve ensuite une collection de spécialisations de ce modèle, correspondant aux divers types employés :

```
class numeric_limits<char>;
class numeric_limits<signed char>;
class numeric_limits<unsigned char>;
class numeric_limits<wchar_t>;
class numeric_limits<short>;
class numeric_limits<int>;
class numeric_limits<long>;
class numeric_limits<unsigned short>;
class numeric_limits<unsigned int>;
class numeric_limits<unsigned long>;
class numeric_limits<float>;
class numeric_limits<double>;
class numeric_limits<long double>;
```

Par exemple, pour connaître le nombre de chiffres significatifs d'un **`float`** et d'un **`double`** il suffit d'évaluer l'expression :

```
cout << "float:  " << numeric_limits<float>::digits10  << '\n'
     << "double: " << numeric_limits<double>::digits10 << '\n';
```

8.2.3 Démarrage et terminaison des programmes

```
<cstdlib>
```

Pour ce qui nous occupe ici, ce fichier apporte trois fonctions (**exit**, **atexit** et **abort**) et deux macros (**EXIT_FAILURE** et **EXIT_SUCCESS**) :

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

Cette fonction provoque la terminaison du programme en cours. Les objets statiques (exemple : les valeurs des variables globales) sont proprement détruits, dans l'ordre inverse de leur création ; les objets automatiques (valeurs de variables locales) ne sont pas détruits. Les fonctions enregistrées avec **atexit** sont exécutées.

L'argument **status** représente une valeur conventionnelle qui est passée au système d'exploitation pour l'informer sur la réussite ou l'échec du programme. Pour être tout à fait portable, sa valeur doit être une des valeurs **EXIT_SUCCESS** (le programme a « réussi » dans l'accomplissement de sa tâche) ou **EXIT_FAILURE** (le programme n'a pas réussi à faire ce qu'on attendait de lui).

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

Cette fonction prend l'adresse **func** d'une fonction sans argument ni résultat et l'ajoute à une liste de fonctions qui seront appelées, dans l'ordre inverse de leur enregistrement, lorsque le programme se terminera normalement ou par un appel de **exit**.

```
void abort();
```

Cette fonction provoque la terminaison du programme en cours. Les objets existants (automatiques ou statiques) ne sont pas détruits et les fonctions enregistrées par **atexit** ne sont pas appelées.

8.2.4 Gestion dynamique de la mémoire

```
<new>
```

Fondamentalement ce fichier déclare un certain nombre de surcharges des opérateurs **new** et **delete** (cf. § 1.10 et § 3.2.2) et les déclarations suivantes :

```
class bad_alloc;
```

Sous-classe de **exception**, les objets de la classe **bad_alloc** sont créés et lancés (par la fonction qui est le « new handler courant », voir ci-après) lorsque l'allocation dynamique de mémoire échoue.

```
typedef void (*new_handler)();

new_handler set_new_handler(new_handler new_p) throw();
```

Cette fonction enregistre la fonction **new_p** comme étant le « new handler » courant. On appelle ainsi la fonction qui est appelée lorsque l'allocation dynamique de mémoire (c.-à-d., la surcharge de l'opérateur **new**) échoue.

Un « new handler » courant peut

- augmenter la mémoire disponible et retourner normalement (**new** tentera alors à nouveau de faire l'allocation, cf. § 1.10.4),
- lancer une exception de type **bad_alloc** ou une classe dérivée de **bad_alloc**,
- appeler une des fonctions **exit** ou **abort**.

Un « new handler » est installé par défaut, il se limite à lancer une exception de type **bad_alloc**.

8.2.5 Identification des types

```
<typeinfo>
```

Ce fichier est nécessaire pour pouvoir utiliser les opérateurs **dynamic_cast<type>** (cf. § 4.5.1) et **typeid(expression)** (cf. § 4.5.2). Il déclare la classe


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

ainsi que les deux classes (sous-classes de **exception**) qui caractérisent les échecs des opérateurs précédents :

```
class bad_cast;
class bad_typeid;
```

8.2.6 Manipulation des exceptions

<exception>

Ce fichier déclare les éléments les plus généraux en rapport avec les exceptions (cf. § 6.1.4), c'est-à-dire l'ancêtre commun de toutes les exceptions lancées par des fonctions de la bibliothèque standard :

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

et trois éléments destinés à gérer l'impossibilité d'attraper une exception lancée : les fonctions **unexpected** et **terminate**, et la classe **bad_exception** :

```
class bad_exception;

typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();
void unexpected();

typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler f) throw();
void terminate();
```

La fonction **unexpected** est appelée lorsque depuis une fonction *f*, comportant une « liste **throw** » T_f, a été lancée une exception qui n'est pas déclarée dans T_f.

La version par défaut de la fonction **unexpected** se réduit à appeler la fonction **terminate**, voir ci-après.

La fonction **set_unexpected** permet d'installer sa propre version de la fonction **unexpected**. Une telle fonction ne doit pas retourner à l'appelant, mais elle peut lancer une nouvelle exception *e*.

Si *e* est prévue dans la liste T_f qui a posé problème, tout va bien : *e* est traitée comme si elle avait été lancée depuis la fonction *f*. Si *e* n'est pas prévue dans T_f, alors

- si **bad_exception** figure dans T_f, alors *e* est remplacée par un objet **bad_exception** et traité comme si elle avait été lancée depuis *f*,
- si **bad_exception** ne figure pas dans T_f, alors la fonction **terminate** est appelée.

La fonction **`terminate`** est appelée dans les situations où la manipulation des exceptions doit être remplacée par une gestion d'erreur moins subtile. Il en est ainsi :

- lorsque le mécanisme de manipulation des exceptions ne trouve pas un gestionnaire d'interception pour une exception lancée,
- lorsqu'une expression **`throw`** (sans opérande) est évaluée en dehors d'un gestionnaire d'interception (c'est-à-dire, lorsqu'on tente de « relancer l'exception » alors qu'aucune exception n'est lancée),
- lorsque, après qu'une exception ait été lancée mais avant qu'elle n'ait été attrapée, une fonction appelée (par exemple, pendant la destruction d'un objet local) lance une deuxième exception non attrapée,
- lorsque la construction ou la destruction d'un objet *statique* se termine en lançant une exception,
- lorsque la destruction d'un objet *local* se termine en lançant une exception,
- lorsque l'exécution d'une fonction enregistrée avec **`atexit`** se termine en lançant une exception,
- lorsque la fonction **`unexpected`** (appelée parce qu'une certaine exception ne figurait pas dans la liste **`throw`** T_f d'une certaine fonction *f*) tente de lancer une exception qui n'est pas prévue dans T_f, et que T_f ne mentionne pas **`bad_exception`**.

La fonction **`terminate`** peut également être explicitement appelée par le programme.

La fonction **`set_terminate`** permet d'installer sa propre version de la fonction **`terminate`**. Une telle fonction ne doit pas retourner à l'appelant ni lancer une exception.

8.2.7 Autres fonctions du « runtime »

`<cstdarg>` **`<csetjmp>`** **`<ctime>`** **`<csignal>`**

Ces fichiers fonctionnent de la même manière que leurs homologues de la bibliothèque C. Contentons-nous de donner la liste des éléments qu'ils déclarent :

stdarg

- un type : **`va_list`**
- trois macros : **`va_arg`**, **`va_start`**, **`va_end`**,

setjmp

- un type : **`jmp_buf`**
- une macro : **`int setjmp(jmp_buf env);`**
- une fonction : **`void longjmp(jmp_buf env, int val);`**

time

- un type : **`clock_t`**
- une macro : **`CLOCKS_PER_SEC`**
- une fonction : **`clock_t clock();`**

signal

- un type : **`sig_atomic_t`**
- neuf macros : **`SIG_DFL`**, **`SIG_IGN`**, **`SIG_ERR`**,
  `SIGABRT`, **`SIGILL`**, **`SIGSEGV`**, **`SIGFPE`**, **`SIGINT`**, **`SIGTERM`**
- deux fonctions :
```
void (*signal(int sig, void (*func) (int sig)))(int sig);
int raise(int sig);
```

8.3 Diagnostics

On trouve ici des composants pour signaler les erreurs à l'exécution. Trois fichiers en-tête sont concernés :

```
<stdexcept>
<cassert>
<cerrno>
```

Les classes définies dans **<stdexcept>** ont été expliquées en § 6.1.4. Contentons-nous d'en rappeler la hiérarchie d'héritage :

```
exception
        bad_exception
        bad_cast
        bad_typeid
        logic_error
                domain_error
                invalid_argument
                length_error
                out_of_range
        bad_alloc
        runtime_error
                range_error
                overflow_error
                underflow_error
```

Les fichiers **<cassert>** et **<cerrno>** sont analogues aux fichiers de C **assert.h** et **errno.h**.

8.4 Chaînes de caractères

Fichiers en-tête :

```
<string>
<cstdlib>
<cstring>
<cctype>
<cwtype>
<cwchar>
```

Le fichier **<string>** définit le modèle de classe **char_traits** et, surtout, la classe **string** (voir ci-dessous).

Le modèle **char_traits** déclare un ensemble d'opérations de bas niveau (comparaison, recherche, déplacement, copie, etc.) sur les types pouvant jouer le rôle de caractère dans un objet **string**. Deux spécialisations de ce modèle sont particulièrement utiles : **char_traits<char>** et **char_traits<wchar_t>**.

Les cinq autres fichiers listés ci-dessus définissent, comme leurs homologues de la bibliothèque C, les utilitaires tournant autour des **char** et des **char *** et, plus généralement, des séquences terminées par zéro.

8.4.1 La classe string

La classe **string** est un type simple à utilisaer, sûr, complet et performant pour le traitement des chaînes de caractères, presque toujours très largement préférable au pénible type **char *** de C.

En réalité, **string** n'est qu'un nom pour une classe qui est la spécialisation d'un modèle :

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

Le modèle **basic_string** offre de nombreuses fonctionnalités, communes aux chaînes de **char**, aux chaînes de **wchar_t**, voire aux chaînes d'autres types. Sa déclaration est touffue et difficile à lire ; pour cette raison, nous allons nous intéresser au cas des **char** uniquement.

La classe **`string`** fonctionne *comme si* elle avait été ainsi définie :

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

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

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

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

        // construction avec les caractères fournis par une itération
        // (au sens des itérateurs de la STL)
    template<class inIter> string(inIter begin, inIter end);

        // destruction
    ~string();

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

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

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

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

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

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

        // concaténation avec une chaîne, un char * , un char ou le produit d'une itération
    string &operator+=(const string &str);
    string &operator+=(const char *s);
    string &operator+=(char c);
    string &append(const string &str);
    string &append(const string &str, size_t pos, size_t n);
    string &append(const char *s, size_t n);
    string &append(const char *s);
    string &append(size_t n, char c);
```

```
template<class inIter> string &append(inIter first, inIter last);
void push_back(const char);

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

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

    // suppression de caractères au milieu d'une chaîne
string &erase(size_t pos = 0, size_t n = npos);
itérateur erase(itérateur position);
itérateur erase(itérateur first, itérateur last);

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

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

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

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

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

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

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

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

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

On notera que, contrairement à certains prédécesseurs des **`string`** (comme les **`CString`** de la bibliothèque *MFC*) il n'existe pas de conversion *implicite* d'une valeur de type **`string`** en une valeur **`char *`**. Pour une telle conversion, il faut *explicitement* appeler la fonction membre **`c_str`**.

8.4.2 Exemples (les string, c'est vraiment bien)

1. Avec la classe **`string`** les concaténations de chaînes deviennent sûres et simples. Exemple : la fabrication d'un chemin à partir d'un nom de fichier.

Version avec **`char *`** :

```
main()
    {
    char tmp[80];
    cout << "fichier? ";
    cin >> tmp;

    char nomfic[80];              // en espérant que cela suffit...
    strcpy(nomfic, "/home/henri/projet/donnees/");
    strcat(nomfic, tmp);
    strcat(nomfic, ".dat");

    ifstream fic(nomfic);
    ...
    }
```

Version avec **`string`** :

```
main()
    {
    string tmp;
    cout << "fichier? ";
    cin >> tmp;

    string nomfic = "/home/henri/projet/donnees/" + tmp + ".dat";

    ifstream fic(nomfic.c_str());
    ...
    }
```

2. Avec la classe **string**, les classes contenant des chaînes de caractères deviennent beaucoup plus simples à gérer. Exemple : la classe **Concurrent** est destinée à représenter les participants à un marathon populaire.

Version avec **char *** :

```
class Concurrent
    {
    char *nom;
    int dossard;

    void def_nom(const char *n)
        {
        nom = new char[strlen(n) + 1];
        strcpy(nom, n);
        }
public:
    Concurrent(const char *n, int d)
        { def_nom(n); dossard = d; }
    Concurrent(Concurrent &c)
        { def_nom(c.nom); dossard = c.dossard; }
    Concurrent &operator=(const Concurrent &c)
        {
        if (&c != this)
            { delete [] nom; def_nom(c.nom); }
        dossard = c.dossard;
        return *this;
        }
    ~Concurrent()
        { delete [] nom; }

//  Autres membres…
    };
```

Version avec **string** :

```
class Concurrent
    {
    string nom;
    int dossard;
public:
    Concurrent(const char *n, int d) : nom(n)
        { dossard = d; }

//  les versions par défaut de la création par copie et de l'affectation par défaut conviennent

//  Autres membres...
    };
```

8.5 Localisation

Fichiers en-tête :

```
<locale>
```

On trouve ici des classes pour la prise en compte des particularités de chaque partie du monde. Cela concerne l'alphabet utilisé, l'expression de la date et l'heure, les unités monétaires, les autres mesures, etc.

Pour *donner une idée* du contenu du fichier **<locale>** voici une liste simplifiée des classes qu'il introduit :

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

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

```
template<class charT> bool  isalpha (charT c, const locale &loc);
template<class charT> bool  isdigit (charT c, const locale &loc);
template<class charT> bool  ispunct (charT c, const locale &loc);
template<class charT> bool  isxdigit(charT c, const locale &loc);
template<class charT> bool  isalnum (charT c, const locale &loc);
template<class charT> bool  isgraph (charT c, const locale &loc);
template<class charT> charT toupper (charT c, const locale &loc);
template<class charT> charT tolower (charT c, const locale &loc);

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

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

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

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

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

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

etc.
}
```

8.6 Composants numériques

Fichiers en-tête concernés :

```
<complex>
<valarray>
<numeric>
<cmath>
<cstdlib>
```

8.6.1 Complex

Le fichier **<complex>** déclare un modèle et trois spécialisations prédéfinies (l'utilisateur peut ajouter ses propres spécialisations) :

```
template<class T> class complex;

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

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

8.6.2 Valarray

Le fichier **<valarray>** introduit cinq modèles de classe :

```
template<class T> class valarray;           // un tableau de T

template<class T> class slice_array;        // tranche de tableau
template<class T> class gslice_array;       // tranche généralisée de tableau
template<class T> class mask_array;         // tableau masqué
template<class T> class indirect_array;     // tableau indirect
```

et deux classes :

```
class slice;    // tranche
class gslice;   // tranche généralisée
```

Un objet de la classe **valarray<T>** est un tableau intelligent (*smart array*) mono-dimensionnel dont les éléments sont numérotés à partir de zéro. Ils réalisent la notion mathématique de suite finie.

Le type **T** est censé être numérique. Les opérations numériques (arithmétique, comparaisons, fonctions transcendantes) engendrent des opérations sur des **valarray<T>**, construites par application composante à composante des opérations correspondantes sur des objets du type **T**.

Les **valarray** ont donc deux avantages principaux :

- ils possèdent une importante collection d'opérateurs induits par les opérateurs numériques,
- ils ont une représentation interne efficace, notamment pour ce qui concerne l'allocation de la mémoire (avec utilisation de structures partagées) et le temps requis pour la construction par copie, l'affectation, l'extraction de sous-tableau, etc.

On notera que, les **valarray** étant optimisés relativement à la vitesse de traitement et à l'espace occupé, ils ne le sont pas pour la sécurité et la qualité de la gestion des erreurs. En particulier, *les erreurs sur les indices et les tailles ne sont pas forcément détectées et signalées sur toutes les implémentations*.

Voici des exemples des fonctions disponibles sur les **valarray** (avec les fonctions membres et les fonctions indépendantes, plus de 120 opérations sont déclarées). Exemples de fonctions indépendantes :

```
    // multiplication composante à composante
template<class T> valarray<T> operator*(const valarray<T>&,
                                        const valarray<T>&)
    // conjonction logique composante à composante
template<class T> valarray<bool> operator&&(const valarray<T>&,
                                        const valarray<T>&);
    // valeur absolue composante à composante
template<class T> valarray<T> abs(const valarray<T>&);

    // puissance
template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> pow(const valarray<T>&, const T&);
template<class T> valarray<T> pow(const T&, const valarray<T>&);
```

etc.

Exemples de fonctions membres :

```
template<class T> class valarray
    {
public:
        // construction par défaut (taille 0)
    valarray();
        // construction avec une taille n
    explicit valarray(size_t n);
        // construction et initialisation avec n copies de x
    valarray(const T &x, size_t n);
        // construction et initialisation avec les n premières valeurs de t
    valarray(const T* t, size_t n);

        // accès aux éléments
    T operator[](size_t) const;
    T &operator[](size_t);

        // accès à des  sous-ensembles (voir plus loin)
    valarray<T>         operator[](slice) const;
    slice_array<T>      operator[](slice);
    valarray<T>         operator[](const gslice&) const;
    gslice_array<T>     operator[](const gslice&);
    valarray<T>         operator[](const valarray<bool>&) const;
    mask_array<T>       operator[](const valarray<bool>&);
    valarray<T>         operator[](const valarray<size_t>&) const;
    indirect_array<T>   operator[](const valarray<size_t>&);

        // opérations dissymétriques
    valarray<T> &operator += (const valarray<T>&);
    valarray<T> &operator *= (const valarray<T>&);

    etc.

        // taille
    size_t size() const;

        // « cumuls »
    T sum() const;
    T min() const;
    T max() const;

        // décalages
    valarray<T> shift (int) const;
    valarray<T> cshift(int) const;

        // application d'une fonction
    valarray<T> apply(T func(T)) const;
    valarray<T> apply(T func(const T&)) const;

        // modification de la taille
    void resize(size_t sz, T c = T());

    etc.

    };
```

Tranches, tableaux masqués et tableaux indirects

A. Une *tranche* (*slice*) représente une sous-suite de la suite (0, 1, 2, … *n*-1), de la forme

(*debut*, *debut* + *pas*, *debut* + 2 × *pas*, … *debut* + (*nombre* - 1) × *pas*)

Elle est définie par la donnée des trois quantités *debut*, *nombre* et *pas* :

```
class slice
    {
public:
    slice();
    slice(size_t debut, size_t nombre, size_t pas);
```



```
    size_t start() const;       // debut
    size_t size() const;        // nombre
    size_t stride() const;      // pas
    };
```

Exemple : avec les déclarations

```
const valarray<double> v(100);
slice s(10, 5, 2);
```

l'expression

```
v[s]
```

désigne le sous-tableau (v_{10}, v_{12}, v_{14}, v_{16}, v_{18}). On peut en faire diverses choses ; par exemple, s'en servir pour créer un deuxième **valarray** :

```
valarray<double> w(v[s]);
```

B. Une *tranche généralisée* est définie par un indice de départ, une *suite de tailles* et, en correspondance avec cette dernière, une *suite de pas*. Elle définit un sous-ensemble de (0, 1, 2, … *n*-1) de la forme

$$(d,\ d + p_1,\ \ldots\ d + (n_1 - 1) \times p_1,\ d + (n_1 - 1) \times p_1 + p_2,\ \ldots\ d + (n_1 - 1) \times p_1 + (n_2 - 1) \times p_2,\ \ldots\ )$$

Une tranche généralisée est représentée par un objet de la classe **gslice** :

```
class gslice
    {
public:
    gslice();
    gslice(size_t d, const valarray<size_t> &n, const valarray<size_t> &p);
    size_t start() const;               // debut
    valarray<size_t> size() const;      // suite des tailles
    valarray<size_t> stride() const;    // suite des pas
    };
```

C. Un *tableau masqué W* est un sous-tableau d'un tableau *V* obtenu en indiquant explicitement, par un tableau de booléens en correspondance avec *V*, ceux des éléments de *V* qui font partie de *W*. Les uns par rapport aux autres, ces éléments sont placés dans *W* comme ils le sont dans *V*.

Par exemple, après l'exécution de :

```
valarray<double> v(30);
valarray<bool> b(false, 30);
b[ 2] = b[ 3] = b[ 5] = b[ 7] = b[11] = true;
b[13] = b[17] = b[19] = b[23] = b[29] = true;
```

l'expression

```
v[b]
```

représente le sous-tableau de **v** formé des éléments dont l'indice est un nombre premier, c'est-à-dire la suite (v_2, v_3, v_5, v_7, v_{11}, v_{13}, v_{17}, v_{19}, v_{23}, v_{29}).

D. Un *tableau indirect W* est un sous-tableau d'un tableau *V* obtenu en indiquant explicitement, par un tableau d'indices, ceux des éléments de *V* qui font partie de *W*, et dans quel ordre.

Par exemple, après l'exécution de :

```
valarray<double> v(tab, 30);
valarray<size_t> ind(10);
ind[0] = 21;  ind[1] = 23;  ind[2] = 25;  ind[3] = 27;  ind[4] = 29;
ind[5] =  0;  ind[6] =  2;  ind[7] =  4;  ind[8] =  6;  ind[9] =  8;
```

l'expression

```
v[ind]
```

représente le sous tableau de **v** formé des cinq derniers éléments de **v** d'indice impair, suivis des cinq premiers éléments de **v** d'indice pair, c'est-à-dire la suite (v_{21}, v_{23}, v_{25}, v_{27}, v_{29}, v_0, v_2, v_4, v_6, v_8).

Les modèles slice_array, gslice_array, mask_array, indirect_array

Ces modèles sont déjà apparus dans la déclaration du modèle **`valarray`** :

```
template<class T> class valarray
    {
    ...
    slice_array<T>      operator[](slice);
    gslice_array<T>     operator[](const gslice&);
    mask_array<T>       operator[](const valarray<bool>&);
    indirect_array<T>   operator[](const valarray<size_t>&);
    ...
    };
```

Le modèle **`slice_array`** est un modèle *auxiliaire* utilisé par l'opérateur « d'indexation par tranche ». Il a la sémantique d'une référence à un sous-tableau spécifié par un objet **`slice`**.

Par exemple, **`v`** et **`w`** étant des objets d'une certaine instance **`valarray<T>`**, l'expression

```
v[slice(1, 5, 3)] = w;
```

a pour effet l'affectation des éléments de **`w`** aux éléments d'une tranche de **`v`** spécifiée par **`slice(1, 5, 3)`**. On aurait obtenu le même effet en écrivant les cinq affectations :

```
v[ 1] = w[0];
v[ 4] = w[1];
v[ 7] = w[2];
v[10] = w[3];
v[13] = w[4];
```

Un programme C++ ne peut pas instancier le modèle **`slice_array`**, dont tous les constructeurs sont privés. En fait, il s'agit d'un modèle auxiliaire, dont l'emploi est transparent pour l'utilisateur.

Cette explication vaut, *mutatis mutandis*, pour les modèles **`gslice_array`**, **`mask_array`** et **`indirect_array`**. Ce sont des modèles auxiliaires, transparents pour l'utilisateur, instanciés lors de l'utilisation sur un **`valarray`** *V* de l'opérateur d'indexation avec pour argument une tranche généralisée, un tableau de booléens ou un tableau d'entiers. Les objets des instances de ces modèles ont la sémantique de références à des sous-tableaux de *V*.

8.6.3 Numeric

N.B. Cette section peut être sautée en première lecture. Sa compréhension nécessite de connaître les notions d'opérateur binaire et d'itérateurs d'entrée et de sortie, expliquées au chapitre 9.

Le fichier **`<numeric>`** déclare les modèles de fonctions **`accumulate`**, **`inner_product`**, **`partial_sum`** et **`adjacent_difference`**, qui effectuent des opérations arithmétiques généralisées sur des suites représentées par des itérateurs (cf. § 9.4).

Cumul

```
template<class inIter, class T>
    T accumulate(inIter premier, inIter dernier, T init);

template<class inIter, class T, class binOper>
    T accumulate(inIter premier, inIter dernier, T init, binOper opbin);
```

`inIter` représente un type pouvant jouer le rôle de *input iterator* (cf. § 9.4) ; **`binOper`** un type pouvant jouer le rôle d'opérateur binaire. Cette fonction calcule la valeur **`result`** ainsi définie :

`result` = **`init`**;
pour chaque **`i`** *de* **`premier`** *jusqu'à*[5] **`dernier`** *faire*
    `result` = **`result`** + **`*i`** *// premier cas*
    `result` = **`opbin(result`**, **`*i)`** *// deuxième cas*

Exemple : avec la déclaration

[5] S'agissant d'itérateurs, l'expression « pour chaque *i* de *a* jusqu'à *b* » représente la suite *a*, *a* + 1, …*a* + *k*, où *k* est le plus grand entier tel que $a + k \neq b$.

```
int t[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
```

l'expression

```
cout << accumulate(&t[0], &t[5], 1000);
```

affiche 1030 (résultat de 1000 + 2 + 4 + 6 + 8 + 10).

Produit intérieur

```
template<class inIter1, class inIter2, class T>
    T inner_product(inIter1 prem1, inIter1 der1, inIter2 prem2, T init);

template<class inIt1, class inIt2, class T, class binOp1, class binOp2>
    T inner_product(inIt1 prem1, inIt1 der1,
                inIt2 prem2, T init, binOp1 opbin1, binOp2 opbin2);
```

inIt1 et **inIt2** représentent des types pouvant jouer le rôle de *input iterator* ; **binOp1** et **binOp2** des types pouvant jouer des rôles d'opérateurs binaires. Cette fonction calcule la valeur **result** ainsi définie :

```
result = init;
pour chaque i1 de prem1 jusqu'à der1
et pour chaque i2 de prem2 jusqu'à prem2 + (der1 – prem1) faire
      result = result + (*i1) * (*i2)              // premier cas
      result = opbin1(result, opbin2(*i1, *i2))    // deuxième cas
```

Exemple : avec la déclaration

```
int t[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
```

l'expression

```
cout << inner_product(&t[0], &t[5], &t[5], 0);
```

affiche 520 (résultat de $0 + 2 \times 12 + 4 \times 14 + 6 \times 16 + 8 \times 18 + 10 \times 20$).

Somme partielle

```
template<class inIt, class outIt>
    outIt partial_sum(inIt prem, inIt der, outIt result);

template<class inIt, class outIt, class binOp>
    outIt partial_sum(inIt first, inIt last, outIt result, binOp opbin);
```

inIt et **outIt** représentent des types pouvant jouer les rôles de *input iterator* et *output iterator* respectivement. **binOp** un type pouvant jouer le rôle d'opérateur binaire.

Cette fonction effectue le travail suivant :

```
pour chaque ie de prem jusqu'à der
et pour chaque is de result jusqu'à result + (der – prem) faire
      *is = accumulate(prem, ie + 1, 0)            // premier cas
      *is = accumulate(prem, ie + 1, 0, opbin)     // second cas
```

Ces fonctions rendent comme résultat la valeur **result + (der – prem)**.

Exemple: avec les déclarations

```
int t[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
int s[10];
```

les expressions

```
partial_sum(&t[0], &t[10], &s[0]);

for (i = 0; i < 10; i++)
    cout << s[i] << ' ';
```

affichent :

```
2 6 12 20 30 42 56 72 90 110
```

Différences adjacentes

```
template<class inIt, class outIt>
    outIt adjacent_difference(inIt prem, inIt der, outIt result);

template<class inIt, class outIt, class binOp>
    outIt adjacent_difference(inIt prem, inIt der, outIt result,
                                                        binOp opbin);
```

inIt et **outIt** représentent des types pouvant jouer les rôles de *input iterator* et *output iterator* respectivement. **binOp** un type pouvant jouer le rôle d'opérateur binaire. Cette fonction effectue le travail suivant :

```
*result = *prem
pour chaque ie de prem + 1 jusqu'à der
et pour chaque is de result + 1 jusqu'à result + (der – prem) faire
      *is = *ie - *(ie - 1)                  // premier cas
      *is = opbin(*ie, *(ie - 1))            // second cas
```

Ces fonctions rendent comme résultat la valeur **result + (der – prem)**.

Exemple: avec les déclarations

```
int t[] = { 100, 102, 98, 100, 100, 101, 102, 101, 100, 0 };
int s[10];
```

les expressions

```
adjacent_difference(&t[0], &t[10], &s[0]);

for (i = 0; i < 10; i++)
    cout << s[i] << ' ';
```

affichent :

```
100 2 -4 2 0 1 1 -1 -1 -100
```

8.7 Flux d'entrée-sortie

8.7.1 Structure générale

La bibliothèque des flux d'entrée-sortie est un des éléments les plus anciens de la bibliothèque C++. Plusieurs fois remaniée au cours de l'histoire de C++, traitée avec plus ou moins de rigueur par les éditeurs d'environnements de développement, elle est devenue de plus en plus pratique, mais aussi de plus en plus complexe. La norme ISO a réduit le nombre de composants et clarifié leur rôle mais, en définissant des modèles à la place de classes, elle a rendu l'exposé encore plus difficile à lire.

Les pages qui suivent sont une explication *allégée* des flux, en ce sens que

- nous avons passé sous silence certaines classes et fonctions secondaires, invisibles ou inutiles pour le programmeur moyen. C'est le cas, par exemple, des éléments définis dans le fichier **<streambuf>**, et de beaucoup de *doublures* (des fonctions dont l'effet est le même que celui de fonctions qui sont expliquées par ailleurs),
- dans la norme, la plupart des classes flux sont introduites comme des modèles, paramétrés par le type des caractères que les flux produisent ou consomment. Par exemple, la norme spécifie le modèle

  ```
  template<class T, class traits = char_traits<T> > class basic_iostream;
  ```

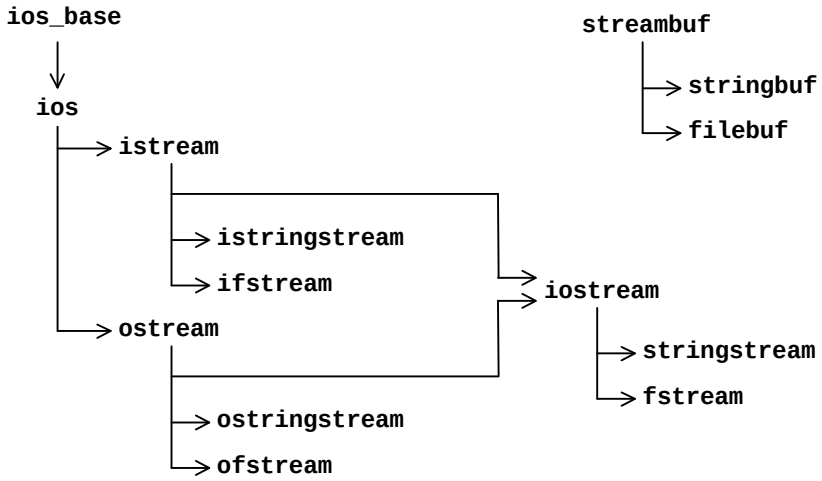
  (on imagine la lourdeur des expressions à manipuler) alors que le programmeur n'est généralement concerné que par les deux spécialisations suivantes, surtout la première :

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

Nous avons donc choisi de « spécialiser » notre explication, en prenant d'entrée **T** = **char** et en ne pas mentionnant les modèles **basic_...** Nous sacrifions ainsi le cas **T** = **wchar_t** (et d'autres cas éventuels) mais le texte devient beaucoup plus facile à lire.

Hiérarchie des flux d'entrée-sortie

Les relations d'héritage existant entre les classes flux peuvent être représentées par le diagramme suivant :



Fichiers en-tête

<iostream>	*(flux d'entrée-sortie standard)*
<ios> **<iosfwd>**	*(classes de base des flux)*
<streambuf>	*(tampons associés aux flux)*
<istream> **<ostream>** **<iomanip>**	*(entrées-sorties formatées)*
<sstream> **<cstdlib>**	*(flux associés à des chaînes de caractères)*
<fstream> **<cstdio>** **<cwchar>**	*(flux associés à des fichiers)*

Ces fichiers s'incluant mutuellement de nombreuses manières, il est plus difficile qu'ailleurs d'associer chaque fichier en-tête à un ensemble bien délimité de classes et fonctions.

Le fichier <ios>

Le fichier **<ios>** déclare les classes **ios_base** et **ios**. La classe **ios_base** définit une collection de constantes utilisées dans **ios**. La classe **ios** dérive de **ios_base** et est la classe de base de toutes les classes flux.

Essentiellement, sont définis dans ce fichier :

- la classe **failure**, sommet de la hiérarchie des exceptions qui peuvent être lancées à l'occasion d'opérations d'entrée sortie,
- divers éléments en rapport avec les *modes* des flux : mode d'ouverture, mode pour l'accès direct, précision, etc.
- des membres pour représenter (et questionner) l'état du flux (**good**, **eof**, **fail** et **bad**)
- des membres pour gérér la base de la représentation écrite des nombres, les caractéristiques du formatage, le caractère de bourrage, etc. Les valeurs de ces membres sont modifiées à l'aide des *manipulateurs* (cf. § 8.7.3).

8.7.2 Lecture et écriture

Lecture

La classe **istream** fonctionne *comme si* elle avait été ainsi définie :

```
class istream : virtual public ios
    {
public:
    explicit istream(streambuf);
    virtual ~istream();

        // LECTURES FORMATEES

    istream &operator>>(bool &b);
    istream &operator>>(short &s);
    istream &operator>>(unsigned short &us);
    istream &operator>>(int &i);
    istream &operator>>(unsigned int &ui);
    istream &operator>>(long &l);
    istream &operator>>(unsigned long &ul);
    istream &operator>>(float &f);
    istream &operator>>(double &d);
    istream &operator>>(long double &ld);
    istream &operator>>(void* &pv);
    istream &operator>>(streambuf *sb);

        // manipulateurs
    istream &operator>>(istream &(*pf)(istream &))
    istream &operator>>(ios &(*pf)(ios &))

        // LECTURES NON FORMATEES

        // lecture d'un caractère
    int get();
    istream &get(char &c);

        // lecture d'une suite de caractères, jusqu'à la fin du fichier, ou jusqu'à n-1 caractères
        // utiles, ou jusqu'à delim (qui reste disponible pour une lecture ultérieure)
    istream &get(char *s, int n);
    istream &get(char *s, int n, char_type delim);
    istream &get(streambuf &sb);
    istream &get(streambuf &sb, char_type delim);

        // les mêmes fonctions que ci-dessus, à peu de choses près
    istream &getline(char *s, int n);
    istream &getline(char *s, int n, char delim);

        // lecture de caractères à fond perdu jusqu'à la fin du fichier, ou jusqu'à n-1 caractères
        //  utiles, ou jusqu'à delim (qui reste disponible pour une lecture ultérieure)
    istream &ignore(int n = 1, int delim = eof());

        // le prochain caractère à lire (sans l'extraire du flux d'entrée)
    int peek();

        // lecture de n caractères (ou jusqu'à la fin du fichier) et rangement dans dest
    istream &read (char *dest, int n);

        // « délecture » du caractère c ou du dernier caractère préalablement lu
    istream &putback(char c);
    istream &unget();

        // synchronisation du tampon de lecture et de la source externe de caractères
    int sync();

        // nombre de caractères obtenus lors de la dernière lecture non formatée
    int gcount() const;

        // position courante sur le flux (position du prochain caractère lu)
    int tellg();
```

```
        // positionnement sur un flux
        // (dir est un de ios_base::beg, ios_base::cur, ou ios_base::end)
    istream &seekg(int p);
    istream &seekg(int o, ios_base::seekdir dir);
    };
```

Comme on s'en doutait, le rôle principal de la classe **istream** est la mise en place de surcharges de l'opérateur **>>** pour un **istream** et une *lvalue*, généralement une référence, d'un type standard (pour surcharger >> pour des types définis par l'utilisateur il faut définir des fonctions non membres).

A propos des fonctions **seekg** de positionnement dans les flux, on notera qu'elles ne sont pas forcément supportées par toutes les classes dérivées de **istream** (seuls les flux basés sur des fichiers supportent le positionnement).

Manipulateurs

Les deux fonctions qualifiées ci-dessus de « manipulateurs » ont un comportement spécial. Elles définissent les surcharges de **>>** utilisées lorsque le second opérande est *une fonction qui prend un* **istream** *(resp. un* **ios***) et rend un* **istream** *(resp. un* **ios***)*. Une telle fonction s'appelle un manipulateur sans arguments (cf. § 8.7.2) ; elle sert à modifier le fonctionnement du flux. Ce que le flux fait d'un manipulateur est extrêmement simple, et laisse la porte ouverte à la définition ultérieure d'autant de manipulateurs qu'on veut :

```
istream &istream::operator >> (istream &(*pf)(istream &))
    {                                          // une implémentation possible
    (*pf)(*this);
    return *this;
    }

istream &istream::operator >> (ios &(*pf)(ios &))
    {                                          // une implémentation possible
    (*pf)(*this);
    return *this;
    }
```

A propos des manipulateurs prédéfinis, voir ci-dessous § 8.7.3.

Ecriture

La classe **ostream** est l'analogue de **istream** pour les écritures :

```
class ostream : virtual public ios
    {
public:
    explicit ostream(streambuf* sb);
    virtual ~ostream();

        // ECRITURES FORMATEES

    ostream &operator<<(bool b);
    ostream &operator<<(short s);
    ostream &operator<<(unsigned short us);
    ostream &operator<<(int i);
    ostream &operator<<(unsigned int ui);
    ostream &operator<<(long l);
    ostream &operator<<(unsigned long ul);
    ostream &operator<<(float f);
    ostream &operator<<(double d);
    ostream &operator<<(long double ld);
    ostream &operator<<(const void *pv);
    ostream &operator<<(streambuf *sb);

        // manipulateurs
    ostream &operator<<(ostream &(*pf)(ostream &));
    ostream &operator<<(ios &(*pf)(ios &));

        // ECRITURES NON FORMATEES

        // ecriture d'un caractère
    ostream &put(char c);
```

```
    // ecriture du contenu d'un tampon
ostream &write(const char *s, int n);

    // vidage du tampon
ostream &flush();

    // position courante sur le flux
int tellp();

    // positionnement
ostream &seekp(int p);
ostream &seekp(int o, ios_base::seekdir dir);
};
```

Lecture et écriture

La classe **`iostream`** est ainsi définie :

```
class iostream : public istream, public ostream
    {
public:
    explicit iostream(streambuf *sb);
    virtual ~iostream();
    };
```

Un objet de cette classe supporte donc toutes les opérations de lecture et d'écriture expliqués précédemment.

8.7.3 Manipulateurs

Les *manipulateurs des flux d'entrée-sortie* sont des objets dont l'insertion dans un flux provoque un changement dans le mode de fonctionnent du flux. Par exemple, si **`x`**, **`y`**, et **`z`** sont des variables de type **`int`**, l'expression

```
cin >> x >> hex >> y >> dec >> z;
```

produit la lecture de trois nombres entiers, le premier et le troisième étant supposés écrits en décimal, le second en hexadécimal.

Manipulateurs sans arguments :

• Affichage symbolique des booléens :

```
boolalpha
noboolalpha
```

Exemple : l'expression

```
cout << (3 + 2 == 5) << ' ' << boolalpha << (3 + 2 == 5);
```

affiche :

```
1 true
```

• Affichage de la base (lorsque ce n'est pas 10) :

```
showbase
noshowbase
```

Exemple : l'expression

```
cout << hex << 256 << ' ' << showbase << 256;
```

affiche :

```
100 0x100
```

• Affichage du point décimal :

```
showpoint
noshowpoint
```

Exemple : l'expression

```
cout << 20.0 << ' ' << showpoint << 20.0;
```


affiche :

```
20 20.0000
```

• Affichage de + devant les nombres non négatifs :

```
showpos
noshowpos
```

Exemple : l'expression

```
cout << 20 << ' ' << showpos << 20;
```

affiche :

```
20 +20
```

• Saut des blancs précédant des données à lire :

```
skipws
noskipws
```

Exemple : si, lors de l'évaluation de l'expression

```
cin >> x >> noskipws >> y;
```

l'utilisateur tape deux nombres précédés de un ou plusieurs blancs, le premier sera bien lu mais le second non.

• Mise en majuscule de *certains* caractères :

```
uppercase
nouppercase
```

Exemple : l'expression

```
cout << hex << showbase << 256 << ' ' << uppercase << 256;
```

affiche :

```
0x100 OX100
```

• Cadrage :

```
left
right
internal
```

Exemple : l'expression

```
cout << "'" << setw(20)          << 123 << "'\n"
     << "'" << setw(20) << left  << 123 << "'\n"
     << "'" << setw(20) << right << 123 << "'\n";
```

affiche les trois lignes :

```
'                 123'
'123                 '
'                 123'
```

• Base, pour l'expression des nombres :

```
dec
hex
oct
```

Exemple : l'expression

```
cout << 27 << ' ' << hex << 27 << ' ' << oct << 27 << ' ' << "\n";
```

affiche :

```
27 1b 33
```

• Ecriture des nombres flottants :

```
fixed
scientific
```

Exemple : l'expression

```
cout << 0.123 << ' ' << fixed << 0.123 << ' ' << scientific << 0.123;
```

affiche :

```
0.123 0.123000 1.230000e-001
```

• Sauter les blancs (**istream** uniquement) :

```
ws
```

Exemple : si **x** est une variable de type **int** et **c** une variable de type **char**, l'expression

```
cin >> noskipws >> x >> ws >> c;
```

affectera à **x** un nombre entier, qu'il aura fallu taper non précédé de blancs, puis lira « à fonds perdu » tous les blancs tapés à la suite du nombre, enfin affectera à **c** le premier caractère non blanc rencontré.

• Ecriture d'une fin de ligne (**ostream** uniquement) :

```
endl
```

Ce manipulateur écrit une fin de ligne **'\n'** sur le flux de sortie, puis vidange ce dernier.

• Ecriture d'une fin de chaîne (**ostream** uniquement) :

```
ends
```

Ce manipulateur écrit une fin de chaîne **'\0'** sur le flux de sortie.

• Vidange du tampon d'écriture (**ostream** uniquement) :

```
flush
```

Le contenu du tampon du flux est « physiquement » envoyé sur l'unité de sortie.

Manipulateurs avec paramètres

NOTE. L'inclusion du fichier **<iomanip>** est nécessaire.

• Spécification de la largeur de la zone.

```
setw(int n)
```

Cette définition n'est valable que pour une opération d'écriture, la première de celles qui suivent. Exemple : l'expression

```
cout << setw(10) << 12345 << '\n' << 12345;
```

affiche les deux lignes :

```
     12345
12345
```

• Spécification de la base :

```
setbase(int base)
```

(base doit être 8, 10 ou 16). Exemple : l'expression

```
cout << setbase(8) << 73 << ' ' << showbase << 73;
```

affiche :

```
111 0111
```

• Spécification du caractère de bourrage :

```
setfill(char c)
```

Exemple : l'expression

```
cout << setw(8) << setfill('#') << 999.50 << '\n';
```


affiche :

```
###999.5
```

• Spécification du nombre de chiffres significatifs de l'affichage :

```
setprecision(int n)
```

Exemple : l'expression

```
cout << setprecision(2) << 0.000111111 << '    '
     <<  1.11111 << '    ' <<  11111.1 << '    ' <<  111111000.0;
```

affiche :

```
0.00011    1.1    1.1e+04    1.1e+08
```

8.7.4 Flux basés sur des chaînes de caractères

Ces flux permettent de tirer profit des possibilités de formatage de données qu'ont les fonctions d'entrée-sortie, en effectuant des opérations qui ne sont pas des échanges avec l'extérieur, mais de simples transferts en mémoire.

Le fichier **<sstream>** définit les quatre classes **stringbuf**, **istringstream**, **ostringstream** et **stringstream**

```
class stringbuf : public streambuf
    {
public:
    explicit stringbuf(
                ios_base::openmode which = ios_base::in | ios_base::out);
    explicit stringbuf(const string &str,
                ios_base::openmode which = ios_base::in | ios_base::out);

        // obtention de la chaîne associée au flux
    string str() const;

        // changement de la chaîne associée au flux
    void str(const string &s);

protected:
    ...
    si vous envisagez d'écrire des classes dérivées de stringbuf
    vous avez intérêt à vous procurer un document plus détaillé que ce poly !
    ...
    };
```

Les classes **istringstream**, **ostringstream** et **stringstream** reproduisent la classe **stringbuf** et, en même temps, héritent des classes **istream**, **ostream** et **iostream**.

1. Voici, par exemple, une manière simple d'obtenir la valeur du nombre dont une chaîne telle que **"-1.234e004"** est l'expression écrite :

```
double x;
istringstream is("-1.234e004");

is >> x;
```

2. Voici un exemple complet : la fonction **aujourdhui()** renvoie un **string** exprimant la date courante en français :

```
#include <sstream>
#include <ctime>
using namespace std;

string aujourdhui()
    {
    static string jsem[] = { "Dimanche", "Lundi", ... "Samedi" };
    static string mois[] = { "janvier", "fevrier", ... "decembre" };
```

```
    time_t t = time(0);
    struct tm *j = localtime(&t);

    ostringstream result;
    result << jsem[j->tm_wday] << ' ' << j->tm_mday << ' '
           << mois[j->tm_mon] << ' ' << 1900 + j->tm_year;
    return result.str();
    }
```

Essai :

```
#include <iostream>

main()
    {
    cout << aujourdhui() << '\n';
    }
```

Affichage obtenu (mais cela change tous les jours !) :

```
Dimanche 2 janvier 2000
```

8.7.5 Flux basés sur des fichiers

Il s'agit ici d'utiliser des flux connectés à des fichiers, c'est-à-dire des entités du type **`FILE`** comme défini dans **`<cstdio>`**.

Le fichier **`<fstream>`** définit les quatre classes **`filebuf`**, **`ifstream`**, **`ofstream`** et **`fstream`**. On notera qu'elles ne différent que par le constructeur avec argument et la fonction membre **`open`** :

```
class filebuf : public streambuf
    {
public:
    filebuf();
    virtual ~filebuf();

    bool is_open() const;
    filebuf *open(const char* s, ios_base::openmode mode);
    filebuf *close();
protected:
    ...
    la remarque dans la classe stringbuf reste valable…
    ...
    };

class ifstream : public istream
    {
public:
    ifstream();
    explicit ifstream(const char* s,
                      ios_base::openmode mode = ios_base::in);
    filebuf *rdbuf() const;
    bool is_open();
    void open(const char* s, ios_base::openmode mode = ios_base::in);
    void close();
    ...
    };
```

```
class ofstream : public ostream
    {
public:
    ofstream();
    explicit ofstream(const char* s,
                          ios_base::openmode mode = ios_base::out);
    filebuf *rdbuf() const;
    bool is_open();
    void open(const char* s, ios_base::openmode mode = ios_base::out);
    void close();
    ...
    };

class fstream : public iostream
    {
public:
    fstream();
    explicit fstream(const char* s,
                ios_base::openmode mode = ios_base::in|ios_base::out);
    filebuf *rdbuf() const;
    bool is_open();
    void open(const char* s,
                ios_base::openmode mode = ios_base::in|ios_base::out);
    void close();
    ...
    };
```

Exemple : la copie « en vrac » d'un fichier sur un autre :

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

void main(int argc, char *argv[])
    {
    if (argc != 3) cerr << "Emploi: copie <fic1> <fic2>";
    else
        {
        ifstream fi(argv[1]);
        if ( ! fi) cerr << "Je ne trouve pas " << argv[1];
        else
            {
            ofstream fo(argv[2]);
            if ( ! fo) cerr << "Je ne peux pas creer " << argv[2];
            char ch;
            while (fo && fi.get(ch))
                fo.put(ch);
            }
        }                   // les fichiers sont fermés lors de la
    }                       // destruction des objets fi et fo
```

8.7.6 Flux standard

Le fichier **<iostream>** déclare huit flux standard :

```
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;

extern wistream wcin;
extern wostream wcout;
extern wostream wcerr;
extern wostream wclog;
```

Ce sont des flux de caractères ; les quatre premiers sont basés sur le type **`char`**, les quatre autres sur le type **`wchar_t`**.

Ces flux sont ouverts lorsque le programme commence. Ils contrôlent les insertions et extractions de caractères dans ou depuis les unités **`stdin`**, **`stdout`** et **`stderr`** (déclarées dans **`<cstdio.h>`**). Le flux **`cerr`** produit des insertions *non bufférisées* de caractères dans l'unité **`stderr`**, alors que **`clog`** effectue des insertions *bufférisées* dans cette même unité.

☺ ☹ ☹ ☹

Chapitre 9. Les structures de données et les algorithmes (la STL)

Dans ce chapitre nous expliquons la partie de la bibliothèque qui correspond à la *STL* ou *Standard Template Librarie*, une bibliothèque restée longtemps expérimentale et qui est maintenant partie intégrante de la norme.

9.1 Utilitaires généraux

Fichiers en-tête :

<utility> **<functional>** **<memory>** **<ctime>**

9.1.1 Utilitaires

<utility>

Ce fichier déclare quelques modèles de fonctions et de classes utilisés par le reste de la bibliothèque.

Opérateurs de relation dérivés

On notera que les identificateurs ci-dessous sont membres de l'espace de noms **rel_ops**, lui-même membre, comme tous les éléments de la bibliothèque standard, de l'espace de noms **std** (autrement dit, ces noms sont membres de l'espace de noms **std::rel_ops**).

```
namespace rel_ops
    {
    template<class T> bool operator!=(const T &, const T &);
    template<class T> bool operator> (const T &, const T &);
    template<class T> bool operator<=(const T &, const T &);
    template<class T> bool operator>=(const T &, const T &);
    }
```

Ces fonctions permettent d'éviter des redondances, en définissant une fois pour toutes les opérateurs de comparaison qui se déduisent des opérateurs fondamentaux **==** et **<**. Ainsi, toute classe qui définit ces deux opérateurs dispose automatiquement des quatre autres.

Exemple :

```
struct Point
    {
    int x, y;
    Point(int a, int b)
        { x = a; y = b; }
    bool operator==(Point b) const
        { return x == b.x && y == b.y; }
    bool operator<(Point b) const
        { return x < b.x || x == b.x && y < b.y; }
    };
```

La classe **`Point`** possède les deux opérateurs **`==`** et **`<`**. Il suffit donc de faire la déclaration

```
#include <utility>
using namespace rel_ops;
```

pour rendre légitimes des expressions comme

```
Point p;

if (p != Point(0, 0)) ...
```

Paires

Le fichier **`<utility>`** déclare également :

```
template<class T1, class T2> struct pair;

template<class T1, class T2> pair<T1,T2> make_pair(const T1 &, const T2 &);

template<class T1, class T2>
    bool operator==(const pair<T1,T2> &, const pair<T1,T2> &);
template<class T1, class T2>
    bool operator<(const pair<T1,T2> &, const pair<T1,T2> &);
```

avec le modèle **`pair`** ainsi défini :

```
template<class T1, class T2> struct pair
    {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair();
    pair(const T1 &x, const T2 &y);
//      : first(x), second(y) { }                   une implémentation possible
    template<class U, class V> pair(const pair<U, V> &p);
//      : first(p.first), second(p.second) { }      une implémentation possible
    };
```

Il s'agit encore ici d'éviter des redondances, en définissant une fois pour toutes le matériel nécessaire à la construction et la comparaison de *paires d'objets* quelconques.

Par exemple, la classe **`Point`** utilisée plus haut aurait pu être définie en écrivant simplement :

```
typedef pair<int, int> Point;
```

On notera que le constructeur **`template<class U, class V> pair(const pair<U, V> &p)`** précise comment *convertir* des paires lorsque les types des composantes sont respectivement convertibles. Par exemple, parce que les valeurs des types **`char`** et **`double`** peuvent être converties dans le type **`int`**, l'expression suivante est [bizarre mais] légitime :

```
Point p = pair<char, double>('A', 1.5);
```

On notera également que la fonction **`make_pair`**, dont la définition est une banalité :

```
template<class T1, class T2>
    pair<T1,T2> make_pair(const T1 &x, const T2 &y);
//      { return pair<T1, T2>(x, y); }              une implémentation possible
```

permet d'alléger des expressions comme le membre droit de l'affectation précédente. En effet, l'expression

```
pair<char, double>('A', 1.5)        // types arguments de pair explicites
```

équivaut à :

```
make_pair('A', 1.5)                 // types arguments de pair déduits de
                                    // l'appel de fonction
```


9.1.2 Objets fonctions

Fichier en-tête :

`<functional>`

Un *objet fonction*, ou *foncteur*, est une entité qui peut être appelée comme une fonction.

Une fonction et un pointeur sur une fonction sont donc des foncteurs. Mais, et c'est le cas qui va nous intéresser principalement ici, un objet pour lequel on a défini une ou plusieurs surcharges de l'opérateur **`()`** est également un foncteur ; nous réserverons l'expression *objet fonction* pour désigner de tels objets. Ils jouent un rôle capital dans la définition et l'utilisation de la bibliothèque standard car, dans la plupart des algorithmes de cette dernière, là où une fonction est requise *on peut* [et pour certains algorithmes *on doit*] mettre un objet fonctionnel.

Objets fonctions de la bibliothèque

Le fichier **`<functional>`** déclare une collection d'objets fonctions fréquemment utilisés, ainsi qu'un certain nombre d'opérations fonctionnelles pour obtenir de nouvelles fonctions à partir de fonctions existantes :

```
    // modèles de base
template<class Arg, class Result> struct unary_function;
template<class Arg1, class Arg2, class Result> struct binary_function;

    // opérations arithmétiques
template<class T> struct plus;
template<class T> struct minus;
template<class T> struct multiplies;
template<class T> struct divides;
template<class T> struct modulus;
template<class T> struct negate;                          // unaire

    // comparaisons
template<class T> struct equal_to;
template<class T> struct not_equal_to;
template<class T> struct greater;
template<class T> struct less;
template<class T> struct greater_equal;
template<class T> struct less_equal;

    // opérations logiques
template<class T> struct logical_and;
template<class T> struct logical_or;
template<class T> struct logical_not;                     // unaire

    // négateurs
template<class Predicate> struct unary_negate;            // unaire
template<class Predicate> unary_negate<Predicate not1(const Predicate &);
template<class Predicate> struct binary_negate;
template<class Predicate> binary_negate<Predicate> not2(const Predicate &);

    // lieurs
template<class Operation> class binder1st;                // unaire
template<class Operation, class T>
               binder1st<Operation> bind1st(const Operation&, const T&);
template<class Operation> class binder2nd;                // unaire
template<class Operation, class T>
               binder2nd<Operation> bind2nd(const Operation&, const T&);

    // adaptateurs  pour les fonctions
template<class Arg, class Result> class pointer_to_unary_function;
template<class Arg, class Result>
       pointer_to_unary_function<Arg,Result> ptr_fun(Result (*)(Arg));
template<class Arg1, class Arg2, class Result>
                                    class pointer_to_binary_function;
template<class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1,Arg2,Result> ptr_fun(Result (*)(Arg1,Arg2));
```

etc.

Nous ne passerons pas en revue chacun des modèles listés ci-dessus, le document de la norme ISO est fait pour cela. Pour fixer les idées, en voici deux : **`binary_function`** et **`plus`** :

```
template<class Arg1, class Arg2, class Result> struct binary_function
    {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
    };

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

Les modèles **`unary_function`** et **`binary_function`** sont des modèles de base : tout modèle représentant une fonction à un [resp. deux] arguments *peut* être défini comme dérivant de **`unary_function`** [resp. **`binary_function`**]. Mais ce n'est pas une obligation, et cela n'a pas une grande utilité : ce modèle ne définit ni donnée membre ni fonction membre (à ce propos, voyez également la remarque ci-dessous, « Les objets fonctionnels ne relèvent pas du polymorphisme »).

Exemple d'utilisation : soit à calculer l'addition composante à composante de deux vecteurs **`a`** et **`b`** dont les éléments sont de type **`double`**, le résultat étant déposé dans **`a`**. En utilisant l'algorithme **`transform`** (cf. § 9.3) de la bibliothèque standard, il suffit d'écrire

```
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
```

NOTE. Lisez correctement le dernier argument **`plus<double>()`** de l'expression précédente : les parenthèses n'expriment pas un appel de la fonction addition, mais un appel du constructeur par défaut de la classe **`plus<double>`** ; ainsi, la fonction **`transform`** est appelée avec un objet « addition de deux doubles » créé pour cette occasion.

Les objets fonctionnels ne relèvent pas du polymorphisme

En étudiant le couple **`binary_function`** et **`plus`** montré ci-dessus, on constate que les classes du fichier **`<functional>`** font une utilisation atypique de la programmation orientée objets, et notamment qu'elles ne tirent aucun profit du polymorphisme. Utiliser le polymorphisme aurait consisté à définir une fonction virtuelle, probablement pure, dans la classe **`binary_function`** :

```
template<class Arg1, class Arg2, class Result> struct binary_function
    {
    ...                                          // CECI N'EXISTE PAS
    virtual Result operator()(const Arg1 &, const Arg2 &) const = 0;
    ...
    };
```

Les classes dérivées de **`binary_function`** auraient alors donné des versions concrètes de cette fonction :

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

Cela aurait permis de programmer des algorithmes où les arguments fonctions auraient été effectivement déclarés comme des pointeurs ou des références sur des **`binary_function`**.

Une telle méthodologie a été jugée trop onéreuse. L'utilisation des opérations de base (addition, multiplication, comparaisons, etc.) faite par les algorithmes de la bibliothèque est assez fréquente et critique pour interdire le surcoût entraîné par des appels de fonctions virtuelles (cf. § 4.3.5).

La classe **`binary_fonction`** n'est donc qu'une commodité « mineure », permettant de nommer une fois pour toutes trois types qui jouent un rôle dans une fonction binaire. Le service habituellement rendu par la virtualité ne joue pas, remplacé par la généricité : là où une fonction binaire est attendue, il faudra mettre *dès la compilation* la fonction binaire précise souhaitée.

Dans la suite de ce chapitre on trouvera plusieurs exemples d'utilisation d'objets fonctionnels (comme le deuxième exemple de **`priority_queue`**, au § 9.2.4).

Liaisons

L'opération de liaison consiste à construire une fonction à *n*-1 arguments à partir d'une fonction *fon* à *n* arguments et une valeur *val*. Si *fon* est unaire, cela produit une fonction sans argument, c'est-à-dire une constante. Si *fon* est binaire, cela produit une fonction unaire. Par exemple, la « liaison du premier argument » prend une fonction binaire *fon* et une valeur *val* et produit la fonction unaire *g* définie par

$$g : y \to fon(val, y)$$

On n'est pas surpris de constater que *g* est essentiellement constitué par la mémorisation de *fon* et de *val* :

```
template<class binOp> class binder1st
        : public unary_function<typename binOp::second_argument_type,
                                typename binOp::result_type>
    {
protected:
    binOp fon;
    typename binOp::first_argument_type val;
public:
    binder1st(const binOp &f, const typename binOp::first_argument_type &v);
//      : fon(f), val(v) { }                        une implémentation possible
    typename binOp::result_type
            operator()(const typename binOp::second_argument_type& y) const;
//      { return fon(val, y); }                     une implémentation possible
    };
```

Exemple d'utilisation : supposons posséder un vecteur

```
vector<double> a(N);
```

sur lequel nous voulons effectuer une opération unaire composante à composante. Diverses possibilités :

1. L'opération unaire est une fonction ordinaire de la bibliothèque (exemple : $a_i \to \sin(a_i)$) :

```
transform(a.begin(), a.end(), a.begin(), sin);
```

2. L'opération unaire est un objet fonction de la bibliothèque (exemple : $a_i \to -a_i$) :

```
transform(a.begin(), a.end(), a.begin(), negate<double>());
```

3. L'opération unaire peut être vue comme la liaison d'une opération binaire (exemple : $a_i \to 3 \times a_i$) :

```
transform(a.begin(), a.end(), a.begin(),
        binder1st<multiplies<double> >(multiplies<double>(), 3));
```

REMARQUE. L'expression précédente peut surprendre. Il aurait été plus satisfaisant de l'écrire :

```
transform(a.begin(), a.end(), a.begin(),                    // ERREUR
    binder1st<binary_function<double, double, double> >     // ERREUR
                            (multiplies<double>(), 3));     // ERREUR
```

mais cela ne marche pas, car le type **`binary_function<double, double, double>`** n'est pas un objet fonction (relisez la section *Les objets fonctionnels ne relèvent pas du polymorphisme* ci-dessus).

AUTRE REMARQUE. Comme nous l'avons vu à d'autres endroits, instancier un modèle de classe à travers une fonction amène à des expressions plus simples, car les arguments du modèle sont déduits de l'appel de la fonction. Ici, c'est la fonction **`bind1st`** (ne pas confondre avec **`binder1st`**) qui rend ce service :

```
template<class Op, class T>
        binder1st<Op> bind1st(const Op &ope, const T &val);
//  { return binder1st<Op>(ope, val); }            une implémentation possible
```

avec cela, l'expression :

```
transform(a.begin(), a.end(), a.begin(),
        binder1st<multiplies<double> >(multiplies<double>(), 3));
```

peut s'écrire plus simplement :

```
transform(a.begin(), a.end(), a.begin(), bind1st(multiplies<double>(), 3));
```

Adaptateurs

Beaucoup de fonctions de la bibliothèque acceptent indifféremment des fonctions (ou des pointeurs vers des fonctions) et des objets fonctions, mais d'autres requièrent précisément des objets fonctions. Les *adaptateurs* servent à construire des objets fonctions à partir de fonctions.

Ainsi, pour les fonctions unaires (on a le même pour les fonctions binaires) :

```
template<class Arg, class Result> class pointer_to_unary_function
        : public unary_function<Arg, Result>
    {
    Result (*fon)(Arg);
public:
    explicit pointer_to_unary_function(Result (* f)(Arg));
//      fon(f) { }                                  une implémentation possible
    Result operator()(Arg x) const;
//      { return (*fon)(x); }                       une implémentation possible
    };
```

EXEMPLE. En supposant qu'on possède une fonction **`odd`** qui caractérise les nombres impairs

```
inline bool odd(int n)
    { return (n % 2) != 0; }
```

imaginons que **`b`** est un vecteur d'entiers et **`c`** un vecteur de booléens, et qu'on souhaite affecter **`c`** de telle manière que c_i vrai $\Leftrightarrow$ b_i est pair.

La première idée ne marche pas :

```
transform(b.begin(), b.end(), c.begin(), not1(odd));   // ERREUR
```

car **`not1`** requiert un objet fonction et n'accepte pas une fonction. La bonne version

```
transform(b.begin(), b.end(), c.begin(),
                not1(pointer_to_unary_function<int, bool>(odd)));
```

Comme d'habitude, passer par une fonction simplifie les constructions :

```
template<class Arg, class Result>
    pointer_to_unary_function<Arg, Result> ptr_fun(Result (* f)(Arg));
//      { return (*fon)(x); }                       une implémentation possible
```

L'appel de **`transform`** devient :

```
transform(b.begin(), b.end(), c.begin(), not1(ptr_fun(odd)));
```

Les autres classes et fonctions de **`<functional>`** sont des adaptateurs pour faire apparaître les pointeurs vers des fonctions membres comme des objets fonctions. Pour les détails on se référera à la norme ISO.

9.1.3 Mémoire : allocateurs et auto-pointeurs

Le fichier

```
<memory>
```

déclare les éléments suivants :

```
    // auto-pointeurs
template<class X> class auto_ptr;

    // l'allocateur par défaut
template<class T> class allocator;
template<> class allocator<void>;

template<class T, class U>
    bool operator==(const allocator<T>&, const allocator<U>&) throw();
template<class T, class U>
    bool operator!=(const allocator<T>&, const allocator<U>&) throw();

    // utilisation de « mémoire brute »
template<class OutputIterator, class T> class raw_storage_iterator;

    // tampons temporaires
template<class T> pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template<class T> void return_temporary_buffer(T* p);

    // algorithmes spécialisés
template<class inIter, class ForwardIterator>
    ForwardIterator
        uninitialized_copy(inIter first,
                           inIter last, ForwardIterator result);
template<class ForwardIterator, class T>
    void uninitialized_fill(ForwardIterator first,
                            ForwardIterator last, const T& x);
template<class ForwardIterator, class Size, class T>
    void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

Auto-pointeurs

Comme les allocateurs du § 6.2.1, les auto-pointeurs sont des objets qui encapsulent un pointeur dont ils garantissent la libération au moment opportun. Les points clés de cette notion sont :

- un auto-pointeur est un objet contenant un pointeur provenant d'un appel de **new** ;
- l'auto-pointeur peut être propriétaire du pointeur, auquel cas il lui incombe de libérer l'espace pointé lorsqu'il est lui-même détruit ;
- à tout instant, un pointeur est la propriété d'un seul auto-pointeur ;
- la copie (que ce soit à l'occasion d'une initialisation ou bien d'une affectation) d'un auto-pointeur transfère la propriété du pointeur.

```
template<class X> class auto_ptr
    {
public:
    explicit auto_ptr(X* p = 0) throw();
//      : owns(p != 0), ptr(p) { }              une implémentation possible
    auto_ptr(auto_ptr &p) throw();
//      : owns(p.owns), ptr(p.release()) { }    une implémentation possible
    ~auto_ptr() throw();
//      { if (owns) delete ptr; }               une implémentation possible

    template<class Y> auto_ptr(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr&) throw();
    template<class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();
    X& operator*() const throw();
    X* operator->() const throw();
```

```
    X* get() const throw();
//      { return ptr; }                          une implémentation possible
    X* release() throw();
//      { owns = false; return ptr; }

private:
//  X *ptr;                                      une implémentation possible
//  bool owns;                                   une implémentation possible
    };
```

Exemple :

```
void main()
    {
    auto_ptr<Point> p(new Point(2, 3));
        ici, p est propriétaire du pointeur (anonyme) sur le point (2,3)
    ...
    les accès à l'objet pointé et à ses membres se font normalement : *p, p->x, etc.
    ...
    auto_ptr<Point> q = p;
        maintenant, c'est q le propriétaire du pointeur : *p et p->x marchent toujours,
        mais c'est q  qui a la responsabilité de la libération de l'objet pointé
    ...
        à la fin de la fonction, l'objet pointé est libéré par la destruction de q
    }
```

ATTENTION. Le principal inconvénient des auto-pointeurs est le danger qu'il y a à passer par copie un auto-pointeur à une fonction, car la fonction acquiert la propriété du pointeur, et donc le droit de le détruire :

```
void uneFonction(auto_ptr<Point> q);

void main()
    {
    auto_ptr<Point> p(new Point(2, 3));
    ...
    uneFonction(p);
    ici, l'objet pointé par p aura été détruit lors de la destruction de q, à la fin de  uneFonction
    ...
    }
```

Deux solutions à ce problème : soit la fonction se prête à être déclarée avec un argument référence :

```
void uneFonction(const auto_ptr<Point> &q);
```

soit la fonction appelante se charge de dupliquer le pointeur au moment de l'appel (ce qui diminue de beaucoup l'intérêt des auto-pointeurs) :

```
void main()
    {

    auto_ptr<Point> r(new Point(p->x, p->y));
    uneFonction(r);

    }
```

Allocateurs

ATTENTION. La notion d'allocateur au sens de la bibliothèque standard, expliquée ici, n'est pas la même que celle « d'allocateur pour encapsuler un pointeur », introduite au § 6.2.1.

Les allocateurs assurent la réservation et la libération de la mémoire. Ils fournissent une interface de bas niveau pour l'allocation efficace de petits objets. Ils permettent l'utilisation de différents schémas de gestion de la mémoire, il suffit pour cela de se donner plusieurs allocateurs différents.

Les principaux utilisateurs des allocateurs sont les classes conteneurs de la bibliothèque.

Le modèle **`allocator`** est appelé « allocateur par défaut » parce que c'est celui qui est utilisé par défaut par les conteneurs :

```
template <class T> class allocator
    {
public:
    ...
    allocator() throw();
    allocator(const allocator&) throw();
    template <class U> allocator(const allocator<U>&) throw();
    ~allocator() throw();

    T *address(T &x) const;
    const_T *address(const_T &x) const;

    T *allocate(size_t, allocator<void>::const_T *hint = 0);
    void deallocate(T *p, size_t n);

    size_t max_size() const throw();
    void construct(T *p, const T& val);
    void destroy(T *p);
    };
```

Exemple :

```
                // utilise l'allocateur par défaut:
vector<double> V(100, 5.0);
                // utilise un allocateur particulier (appelé pthread_alloc):
vector<double, pthread_alloc> local(V.begin(), V.end());
```

9.2 Conteneurs

9.2.1 Notion

Les conteneurs sont des objets qui représentent des collections d'objets.

Les fichiers en-tête concernés sont :

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

Il y a deux familles principales de conteneurs : les *séquences* et les *conteneurs associatifs*.

1. Les *séquences* donnent à un ensemble fini d'objets de même type une organisation linéaire. Trois sortes de séquences de base sont fournies : les *vecteurs* (**`vector`**), les *listes* (**`list`**) et les *files à double entrée* (**`deque`**). Chacune représente un compromis espace/temps optimal pour la complexité des opérations auxquelles elle est destinée en priorité.

En association avec les séquences, la bibliothèque fournit des *adaptateurs*, qui sont des interfaces permettant d'utiliser les séquences pour implémenter des types de données abstraits : les piles (**`stack`**), les files (**`queue`**) et les files de priorité (**`priority_queue`**).

2. Les *conteneurs associatifs* gèrent des collections de valeurs, chacune associée à une clé qui en est le moyen d'accès. Les notions de premier, dernier, $n^{\text{ème}}$ élément n'ont pas cours ici. Dans les ensembles (**`set`**) et les ensembles multiples (**`multiset`**) la clé et la valeur sont la même chose. Au contraire, dans les tables associatives (**`map`**) et les tables associatives multiples (**`multimap`**) la clé et la valeur associée sont séparées.

3. Enfin, le cas particulier des conteneurs de bits est pris en charge par deux types particuliers : les vecteurs de bits (**`vector<bool>`**) et les ensembles de bits (**`bitset`**).

9.2.2 Eléments communs à tous les conteneurs

NOTE. La documentation officielle des conteneurs est très importante et comporte beaucoup de « redites », de nombreux membres étant communs à plusieurs sortes de conteneurs, voire à tous les conteneurs.

Nous allons dons nous contenter d'expliquer ici les propriétés et les membres les plus utiles, sachant que tous ne concernent pas forcément tous les conteneurs. Nous renvoyons aux documents officiels pour la spécification précise de chaque membre de chaque modèle.

Propriétés et contraintes

1. Les conteneurs « possèdent » leurs éléments. Première conséquence : lorsqu'un conteneur est détruit, ses éléments le sont également. Cette contrainte signifie que les valeurs introduites dans les conteneurs sont copiées au moment de leur introduction dans le conteneur ; lorsque celui-ci est détruit, ces copies sont détruites.

Les valeurs introduites dans un conteneur peuvent être des pointeurs sur des objets ; la destruction du conteneur entraîne alors uniquement la destruction des pointeurs, non des objets pointés.

Deuxième conséquence : lorsqu'un conteneur est copié, ses éléments le sont également. Il faut le savoir, et n'effectuer des copies de conteneurs que lorsque c'est inévitable.

2. Les éléments des conteneurs doivent supporter la construction par copie et l'affectation.

La construction par copie est utilisée dès l'introduction de l'objet dans le conteneur (voir point précédent) ; l'affectation est utilisée par les fonctions qui modifient les conteneurs.

3. Tous les conteneurs possèdent les fonctions membres **`size_t size()`** et **`bool empty()`** qui calculent et renvoient, en un temps constant, le nombre d'éléments de la structure et le fait que cette dernière est vide.

4. Tous les conteneurs possèdent les fonctions membres **`begin()`**, qui rend un itérateur positionné sur le premier élément du conteneur, et **`end()`**, qui rend un itérateur positionné immédiatement après le dernier élément.

Si l'itérateur associé à un conteneur (renvoyé par **`begin()`**) est un itérateur bidirectionnel ou d'accès aléatoire (cf. § 9.4) le conteneur est appelé *réversible*.

L'itérateur associé à un conteneur associatif (celui renvoyé par **`begin()`**) est bidirectionnel.

5. Si l'appel d'une fonction d'insertion (**`insert()`**, **`push_back()`** ou **`push_front()`**) sur un conteneur lance une exception, la fonction est sans effet.

Aucune des fonctions **`erase()`**, **`pop_back()`** ou **`pop_front()`** ne lance des exceptions.

Aucun constructeur par copie ou opérateur d'affectation d'un conteneur ne lance des exceptions.

Les fonctions **`swap()`** (échange) ne lancent pas d'exceptions, autres que celles que pourrait lancer le constructeur par copie ou l'opérateur d'affectation de l'objet **`Compare`** associé au conteneur, et n'invalident aucune référence, pointeur ou itérateur faisant référence aux éléments du conteneur qui sont en cours d'échange.

Membres communs

Parmi les fonctions membres des conteneurs les plus courantes on trouve les suivantes (toutes ne sont pas définies pour tous les conteneurs) :

`bool empty() const;`

Le conteneur est-il vide ?

`size_t size() const;`

Nombre d'éléments dans le conteneur.

`void resize(size_t nouvelle_taille, T valeur);`

Changement de la taille du conteneur. Dans le cas d'un agrandissement, la **`valeur`** indiquée (par defaut : la valeur par défaut du type **`T`**) est utilisée pour garnir les nouveaux emplacements.

`size_t capacity() const;`

Nombre d'éléments que le conteneur peut maintenir sans nécessiter une réallocation.

`void reserve(size_t n);`

Modifie la taille de l'espace réservé pour le conteneur (celui dont la taille est donnée par **`capacity`**)

`size_t max_size() const;`

Nombre maximum d'éléments que le conteneur peut contenir.

T &operator[](*clé* **);**
const T &operator[](*clé* **) const;**
T &at(*clé* **);**
const T &at(*clé* **) const;**

Accès indexé (dans le cas de **map**, *clé* est du type précisé lors de l'instanciation du modèle ; dans les autres cas, *clé* d'un type entier).

T &front();
const T &front() const;

Accès à l'élément qui est en tête.

T &back();
const T &back() const;

Accès à l'élément qui est en queue.

void push_front(const T &);
void push_back(const T &);

Ajout d'un élément en tête [resp. en queue].

void pop_front();
void pop_back();

Suppression de l'élément en tête [resp. en queue].

itérateur **insert(** *itérateur* **, const T& x);**
void insert(*itérateur* **, size_t n, const T& x);**

Insertion d'une valeur **x** [resp. de **n** copies d'une valeur **x**].

iterator **erase(** *itérateur* **);**
void clear();

Suppression d'une valeur [resp. de toutes les valeurs].

itérateur **begin();**
const *itérateur* **begin() const;**
itérateur **end();**
const *itérateur* **end() const;**

Itérateur positionné sur le premier [resp. le dernier] élément.

itérateur **rbegin();**
const *itérateur* **rbegin() const;**
itérateur **rend();**
const *itérateur* **rend() const;**

Itérateur inverse positionné sur le premier [resp. le dernier] élément.

void swap(vector<T> &v);

Echange des valeurs des vecteurs ***this** et **v**.

EXEMPLE. A propos de la création et la destruction des éléments des conteneurs :

```
struct Point
    {
    int x, y;
    Point(int a = 0, int b = 0) : x(a), y(b)
        { cout << "\tPoint(" << a << "," << b << ") [" << this << "]\n"; }
    Point(const Point &p) : x(p.x), y(p.y)
        { cout << "\tPoint(Point &) [" << this << " <-- "
                                            << (void *)&p << "]\n"; }
    ~Point()
        { cout << "\t~Point() [" << this << "]\n"; }
    };
```

```
void uneFonction(vector<Point> v)
    {
    for (int i = 0; i < v.size(); i++)
        cout << '(' << v[i].x << ',' << v[i].y << ") ";
    cout << '\n';
    }

void main()
    {
    cout << "Debut du programme\n";
    Point a(1, 2), b(3, 4), c(5, 6);

    cout << "Création du vecteur\n";
    vector<Point> *v = new vector<Point>;
    v->push_back(a);
    v->push_back(b);
    v->push_back(c);

    cout << "Appel d'une fonction\n";
    uneFonction(*v);

    cout << "Destruction du vecteur\n";
    delete v;

    cout << "Fin du programme\n";
    }
```

Affichage obtenu :

```
Debut du programme
   Point(1,2) [0066FDFC]
   Point(3,4) [0066FDF4]
   Point(5,6) [0066FDEC]
Création du vecteur
   Point(Point &) [006842B0 <-- 0066FDFC]
   Point(Point &) [006842B8 <-- 0066FDF4]
   Point(Point &) [006842C0 <-- 0066FDEC]
Appel d'une fonction
   Point(Point &) [006846B4 <-- 006842B0]
   Point(Point &) [006846BC <-- 006842B8]
   Point(Point &) [006846C4 <-- 006842C0]
(1,2) (3,4) (5,6)
   ~Point() [006846B4]
   ~Point() [006846BC]
   ~Point() [006846C4]
Destruction du vecteur
   ~Point() [006842B0]
   ~Point() [006842B8]
   ~Point() [006842C0]
Fin du programme
   ~Point() [0066FDEC]
   ~Point() [0066FDF4]
   ~Point() [0066FDFC]
```

EXEMPLE. A propos de **size**, **capacity**, **max_size** :

```
void main()
    {
    vector<int> v;

    for (int i = 0; i < 259; i++)
        {
        cout << "size: " << v.size() << " -- capacity: " << v.capacity()
                                    << " -- max: " << v.max_size() << "\n";
        v.push_back(i);
        }
    }
```


Affichage obtenu :

```
size: 0 -- capacity: 0 -- max: 1073741823
size: 1 -- capacity: 256 -- max: 1073741823
size: 2 -- capacity: 256 -- max: 1073741823
...
size: 256 -- capacity: 256 -- max: 1073741823
size: 257 -- capacity: 512 -- max: 1073741823
size: 258 -- capacity: 512 -- max: 1073741823
```

9.2.3 Séquences : vector, list, deque

vector

Le conteneur **`vector`** est le type de séquence qui doit être utilisé à défaut d'informations plus précises.

Un vecteur offre l'accès direct à ses éléments. Il offre également l'insertion et la suppression à la fin, en un temps constant. L'insertion et la suppression ailleurs qu'à la fin prennent un temps linéaire.

Le nombre d'éléments d'un vecteur n'est pas fixé lors de la construction de ce dernier. Le vecteur gère automatiquement l'accroissement de la mémoire occupée, lorsque cela est nécessaire.

list

Le type **`list`** doit être utilisé lorsque le conteneur est destiné à subir principalement des insertions et des effacements ailleurs qu'à ses extrémités.

Une liste est une sorte de séquence qui offre des itérateurs bidirectionnels (la liste peut être parcourue dans les deux sens) et permet l'insertion et l'effacement d'éléments n'importe où dans la liste en un temps constant, avec une gestion automatique de l'espace utilisé.

Contrairement aux vecteurs et aux files à double entrée (**`deque`**) les listes n'offrent pas l'accès rapide à un élément quelconque.

Les listes possédent quelques membres tout à fait spécifiques :

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

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

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

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

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

Chaque groupe d'éléments consécutifs l_a, l_{a+1}, … l_b vérifiant $l_i = l_{i+1}$ (premier cas) ou **`pred`**(l_i, l_{i+1}) (deuxième cas) est remplacé par l_a, le premier des éléments du groupe.

```
void sort();
template <class Compare> void sort(Compare comp);

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

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

```
void reverse();
```

Inversion d'une liste.

EXEMPLE. Opérations sur les listes :

```
void main()
    {
    list<int> L;

    for (int i = 0; i < 16; i++)
        L.push_back(i / 4);
    for (int i = 0; i < 16; i++)
        L.push_back(i % 4);
    cout << L << '\n';

    L.remove(3);
    cout << L << '\n';

    L.unique();
    cout << L << '\n';
    }
```

Affichage obtenu :

```
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
0 0 0 0 1 1 1 1 2 2 2 2 0 1 2 0 1 2 0 1 2 0 1 2
0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```

EXEMPLE. Opérations sur les listes : tri et fusion.

```
void main()
    {
    list<int> L1, L2;

    for (int i = 0; i < 10; i++)
        L1.push_back(rand() % 100);
    for (int i = 0; i < 10; i++)
        L2.push_back(rand() % 100);
    cout << "L1: " << L1 << "\nL2: " << L2 << '\n';

    L1.sort();
    L2.sort();
    cout << "L1: " << L1 << "\nL2: " << L2 << '\n';

    L1.merge(L2);
    cout << "L1: " << L1 << "\nL2: " << L2 << '\n';
    }
```

Affichage obtenu (remarquer le sort fait à **L2** par la fonction **merge**) :

```
L1: 46 30 82 90 56 17 95 15 48 26
L2: 4 58 71 79 92 60 12 21 63 47
L1: 15 17 26 30 46 48 56 82 90 95
L2: 4 12 21 47 58 60 63 71 79 92
L1: 4 12 15 17 21 26 30 46 47 48 56 58 60 63 71 79 82 90 92 95
L2:
```

NOTE. L'injection d'une liste dans un flux de sortie n'est pas une opération prédéfinie. Voici comment on peut la définir :

```
template<class T> ostream &operator<<(ostream &o, list<T> &l)
    {
    list<T>::iterator i = l.begin(), j = l.end();
    while (i != j)
        o << *i++ << ' ';
    return o;
    }
```

deque

Le conteneur **deque** est le meilleur choix lorsque beaucoup d'insertions et suppressions seront effectués au début et à la fin du conteneur.

Une *file à double entrée*, ou **deque**, est une structure qui, comme un vecteur, offre l'accès direct à ses membres en un temps constant (amorti). L'insertion et la suppression au début et à la fin de la structure se font en un temps constant ; les insertions et suppressions à d'autres endroits se font en un temps qui, en moyenne, dépend linéairement du nombre d'éléments de la structure.

Le principal avantage des files à double entrée par rapport aux vecteurs est que les insertions *au début* s'y font en un temps constant (dans un vecteur elles prennent un temps linéaire).

Le principal inconvénient des files à double entrée par rapport aux vecteurs est l'absence des fonctions membres **capacity** et **reserve** et, dans le même ordre d'idées, le fait que les insertions dans une telle file invalident les itérateurs couramment définis.

9.2.4 Types abstraits : stack, queue, priority_queue

Les *adaptateurs* permettent de définir une nouvelle interface sur un conteneur, pour lui donner le comportement correspondant à un type abstrait parmi :

- **stack**, type de données dans lequel l'opération d'extraction fournit l'élément le plus récemment inséré,
- **queue**, type de données dans lequel l'opération d'extraction fournit l'élément le moins récemment inséré,
- **priority_queue**, type de données dans lequel l'opération d'extraction fournit l'élément ayant la plus grande priorité (déterminé par une opération **Compare**, argument du modèle).

Les adaptateurs sont des modèles qui prennent un conteneur comme argument-type ; leurs constructeurs prennent également pour argument un conteneur (dans l'un et l'autre cas, des valeurs par défaut sont fournies).

stack

Le conteneur sous-jacent peut être n'importe quel conteneur supportant les opérations **back**, **push_back** and **pop_back** ; les types **vector** et **deque** (valeur par défaut) conviennent parfaitement.

Les opérations principales des piles sont **push**, **top** et **pop** :

```
template <class T, class Container = deque<T> >
class stack
    {
public:
    typedef typename Container::value_type value_type;
    typedef typename Container::size_type size_type;
    typedef Container container_type;

protected:
    Container c;

public:
    explicit stack(const Container& = Container());

    bool empty() const;
    size_type size() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
    };
```

queue

Le conteneur sous-jacent peut être n'importe quel conteneur supportant les opérations **front**, **back**, **push_back** and **pop_front** ; le type **deque** (valeur par défaut) est presque toujours le meilleur choix.

Les opérations principales des files sont **push**, **front** et **pop** :

```
template <class T, class Container = deque<T> >
class queue
    {
public:
    typedef typename Container::value_type value_type;
    typedef typename Container::size_type size_type;
    typedef Container container_type;

protected:
    Container c;

public:
    explicit queue(const Container& = Container());

    bool empty() const;
    size_type size() const;
    value_type& front();
    const value_type& front() const;
    value_type& back();
    const value_type& back() const;
    void push(const value_type& x);
    void pop();
    };
```

priority_queue

Le conteneur sous-jacent peut être n'importe quel conteneur supportant les opérations **front**, **push_back** and **pop_back**. Les types **vector** (valeur par défaut) et **deque** conviennent.

La notion de priorité est réalisée par l'objet fonctionnel **Compare**, argument du modèle, dont il est supposé qu'il définit un *ordre strict faible* (cf. au § 9.2.5, la section *A propos de compare*).

Les opérations principales sont **push**, **top** et **pop** :

```
template <class T, class Container = vector<T>,
            class Compare = less<typename Container::value_type> >
class priority_queue
    {
public:
    typedef typename Container::value_type value_type;
    typedef typename Container::size_type size_type;
    typedef Container container_type;

protected:
    Container c;
    Compare comp;

public:
    explicit priority_queue(const Compare& x = Compare(),
                            const Container& = Container());

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    const value_type& top() const { return c.front(); }
    void push(const value_type& x);
    void pop();
    };
```

EXEMPLE, dans lequel la priorité est donnée par l'ordre existant sur les données (des entiers) :

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

void main()
    {
    int x, i;
    priority_queue<int> p;
```

```
    for (i = 0; i < 16; i++)
        {
        cout << (x = rand() % 100) << ' ';
        p.push(x);
        }
    cout << '\n';

    while ( ! p.empty())
        {
        cout << p.top() << ' ';
        p.pop();
        }
    cout << '\n';
}
```

Affichage obtenu :

```
46 30 82 90 56 17 95 15 48 26 4 58 71 79 92 60
95 92 90 82 79 71 60 58 56 48 46 30 26 17 15 4
```

Il ne faut pas voir une file de priorité comme une méthode de tri mais comme une manière efficace de réaliser une structure dont les éléments sortent de manière ordonnée : si le conteneur sous-jacent permet l'accès indexé en temps constant (comme un **`vector`**), le opérations **`push`** et **`pop`** sur une **`priority_queue`** demandent un temps en log *n*.

EXEMPLE. L'utilisateur peut fournir sa propre notion de priorité. Dans le programme suivant, tous les nombres pairs sont tenus pour inférieurs à tous les impairs :

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

struct moindre : public binary_function<int, int, bool>
    {
    bool operator()(int a, int b)
        {
        if (a % 2 == 0)
            return b % 2 != 0 ? true : a < b;
        else
            return b % 2 == 0 ? false : a < b;
        }
    };

void main()
    {
    int x, i;
    priority_queue<int, vector<int>, moindre> p;

    for (i = 0; i < 16; i++)
        {
        cout << (x = rand() % 100) << ' ';
        p.push(x);
        }
    cout << '\n';

    while ( ! p.empty())
        {
        cout << p.top() << ' ';
        p.pop();
        }
    cout << '\n';
    }
```

Affichage obtenu :

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

9.2.5 Conteneurs associatifs : set, map

Chaque conteneur associatif est paramétré par le type de la clé, **`Key`**, et par celui de la relation de comparaison, **`Compare`**.

La propriété fondamentale d'un conteneur associatif est de permettre l'accès à un élément à partir d'une valeur de clé. On peut imaginer le conteneur comme formé de paires <*clé*, *valeur*> et l'opération d'accès comme la recherche d'une paire dont la clé est *égale*, ou plutôt *équivalente*, à une valeur donnée.

Il faut préciser ce qu'on entend par *équivalente* :

A propos de compare

Le foncteur binaire **`Compare`** définissant le conteneur doit induire un *ordre strict faible* sur les éléments de **`Key`**, l'ensemble des valeurs de clés possibles. *Strict* signifie que la relation n'est pas réflexive ($\forall x, \neg compare(x, x)$) ; *faible* signifie qu'il s'agit d'une relation plus générale qu'un ordre total, mais moins qu'un ordre partiel.

Plus précisément, définissant *equiv* par

$$equiv(x, y) \Leftrightarrow \neg\ \mathbf{Compare}(x, y) \wedge \neg\ \mathbf{Compare}(y, x)$$

les relations **`Compare`** et *equiv* sont transitives :

$$\forall x\ \forall y\ \forall z \qquad \mathbf{Compare}(x, y) \wedge \mathbf{Compare}(y, z) \Rightarrow \mathbf{Compare}(x, z)$$

$$\forall x\ \forall y\ \forall z \qquad equiv(x, y) \wedge equiv(y, z) \Rightarrow equiv(x, z)$$

Sous ces conditions on peut montrer que

- *equiv* est une relation d'équivalence,
- **`Compare`** induit une relation d'ordre strict total sur l'ensemble des classes pour la relation *equiv*.

La relation *equiv* est à la base des conteneurs associatifs : deux clés c_1 et c_2 sont tenues pour équivalentes si $equiv(c_1,c_2)$ est vraie.

On notera que, par défaut, **`Compare`** est la relation **`less`** définie sur le type **`Key`**, dont la valeur par défaut est **`operator<`**, ce qui a généralement pour conséquence que la valeur par défaut de *equiv* est **`==`**.

set et multiset

Un ensemble (**`set`**) est un conteneur associatif dans lequel la clé et la valeur d'un élément sont la même chose. Les opérations de modification fondamentales sont **`insert`** et **`erase`**. Les opérations utilisant la notion de position d'un élément dans la structure (**`[]`**, **`at`**, **`push_front`**, **`push_back`**, **`pop_front`**, **`pop_back`**) ne sont pas définies pour les ensembles.

```
template <class Key, class Compare = less<Key>,
                                class Allocator = allocator<Key> >
class set
    {
public:
    ...
    explicit set(const Compare& comp = Compare(),
                                       const Allocator& = Allocator());
    ...
    iterator begin();
    iterator end();
    ...
    pair<iterator, bool> insert(const Key& x);
    ...
    size_t erase(const Key& x);
    void erase( iterator_position );
    ...
    };
```

Un conteneur de type **`set`** ne conserve pas deux éléments *équivalents*.

Exemple :

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

void main()
    {
    int x, i;
    set<int> s;

    for (i = 0; i < 16; i++)
        {
        cout << (x = rand() % 10) << ' ';
        s.insert(x);
        }
    cout << '\n';

    set<int>::iterator i0, i1;
    for (i0 = s.begin(), i1 = s.end(); i0 != i1; i0++)
        cout << *i0 << ' ';
    cout << '\n';
    }
```

Affichage obtenu :

```
1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1
0 1 2 4 5 7 8 9
```

Un conteneur de type **multiset** conserve les éléments équivalents. Exemple :

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

void main()
    {
    int x, i;
    multiset<int> s;

    for (i = 0; i < 16; i++)
        {
        cout << (x = rand() % 10) << ' ';
        s.insert(x);
        }
    cout << '\n';

    multiset<int>::iterator i0, i1;
    for (i0 = s.begin(), i1 = s.end(); i0 != i1; i0++)
        cout << *i0 << ' ';
    cout << '\n';
    }
```

Affichage obtenu :

```
6 0 2 0 6 7 5 5 8 6 4 8 1 9 2 0
0 0 0 1 2 2 4 5 5 6 6 6 7 8 8 9
```

map et multimap

Une *table associative* (**map**), appelée parfois *dictionnaire*, est une collection de couples (*clé*, *valeur*) offrant l'accès à un élément à partir de la valeur de sa clé.

Pas moins de quatre arguments sont nécessaires pour instancier le modèle **map** :

```
template <class Key, class T, class Compare = less<Key>,
                    class Allocator = allocator<pair<const Key, T> > >
class map
    {
public:
        // types
    typedef Key key_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;
    typedef Compare key_compare;
    typedef Allocator allocator_type;

    ...

        // accès aux éléments
    T& operator[](const key_type& x);

    itérateur lower_bound(const key_type& x);
    itérateur_constant lower_bound(const key_type& x) const;
    itérateur upper_bound(const key_type& x);
    itérateur_constant upper_bound(const key_type &x) const;

    ...
    };
```

La principale opération sur les tables associatives, l'accès à un élément, est disponible à travers l'opérateur **[]**, dont l'argument est du type des clés. Cet opérateur met en évidence la principale similarité entre les tableaux et les tables associatives : l'accès « direct » à une valeur à partir d'une clé (numérique dans le cas d'un tableau, quelconque dans le cas d'une table associative).

Les tables associatives simples (**map**) ne conservent pas plusieurs éléments dont les clés sont équivalentes.

Exemple :

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

template<class A, class B>
    ostream &operator<<(ostream &o, pair<A, B> &p)
        { return o << '<' << p.first << ',' << p.second << '>'; }

void main()
    {
    map<string, int> m;

    m["Bernard"] = 10;
    m["Denis"  ] = 20;
    m["Andre"  ] = 30;
    m["Charles"] = 40;

    m["Andre"  ] = 50;
    m["Bernard"] = 60;
    m["Andre"  ] = 70;

    cout << m.size() << '\n' << m["Bernard"] << ' '
        << m["Denis"] << ' ' << m["Andre"] << ' ' << m["Charles"] << '\n';
```



```
    map<string, int>::iterator courant = m.begin();
    map<string, int>::iterator dernier = m.end();
    while (courant != dernier)
        cout << *courant++ << ' ';
    cout << '\n';
    }
```

Affichage obtenu :

```
4
60 20 70 40
<Andre,70> <Bernard,60> <Charles,40> <Denis,20>
```

Les tables associatives multiples (**multimap**) peuvent contenir plusieurs éléments dont les clés sont équivalentes. L'opérateur **[]** n'est plus disponible ; l'accès aux éléments en devient plus complexe :

```
void main()
    {
    multimap<string, int> m;

    m.insert(make_pair(string("Bernard"), 10));
    m.insert(make_pair(string("Denis"  ), 20));
    m.insert(make_pair(string("Andre"  ), 30));
    m.insert(make_pair(string("Charles"), 40));
    m.insert(make_pair(string("Andre"  ), 50));
    m.insert(make_pair(string("Bernard"), 60));
    m.insert(make_pair(string("Andre"  ), 70));

    cout << m.size() << '\n';

    multimap<string, int>::iterator p = m.lower_bound("Andre");
    multimap<string, int>::iterator q = m.upper_bound("Andre");
    while (p != q)
        cout << *p++ << ' ';
    cout << '\n';
    }
```

Affichage obtenu :

```
7
<Andre,30> <Andre,50> <Andre,70>
```

9.2.6 Conteneurs de bits : vector<bool>, bitset

La classe **vector<bool>**, spécialisation de **vector**, et le modèle **bitset** procèdent de la même idée : minimiser l'encombrement de conteneurs dont les éléments sont les plus simples possibles (**false** et **true**, **0** et **1**).

vector<bool>

L'interface est celle d'un **vector**, l'espace est minimisé. Exemple (intéressant ?), le calcul des nombres premiers inférieurs à un nombre donné :

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

void main(int argc, char **argv)
    {
    vector<bool> b;
    int n, i, j;

    n = atoi(argv[1]);
    b.push_back(0);     // 0
    b.push_back(0);     // 1
```

```
    for (j = 2; j < n; j++)
        {
        for (i = 2; i < j; i++)
            if (b[i] && ((j % i) == 0))
                break;
        b.push_back(i >= j);
        }

    for (j = 2; j < n; j++)
        if (b[j])
            cout << j << ' ';
    }
```

bitset

Un bitset est un tableau de bits rangé de manière compacte et principalement destiné à tirer profit des opérations logiques bit à bit disponibles sur la machine utilisée.

On notera que la taille d'un bitset est nécessairement connue à la compilation.

```
template<size_t N> class bitset
    {
public:

    class reference
        {
        friend class bitset;
    public:
        reference& operator=(bool x);               // pour b[i] = x;
        reference& operator=(const reference&);     // pour b[i] = b[j];
        bool operator~() const;                     // bascule d'un bit
        operator bool() const;                      // pour x = b[i];
        reference& flip();                          // pour b[i].flip();
        };
            // construction
    bitset();
    bitset(unsigned long val);
    explicit bitset(string s, size_t pos, size_t nbr)

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

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

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

            // accès et conversion
    reference operator[](size_t pos);
    unsigned long to_ulong() const;
    string to_string() const;
```



```
            // tests
        bool operator==(const bitset<N>& rhs) const;    // égal
        bool operator!=(const bitset<N>& rhs) const;    // différent
        bool test(size_t pos) const;                    // LE bit est 1
        bool any() const;                               // UN bit est 1
        bool none() const;                              // AUCUN bit est 1
        };
```

Exemple :

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

template<int N> ostream &operator<<(ostream &o, bitset<N> b)
    {
    for (int i = 0; i < N; i++)
        o << (b.test(i) ? '1' : '0');
    return o;
    }

void main()
    {
    bitset<30> a, b;

    a.reset();
    for(int i = 0; i < 30; i += 2)
        a.set(i);
    b.reset();
    for(int i = 0; i < 30; i += 3)
        b.set(i);

    cout << "  a   : " << a << "\n" << "  b   : " << b << "\n"
        << "a * b : " << (a & b) << "\n"
        << "a + b : " << (a | b) << "\n";
    }
```

Affichage obtenu :

```
  a   : 101010101010101010101010101010
  b   : 100100100100100100100100100100
a * b : 100000100000100000100000100000
a + b : 101110101110101110101110101110
```

9.3 Itérateurs

Fichier en-tête :

```
<iterator>
```

9.3.1 Notion

Un itérateur est un outil pour accéder aux membres d'une collection de données, généralement pour la parcourir. C'est une abstraction, caractérisée par les éléments suivants :

- un itérateur (s'il est valide) est constamment placé « sur » un certain *élément courant* de la collection, auquel on accède par les opérateurs d'indirection ***** et **->**
- une opération, représentée par les deux formes, préfixée (**++***i*) et postfixée (*i***++**) de l'opérateur **++**, place l'itérateur sur l'*élément suivant* celui sur lequel il était placé avant cette opération,
- la relation d'égalité entre itérateurs, notée **==**, est définie et permet en particulier de comparer un itérateur à des valeurs particulières, comme le début ou la fin de la séquence.

Un itérateur est une abstraction pure : tout ce qui se comporte comme un itérateur *est* un itérateur. Un pointeur ordinaire est donc un itérateur. Inversement, tout itérateur est une sorte de pointeur généralisé, ce qui justifie l'utilisation des opérateurs et du vocabulaire des pointeurs ; ainsi, on parle de l'*élément pointé par un itérateur*.

Toute collection de données qu'on peut voir comme une séquence est susceptible d'être accédée par un itérateur. En fait, « collection de données qui peut être accédée par un itérateur » est une bonne définition de la notion abstraite de séquence.

Les tableaux du langage, les conteneurs de la bibliothèque peuvent tous être accédés à travers des itérateurs (aussi longtemps qu'ils sont manipulés à l'aide d'itérateurs, même les ensembles, les tables associatives et les files de priorité présentent une organisation séquentielle). Les itérateurs constituent donc une interface uniforme qui permet d'implémenter les algorithmes de la bibliothèque (cf. § 9.4) d'une manière tout à fait indépendante de la structure de données sur laquelle ils agissent.

Il y a cinq catégories d'itérateurs :

- Itérateurs de sortie (*output itérator*). Ces itérateurs supportent les écritures (expression **`*i`** en position de *lvalue*), el l'avancement (opération **`++`**).
- Itérateurs d'entrée (*input iterator*). Ces itérateurs supportent la lecture (expression **`*i`**, en position de *rvalue*, et opération **`i->`**), l'avancement (opération **`i++`**) et l'égalité (opérations **`==`** et **`!=`**).
- Itérateurs en avant (*forward itérator*). Ces itérateurs sont d'entrée et de sortie ; ils supportent la lecture, l'écriture, l'avancement et l'égalité.
- Itérateurs bidirectionnels (*bidirectional iterator*). Comme les précédents, avec en plus le recul (opération **`--`**).
- Itérateurs d'accès aléatoire (*random access iterator*). Ces itérateurs possédent les opérations des itérateurs bidirectionnels, et en plus, l'accès direct à un élément (opération **`[]`**), l'addition et la soustraction d'un entier (opérations **`+`**, **`-`**, **`+=`** et **`-=`**), la soustraction de deux itérateurs (opération **`-`**) et une relation d'ordre (opérations **`<`**, **`<=`**, **`>`** et **`>=`**).

Ces catégories sont hiérarchisées : les itérateurs en avant peuvent être utilisés là où des itérateurs d'entrée ou de sortie sont requis ; les itérateurs bidirectionnels peuvent être utilisés là où des itérateurs d'avance sont requis, et ainsi de suite. Noter que les (vrais) pointeurs sont considérés comme des itérateurs d'accès aléatoire, et peuvent donc prendre la place de n'importe quel autre itérateur.

Les itérateurs peuvent être *constants* et *non constants*. Un objet accédé à travers un itérateur constant ne peut pas être modifié.

Itérateurs et conteneurs

Les conteneurs de la bibliothèque se chargent de définir leurs propres itérateurs associés :

- le type de l'itérateur est un type imbriqué dans la classe conteneur,
- la classe conteneur fournit des fonctions membres pour créer des valeurs d'itérateur intéressantes (**`begin()`**, **`end()`**, etc.)

Exemple :

```
list<string> l;                 // l est une liste
list<string>::iterator p;       // p est un itérateur
...
p = find(l.begin(), l.end(), string("Henri"));
if (p == l.end())
    cout << "Elément absent\n";
else
    {
    exploitation du résultat de la recherche
    }
```

Notez que **`begin()`** fournit un itérateur positionné sur le premier élément de la séquence, alors que **`end()`** renvoie un itérateur positionné sur un élément fictif qui suivrait immédiatement le dernier élément de la séquence.

On raisonne de la manière suivante : si le conteneur t était un tableau déclaré « **`int T[N];`** » alors **`t.begin()`** serait la valeur **`T`** et **`t.end()`**serait la valeur **`T + N`**.

Exemple : l'algorithme **find** de la bibliothèque standard :

```
template<class inIter, class T>
inIter find(inIter premier, inIter dernier, const T &valeur)
    {
    while (premier != dernier && *premier != valeur)
        ++premier;
    return premier;
    }
```

9.3.2 Programmation avec des itérateurs

Le modèle **iterator_traits** permet l'implantation d'algorithmes en termes d'itérateurs uniquement, sans mention de la structure de données sous-jacente :

```
template<class Iterator> struct iterator_traits
    {
    typedef typename Iterator::difference_type difference_type;
    typedef typename Iterator::value_type value_type;
    typedef typename Iterator::pointer pointer;
    typedef typename Iterator::reference reference;
    typedef typename Iterator::iterator_category iterator_category;
    };
```

le principal intérêt de ce modèle est qu'il cohabite avec une spécialisation qui permet d'accepter des pointeurs comme itérateurs :

```
template<class T> struct iterator_traits<T *>
    {
    typedef ptrdiff_t difference_type;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef random_access_iterator_tag iterator_category;
    };
```

Exemple : l'algorithme **count()**de la bibliothèque standard compte le nombre d'occurrences d'une valeur donnée dans une séquence :

```
template<class Iter, class T>
    typename iterator_traits<Iter>::difference_type
                        count(Iter debut, Iter fin, const T &val)
        {
        typename iterator_traits<Iter>::difference_type result = 0;
        while (debut != fin)
            if (*debut++ == val)
                result++;
        return result;
        }
```

Essai :

```
void main()
    {
    list<int> l;

    for (int i = 0; i < 1000; i++)
        l.push_back(rand() % 100);

    cout << count(l.begin(), l.end(), 50) << "\n";
    }
```

9.3.3 Itérateurs prédéfinis

La norme ISO prévoit un certain nombre d'itérateurs prédéfinis, construits au-dessus d'autres éléments de la bibliothèque.

Itérateurs d'insertion

En général, les opérations qui écrivent des données sur des itérateurs de sortie, comme

```
fill_n(v.begin(), 100, -1);
```

(remplir **v** avec 100 fois la valeur **-1**) supposent que l'itérateur pointe un emplacement valide et que l'information qui se trouve à cet emplacement peut être écrasée. Ainsi, notre exemple sera techniquement erroné si le tableau **v** n'a pas au moins 100 éléments, et il sera logiquement erroné si les 100 premières valeurs doivent être préservées.

Les *itérateurs d'insertion* sont des itérateurs de sortie qui prennent en charge l'augmentation de la structure de données sous-jacente. Il y en a de trois variétés:

- les itérateurs d'insertion à la fin (back_insert_iterator) sont créés par la fonction

```
template<class Conteneur>
    back_insert_iterator<Conteneur> back_inserter(Conteneur &c);
```

- les itérateurs d'insertion au début (front_insert_iterator) sont créés par la fonction

```
template<class Conteneur>
    front_insert_iterator<Conteneur> front_inserter(Conteneur &c);
```

- les itérateurs d'insertion à une position donnée (insert_iterator) sont créés par la fonction

```
template<class Conteneur, class IterOut>
    insert_iterator<Conteneur> inserter(Conteneur &c, IterOut p);
```

Exemple : la création d'une liste par copie d'une autre :

```
list<int> a, b;
```

création des éléments de la liste **a**

```
copy(a.begin(), a.end(), back_inserter(b));
```

Donnons une idée de la définition de ces modèles. Par exemple, le modèle **insert_iterator** :

```
template<class Conteneur> class insert_iterator
    {
protected:
    Conteneur &cont;
    typename Conteneur::iterator pos;

public:
    explicit insert_iterator(Conteneur &c,
        typename Conteneur::iterator p) : cont(c), pos(p)  { }

    insert_iterator &operator=(const typename Conteneur::value_type valeur)
        {
        cont.insert(pos, val);
        ++pos;
        return *this;
        }

    insert_iterator &operator*()       { return *this; }
    insert_iterator &operator++()      { return *this; }
    insert_iterator &operator++(int)   { return *this; }
    };
```

Itérateurs de flux

Un itérateur de flux de sortie permet de faire apparaître un tel flux comme une séquence que l'on manipule à travers des itérateurs :

```
template<class T, class C = char> struct ostream_iterator
    {
    typedef basic_ostream<C> ostream_type;

    ostream_iterator(ostream_type &s);
    ostream_iterator(ostream_type &s, const C *delimiteur);
    ~ostream_iterator();
                                // écriture de valeur en sortie
    ostream_iterator &operator=(const T &valeur);

    ostream_iterator &operator*();
    ostream_iterator &operator++();
    ostream_iterator &operator++(int);
    ...
    };
```

Bien entendu, un itérateur de flux de sortie est un itérateur de sortie.

Exemple. Création :

```
ostream_iterator<int> os(cout, " + ");
```

Ecritures individuelles:

```
*os = 1;                // écriture de 1 +
++os;                   // préparation pour une nouvelle sortie
*os = 2;                // écriture de 2 +
++os;                   // etc.
```

Beaucoup plus intéressant, l'affichage en un coup de toute une séquence :

```
list<int> L;
...
copy(L.begin(), L.end(), os);
```

On peut de la même manière utiliser des itérateurs de flux d'entrée :

```
template<class T, class C = char> struct istream_iterator
    {
    typedef basic_istream<C> istream_type;

    istream_iterator();         // la fin de l'entrée
    istream_iterator(istream_type &s);
    ~istream_iterator();

    const T &operator*() const;
    const T *operator->() const;
    istream_iterator &operator++();
    istream_iterator operator++(int);
    ...
    };
```

Exemple. Création :

```
istream_iterator<int> is(cin);
```

Ecritures individuelles:

```
x = *is;                // lecture d'un entier
++is;                   // préparation pour une nouvelle entrée
y = *is;                // lecture d'un entier
++is;                   // etc.
```

Acquision en un coup de toute une séquence :

```
vector<int> v;
...
copy(istream_iterator<int>(cin),
                        istream_iterator<int>(), back_inserter(v));
```

D'autres types d'itérateurs sont définis ou suggérés, nous ne les détaillerons pas ici. Par exemple

- un itérateur inverse (**`reverse_iterator`**) se construit au-dessus d'un itérateur *i* existant et représente le parcours de la séquence dans le sens contraire de celui que représente *i*,
- un itérateur contrôlé (**`checked_iterator`**), laissé à implémenter au programmeur, prend soin de détecter et signaler les accès à des positions invalides et les débordements de la structure de données sous-jacente.

Le document de la norme ISO spécifie tous ces éléments.

9.4 Algorithmes

Fichier en-tête :

```
<algorithm>
```

Certaines opérations à faire sur les éléments d'un conteneur sont trop dépendantes des particularités de ce dernier pour qu'on les implémente autrement que comme fonctions membres du conteneur. D'autres opérations, au contraire, se prêtent à être programmées sous une forme indépendante du conteneur concerné, lequel n'est manipulé qu'à travers un ou plusieurs itérateurs.

Les algorithmes de la bibliothèque standard sont des implantations d'opérations de cette deuxième sorte. Leurs arguments et résultats sont décrits en termes d'itérateurs (cf. § 9.3), qui permettent d'accéder aux éléments des conteneurs, et de foncteurs (cf. § 9.1.2), pour exprimer les opérations à faire sur ces éléments.

Remarques communes

1. Beaucoup d'algorithmes de la bibliothèque standard sont si simples qu'ils sont implémentés par des fonctions en ligne, ce qui contribue grandement à leur efficacité.

2. La plupart des algorithmes de la bibliothèque standard s'appliquent à une ou plusieurs séquences, chacune représentée par deux itérateurs *debut* et *fin*. Il s'agit toujours de séquences « fermées à gauche et ouvertes à droite », que nous représentons avec la notation [*debut*, *fin* [, composées donc des valeurs pointées par

$$debut,\ debut + 1,\ \ldots\ fin - 1$$

Autrement dit, la valeur *fin* (souvent notée *last* dans la norme ISO) ne représente pas le dernier élément, mais un élément, éventuellement fictif, qui serait immédiatement après le dernier élément valide de la séquence.

3. Les algorithmes qui renvoient un itérateur utilisent la valeur de fin de séquence (qu'ils reçoivent comme argument) pour signaler l'arrivée d'un incident l'échec de l'opération demandée. Par exemple, la recherche d'une valeur **`x`** dans une liste **`L`** s'écrit :

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

4. Les algorithmes ne font *aucun contrôle* de la validité des opérations qu'ils effectuent, et notamment de leurs accès à la mémoire, que ce soit en lecture ou en écriture. Par exemple, l'algorithme standard **`copy`**, qu'on appelle sous la forme

```
copy(debut, fin, result);
```

(**`debut`**, **`fin`** et **`result`** sont des itérateurs) copie les éléments de la séquence [**`debut`**, **`fin`**[sur les emplacements pointés par **`result`**, **`result`** + 1, ... **`result`** + **`fin`** – **`debut`** – 1, sans se soucier de savoir si ces emplacements correspondent à de la mémoire disponible et accessible en écriture.

L'algorithme ne vérifie même pas que **debut** et **fin** délimitent bien une séquence, c'est-à-dire que la boucle

```
for (iterateur i = debut; i != fin; i++) ...
```

a une fin.

Conventions

Dans la description qui suit les conventions suivantes sont adoptées pour représenter les types arguments des modèles :

- **Iter**, un type d'itérateur quelconque
- **IterIn**, un type d'itérateur d'entrée
- **IterOut** un type d'itérateur de sortie
- **IterAv** un type d'itérateur en avant
- **IterBid** un type d'itérateur bidirectionnel
- **IterAlea** un type d'itérateur d'accès aléatoire
- **Fonc** un type de foncteur quelconque (fonction ou objet fonction)
- **PredUn** un type de prédicat unaire
- **PredBin** un type de prédicat binaire
- **OperUn** un type d'opération unaire
- **OperBin** un type d'opération binaire
- **Compare** un type de relation binaire induisant un *ordre strict faible* (cf. § 9.2.5)

9.4.1 Algorithmes de consultation de séquences

Les algorithmes suivants ne modifient pas les éléments des conteneurs auxquels ils s'appliquent :

♦ **for_each**

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

Action : appel de **f** sur chaque élément de la séquence [**debut, fin** [.

Retour : **f**

♦ **find**

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

Action : recherche le premier élément de la séquence [**debut, fin** [égal à la **valeur** indiquée.

Retour : un itérateur positionné sur l'élément en question, s'il existe ; la valeur **fin** sinon.

Contrainte : la relation égalité doit être définie sur le type **T**.

♦ **find_if**

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

Action : recherche le premier élément *x* de la séquence [**debut, fin** [tel que **pred**(*x*) ≠ **false**.

Retour : un itérateur positionné sur l'élément en question, s'il existe ; la valeur **fin** sinon.

Exemple :

```
list<int> l;
l.push_back(-3);
l.push_back( 0);
l.push_back( 3);
l.push_back(-2);
l.push_back( 5);
list<int>::iterator result =
    find_if(l.begin(), l.end(), bind2nd(greater<int>(), 0));
if (result != l.end())
    cout << *result;
```

- **find_end**

```
template<class IterAv1, class IterAv2>
    IterAv1 find_end(IterAv1 debut1, IterAv1 fin1,
                     IterAv2 debut2, IterAv2 fin2);
```

Action : recherche de la dernière occurrence de la séquence [**debut2, fin2** [comme sous-séquence de la séquence [**debut1, fin1** [.

Plus précisément, on recherche une valeur d'itérateur *r* telle que

pour tout $0 \le i <$ **fin2** – **deb2** on ait $r + i <$ **fin1** et ***(***r* **+** *i***)** = ***(debut2 +** *i***)**

Retour : la valeur r en question, si elle existe ; la valeur **fin1** sinon.

```
template<class IterAv1, class IterAv2, class PredBin>
    IterAv1 find_end(IterAv1 debut1, IterAv1 fin1,
                     IterAv2 debut2, IterAv2 fin2, PredBin pred);
```

Action : recherche une valeur d'itérateur *r* telle que

pour tout $0 \le i <$ **fin2** – **deb2** on ait $r + i <$ **fin1** et **pred(*(***r* **+** *i***), *(debut2 +** *i***))** $\neq$ **false**

Retour : la valeur r en question, si elle existe ; la valeur **fin1** sinon.

Exemple :

```
char *mot = "experimentalement";
char *suffixe = "ment";
int lmot = strlen(mot), lsuf = strlen(suffixe);
char *pos = find_end(mot, mot + lmot, suffixe, suffixe + lsuf);
cout << "le mot '" << mot << "' se termine par '" << suffixe << "' : "
    << (pos + lsuf == mot + lmot ? "oui" : "non") << endl;
```

Affichage obtenu :

```
le mot 'experimentalement' se termine par 'ment' : oui
```

- **find_first_of**

```
template<class IterAv1, class IterAv2>
    IterAv1 find_first_of(IterAv1 debut1, IterAv1 fin1,
                          IterAv2 debut2, IterAv2 fin2);
```

Action : recherche le premier élément de la séquence [**debut1, fin1** [qui apparaît dans la séquence [**debut2, fin2** [.

Retour : la valeur d'un itérateur positionné sur un tel élement, s'il existe ; la valeur **fin1** sinon.

```
template<class IterAv1, class IterAv2, class PredBin>
    IterAv1 find_first_of(IterAv1 debut1, IterAv1 fin1,
                          IterAv2 debut2, IterAv2 fin2, PredBin pred);
```

Action : recherche la plus petite valeur d'itérateur *r* telle que

debut1 $\le r <$ **fin1** et il existe *s* vérifiant **debut2** $\le s <$ **fin2** et **pred(***r**, ***s**)** $\neq$ **false**

Retour : la valeur *r* si elle existe ; la valeur **fin1** sinon.

♦ **adjacent_find**

```
template<class IterAv>
    IterAv adjacent_find(IterAv debut, IterAv fin);
```

Action : recherche du premier couple d'éléments adjacents égaux de la séquence [**debut, fin** [

Retour : la valeur d'un itérateur positionné sur le premier élément d'un tel couple, s'il existe ; **fin** sinon.

```
template<class IterAv, class PredBin>
    IterAv adjacent_find(IterAv debut, IterAv fin, PredBin pred);
```

Action : recherche de la plus petite valeur d'itérateur *r* telle que

debut ≤ *r*, *r* + 1 < **fin** et **pred(***r*, ***(***r* **+ 1))** ≠ **false**

Retour : la valeur *r* si elle existe ; la valeur **fin** sinon.

♦ **count**

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

Action : comptage du nombre d'éléments de la séquence [**debut, fin** [égaux à la **valeur** indiquée.

Retour : le nombre de tels éléments.

Contrainte : la relation égalité doit être définie sur le type **T**.

♦ **count_if**

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

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

Retour : le nombre de tels éléments.

♦ **mismatch**

```
template<class IterIn1, class IterIn2>
    pair<IterIn1, IterIn2> mismatch(IterIn1 debut1, IterIn1 fin1,
                                    IterIn2 debut2);
```

Action : recherche de la première discordance entre éléments correspondants des séquences [**debut1, fin1** [et [**debut2, fin2** [.

Retour : la paire **(***i***,** *j***)** , avec *j* = **debut2** + *i* – **debut1**, si un tel couple existe, autrement la paire **(fin1, debut2 + (fin1 - debut1))**.

```
template<class IterIn1, class IterIn2, class PredBin>
    pair<IterIn1, IterIn2> mismatch(IterIn1 debut1, IterIn1 fin1,
                                    IterIn2 debut2, PredBin pred);
```

Action : on recherche un couple d'itérateurs (*i*, *j*) tels que

i est la plus petite valeur telle que **debut1** ≤ *i* < **fin1** et
posant *j* = **debut2 +** *i* – **debut1**, on a *j* < **fin2** et **pred(***i***,** ***j***)** = **false**

Retour : la paire **(***i***,** *j***)** si un tel couple existe, autrement la paire **(fin1, debut2 + (fin1 - debut1))**.

♦ **equal**

```
template<class IterIn1, class IterIn2>
    bool equal(IterIn1 debut1, IterIn1 fin1, IterIn2 debut2);
```

Retour : la réponse à la question : les séquences [**debut1, fin1** [et [**debut2, fin2** [sont-elles égales ?

```
template<class IterIn1, class IterIn2, class PredBin>
    bool equal(IterIn1 debut1, IterIn1 fin1, IterIn2 debut2, PredBin pred);
```

Action : évaluation de la condition

pour chaque *i* vérifiant **debut1** ≤ *i* < **fin1** on a
debut2 + *i* – **debut1** < **fin2** et **pred(***i*, ***(debut2** + *i* – **debut1))** ≠ **false**

Retour : le résultat de cette évaluation

♦ **search**

```
template<class IterAv1, class IterAv2>
    IterAv1 search(IterAv1 debut1, IterAv1 fin1,
                   IterAv2 debut2, IterAv2 fin2);
```

Retour : un itérateur positionné sur le début de la première occurrence de la sous-séquence [**debut2, fin2** [dans la séquence [**debut1, fin1** [, si une telle occurrence existe ; la valeur **fin1** sinon.

```
template<class IterAv1, class IterAv2, class PredBin>
    IterAv1 search(IterAv1 debut1, IterAv1 fin1,
                   IterAv2 debut2, IterAv2 fin2, PredBin pred);
```

Action : recherche de la plus petite valeur d'itérateur *r* telle que

debut1 ≤ *r*, *r* + **fin2** – **debut2** < **fin1** et
pour tout 0 ≤ *i* < **fin2** – **debut2** on a **pred(*(***r** + *i***), *(debut2** + *i***))** ≠ **false**.

Retour : la valeur *r* si elle existe ; la valeur **fin1** sinon.

♦ **search_n**

```
template<class IterAv, class Taille, class T>
    IterAv search_n(IterAv debut, IterAv fin,
                            Taille nombre, const T& valeur);
```

Retour : un itérateur positionné sur le début de la première occurrence dans la séquence [**debut, fin** [d'une sous-séquence constituée de **nombre** fois la **valeur** indiquée.

Contrainte : l'égalité doit être définie sur le type **T** ; le type **Taille** doit pouvoir être converti en entier.

```
template<class IterAv, class Taille, class T, class PredBin >
    IterAv search_n(IterAv debut, IterAv fin,
                            Taille nombre, const T& valeur, PredBin pred);
```

Action : recherche de la plus petite valeur d'itérateur *r* telle que

debut ≤ *r* < **fin** – **nombre** et pour tout 0 ≤ *i* < **nombre** on a **pred(*(***r** + *i***), valeur)** ≠ **false**.

Retour : la valeur *r* si elle existe ; la valeur **fin** sinon.

Contrainte : l'égalité doit être définie sur le type **T** ; le type **Taille** doit pouvoir être converti en entier.

9.4.2 Algorithmes qui modifient les séquences

♦ **copy**

```
template<class IterIn, class IterOut>
    IterOut copy(IterIn debut, IterIn fin, IterOut result);
```

Action : copie, *en avançant*, des éléments de la séquence [**debut, fin** [sur les éléments de la séquence [**result, result + fin - debut** [. Equivaut à :

pour *i* allant de 0 à **fin – debut - 1**, faire ***(result** + *i***) = *(debut** + *i***)**

Retour : la valeur **result + fin – debut**.

Contrainte : **result** ne doit pas appartenir à l'intervalle [**debut, fin** [.

Exemple :

```
int t[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int n = sizeof t / sizeof t[0];
list<int> l(n);
copy(t, t + n, l.begin());
...
```

- **copy_backward**

```
template<class IterBid1, class IterBid2>
    IterBid2 copy_backward (IterBid1 debut, IterBid1 fin, IterBid2result);
```

Action : copie, *en reculant*, des éléments de la séquence [**debut, fin** [sur les éléments de la séquence [**result – (fin - debut), result** [. Equivaut à :

pour *i* allant de 0 à **fin – debut - 1**, faire ***(result - *i*) = *(fin - *i*)**

Retour : la valeur **result - (fin – debut)**.

Contrainte : **result** ne doit pas appartenir à l'intervalle [**debut, fin** [.

- **swap**

```
template<class T> void swap(T& a, T& b);
```

Action : échange les éléments **a** et **b**.

Contrainte : le type **T** doit supporter l'affectation.

- **iter_swap**

```
template<class IterAv1, class IterAv2>
    void iter_swap(IterAv1 a, IterAv2 b);
```

Action : échange les éléments ***a** et ***b**.

- **swap_ranges**

```
template<class IterAv1, class IterAv2>
    IterAv2 swap_ranges(IterAv1 debut1, IterAv1 fin1, IterAv2 debut2);
```

Action : échange les éléments de la séquence [**debut1, fin1** [avec les éléments correspondants de la séquence [**debut2, debut2 + fin1 – debut1** [.Equivaut à :

pour *i* allant de 0 à **fin1 – debut1 - 1**, faire **swap(*(debut1 + *i*), *(debut2 + *i*))**

Retour : la valeur **debut2 - (fin1 – debut1)**.

Contrainte : les intervalles [**debut1, fin1** [et [**debut2, debut2 + fin1 – debut1** [sont disjoints.

- **transform**

```
template<class IterIn, class IterOut, class OperUn>
    IterOut transform(IterIn debut, IterIn fin, IterOut result, OperUn op);
```

Action : affectation des éléments de la séquence [**result, result + fin - debut** [par application de l'opération **op** aux éléments correspondants de la séquence [**debut, fin** [. Equivaut à

pour *i* allant de 0 à **fin – debut - 1**, faire ***(result + *i*) = op(*(debut + *i*))**

Retour : la valeur **result + fin – debut**.

Contrainte : **op** ne doit pas avoir d'effet de bord.

```
template<class IterIn1, class IterIn2, class IterOut, class OperBin>
    IterOut transform(IterIn1 debut1, IterIn1 fin1,
                      IterIn2 debut2, IterOut result, OperBin opbin);
```

Action : affectation des éléments de la séquence [**result, result + fin1 – debut1** [par application de **opbin** aux éléments des séquences [**debut1, fin1** [et [**debut2, debut2 + fin1 - debut1** [. Equivaut à

pour *i* allant de 0 à **fin – debut - 1**, faire ***(result + *i*) = opbin(*(debut1 + *i*), *(debut2 + *i*))**

Retour : la valeur **`result + fin – debut`**.

Contrainte : **`opbin`** ne doit pas avoir d'effet de bord.

Exemple : somme de deux vecteurs :

```
vector<double> a(N);
vector<double> b(N);
vector<double> c(N);

...

transform(a.begin(), a.end(), b.begin(), c.begin(), plus<double>());
```

- **replace**

```
template<class IterAv, class T>
    void replace(IterAv debut, IterAv fin,
                            const T& oldVal, const T& newVal);
```

Action : toutes les occurrences de **`oldVal`** dans la séquence [**`debut, fin`** [sont remplacées par **`newVal`**.

Contrainte : le type **`T`** doit supporter l'égalité et l'affectation.

Exemple (l'affichage obtenu est : **`Aix_en_Provence`**) :

```
char s[] = "Aix-en-Provence";
replace(s, s + strlen(s), '-', '_');
cout << s << "\n";
```

- **replace_if**

```
template<class IterAv, class PredUn, class T>
    void replace_if(IterAv debut, IterAv fin,
                            PredUn pred, const T& newVal);
```

Action : tout *x* de la séquence [**`debut, fin`** [qui vérifie **`pred(`***x***`)`** ≠ **`false`** est remplacé par **`newVal`**.

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

Exemple :

```
struct EstNegatif
    { bool operator()(int i) { return i < 0; } };

list<int> l;
l.push_back( 0);    l.push_back(-1);
l.push_back( 2);    l.push_back(-3);
l.push_back( 4);    l.push_back(-5);

replace_if(l.begin(), l.end(), EstNegatif(), 0);
...
```

- **replace_copy**

```
template<class IterIn, class IterOut, class T>
    IterOut replace_copy(IterIn debut, IterIn fin, IterOut result,
                                    const T& oldVal, const T& newVal);
```

Action : copie la séquence [**`debut, fin`** [sur la séquence [**`result, result + fin - debut`** [en remplaçant par **`newVal`** les éléments égaux à **`oldVal`**.

Retour : la valeur **`result + fin – debut`**.

Contrainte : le type **`T`** doit supporter l'égalité et l'affectation.

- **replace_copy_if**

```
template<class Iter, class IterOut, class PredUn, class T>
    IterOut replace_copy_if(Iter debut, Iter fin, IterOut result,
                                        PredUn pred, const T& newVal);
```

Action : copie la séquence [**`debut, fin`** [sur la séquence [**`result, result + fin - debut`** [en remplaçant par **`newVal`** les éléments *x* qui vérifient **`pred(`***x***`)`** ≠ **`false`**.

Retour : la valeur **result + fin – debut**.

Contrainte : le type **T** doit supporter l'affectation.

- **fill**

```
template<class IterAv, class T>
    void fill(IterAv debut, IterAv fin, const T& valeur);
```

Action : tous les éléments de la séquence [**debut, fin** [sont affectés de la **valeur** indiquée.

Contrainte : le type **T** doit supporter l'affectation.

- **fill_n**

```
template<class IterOut, class Taille, class T>
    void fill_n(IterOut debut, Taille nombre, const T& valeur);
```

Action : tous les éléments de la séquence [**debut, debut + nombre** [sont affectés de la **valeur** indiquée.

Contrainte : le type **T** doit supporter l'affectation et le type **Taille** doit pouvoir être converti en entier.

- **generate**

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

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

Contrainte : **Generateur** est un type objet fonction (cf. § 9.1.2) sans argument.

Exemple : création de la liste des **N** premier entiers naturels :

```
struct Compteur
    {
    Compteur()          { cpt = 0; }
    int operator()()    { return cpt++; }
private:
    int cpt;
    };

list<int> l(N);
generate(l.begin(), l.end(), Compteur());
...
```

- **generate_n**

```
template<class IterOut, class Taille, class Generateur>
    void (IterOut debut, Taille nombre, Generateur gen);
```

Action : tous les éléments de la séquence [**debut, debut + nombre** [sont affectés de la valeur **gen()**.

Contrainte : **Generateur** est un type objet fonction (cf. § 9.1.2) sans argument et le type **Taille** peut être converti en entier.

- **remove**

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

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

Retour : la fin de la séquence résultante.

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

- **remove_if**

```
template<class IterAv, class PredUn>
    IterAv remove_if(IterAv debut, IterAv fin, PredUn pred);
```

Action : suppression de tout *x* de la séquence [**debut, fin** [vérifiant **pred(***x***)** ≠ **false**.

Retour : la fin de la séquence résultante.

- **remove_copy**

```
template<class IterIn, class IterOut, class T>
    IterOut remove_copy(IterIn debut, IterIn fin,
                                    IterOut result, const T& valeur);
```

Action : copie dans la séquence commençant en **result** les éléments de la séquence [**debut, fin** [qui ne sont pas égaux à la **valeur** indiquée.

Retour : la fin de la séquence résultante.

Contrainte : **T** supporte l'égalité ; les intervalles [**debut, fin** [et [**result, result + debut - fin** [ne se chevauchent pas.

- **remove_copy_if**

```
template<class IterIn, class IterOut, class Predicate>
    IterOut remove_copy_if(IterIn debut, IterIn fin,
                                    IterOut result, Predicate pred);
```

Action : copie dans la séquence commençant en **result** les éléments *x* de la séquence [**debut, fin** [pour lesquels **pred(***x***)** = **false**.

Retour : la fin de la séquence résultante.

Contrainte : Les intervalles [**debut, fin** [et [**result, result + debut - fin** [ne se chevauchent pas.

- **unique**

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

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

Retour : la fin de la séquence résultante.

```
template<class IterAv, class PredBin>
    IterAv unique(IterAv debut, IterAv fin, PredBin pred);
```

Action : remplacement par son premier élément de chaque sous-séquence [*a*, *a* + *n*[vérifiant : pour tout 0 < *i* < *n*, **pred(*(***a** **+** *i* **- 1), *(***a** + *i***))** ≠ **false**.

Retour : la fin de la séquence produite.

Exemple :

```
char s[] = "AAABBBCCCAAABBBCCCAAABBBCCC";

cout << "s : '" << s << "'\n";
char *r = unique(s, s + strlen(s));
cout << "s : '" << s << "'\n";
*r = '\0';
cout << "s : '" << s << "'\n";
```

Affichage obtenu :

```
s : 'AAABBBCCCAAABBBCCCAAABBBCCC'
s : 'ABCABCABCAAABBBCCCAAABBBCCC'
s : 'ABCABCABC'
```

- **unique_copy**

```
template<class IterIn, class IterOut>
    IterOut unique_copy(IterIn debut, IterIn fin, IterOut result);
```

Action : copie les éléments de la séquence [**debut, fin** [dans la séquence commençant à la position **result** en remplaçant par un unique élément chaque sous-séquence d'éléments identiques.

Retour : la fin de la séquence résultante.

Contrainte : Les intervalles [**debut, fin** [et [**result, result + debut - fin** [ne se chevauchent pas.

```
template<class IterIn, class IterOut, class PredBin>
    IterOut unique_copy(IterIn debut, IterIn fin, IterOut result,
                                                  PredBin pred);
```

Action : copie les éléments de la séquence [**debut, fin** [dans la séquence commençant à la position **result** en remplaçant par son premier élément chaque sous-séquence [*a*, *a* + *n*[vérifiant : pour tout 0 < *i* < *n*, **pred(*(***a** + *i* **- 1), *(***a** + *i***))** ≠ **false**.

Retour : la fin de la séquence résultante.

Contrainte : Les intervalles [**debut, fin** [et [**result, result + debut - fin** [ne se chevauchent pas.

♦ **reverse**

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

Action : renversement de la séquence indiquée. Equivaut à :

pour *i* allant de 0 à (**fin** – **debut**) / 2 **–** 1 faire **swap(*(debut +** *i***), *(fin –** *i* **– 1))**

♦ **reverse_copy**

```
template<class IterBid, class IterOut>
    IterOut reverse_copy(IterBid debut, IterBid fin, IterOut result);
```

Action : copie de la séquence [**debut, fin** [sur la séquence [**result, result + fin - debut** [en inversant l'ordre des éléments. Equivaut à :

pour i allant de 0 à **fin** – **debut** – **1** faire ***(result + fin – debut – 1 -** *i***) = *(debut + i)**.

Retour : la valeur **result + fin - debut**.

Contrainte : Les intervalles [**debut, fin** [et [**result, result + debut - fin** [ne se chevauchent pas.

♦ **rotate**

```
template<class IterAv>
    void rotate(IterAv debut, IterAv milieu, IterAv fin);
```

Action : « rotation » des éléments de la séquence [**debut, fin** [« autour » de l'élément pointé par **milieu** : chaque élément de la séquence [**debut, fin** [est poussé vers la droite de **fin** – **milieu** positions (en *imaginant* que la séquence est circulaire, c'est-à-dire que **debut** = **fin**).

Autrement dit, pour *i* allant de 0 à **fin** – **debut** – 1, l'élément qui était à la position **debut** + *i* se retrouve à la position **debut + (***i* **+ fin – milieu) % (fin – debut)**.

Contrainte : [**debut, milieu** [et [**milieu, fin** [sont des intervalles valides.

♦ **rotate_copy**

```
template<class IterAv, class IterOut>
    IterOut rotate_copy(IterAv debut, IterAv milieu, IterAv fin,
                                                  IterOut result);
```

Action : copie la séquence [**debut, fin** [sur la séquence [**result, result + fin - debut** [en « la faisant tourner autour de **milieu** ». Ce qui signifie : pour *i* allant de 0 à **fin** – **debut** – 1, faire

(result +** *i) = *(debut + (***i* **+ fin – milieu) % (fin – debut))**

Retour : la valeur **result + fin – debut**.

Contrainte : Les intervalles [**debut, fin** [et [**result, result + debut - fin** [ne se chevauchent pas.

♦ **random_shuffle**

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

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

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

Action : arrangement aléatoire des éléments de la séquence [**debut, fin** [en utilisant le générateur de nombres aléatoires **rand**.

Exemple :

```
int t[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
random_shuffle(t, t + sizeof t / sizeof t[0];);
...
```

t est le tableau { 7, 9, 10, 3, 2, 5, 4, 8, 1, 6 } ou un autre des 3 628 800 (10 !) arrangements possibles.

- **partition**

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

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

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

Exemple :

```
int t[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int n = sizeof t / sizeof t[0];
vector<int> v(n);
vector<int>::iterator r;

copy(t, t + n, v.begin());
cout << "v : " << v << "\n";

random_shuffle(v.begin(), v.end());
cout << "v : " << v << "\n";

r = partition(v.begin(), v.end(), bind2nd(less<int>(), 6));
cout << "v : " << v << "\n";

cout << "*r : " << *r << "\n";
```

Affichage obtenu :

```
v : [ 1 2 3 4 5 6 7 8 9 10 ]
v : [ 7 9 10 3 2 5 4 8 1 6 ]
v : [ 1 4 5 3 2 10 9 8 7 6 ]
*r : 10
```

- **stable_partition**

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

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

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

De plus, deux éléments de la séquence [**debut,** *r* [ou de la séquence [*r*, **fin** [sont placés l'un par rapport à l'autre comme ils l'étaient dans [**debut, fin** [.

9.4.3 Algorithmes des séquences ordonnées

Les algorithmes de cette section portent sur des collections d'objets sur lesquels un ordre est défini. Chaque algorithme existe sous deux versions (voyez par exemple **sort**, ci-dessous) :

- dans la première, la relation définissant l'ordre ne figure pas parmi les types arguments du modèle ; il est supposé dans ce cas que l'opérateur binaire **operator<** est accessible et peut être appliqué à des couples d'objets ayant le type des objets que les itérateurs de l'algorithme atteignent ;
- dans la deuxième version, un type argument, noté **Compare**, représente explicitement l'ordre par rapport auquel l'algorithme travaille ; c'est un type de relation binaire, censée définir un *ordre strict faible* (cf. § 9.2.5, « A propos de **Compare** »).

Tris

Les opérations **sort**, **stable_sort**, **partial_sort**, **partial_sort_copy** sont des algorithmes de tri.

♦ **sort**

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

template<class IterAlea, class Compare>
    void sort(IterAlea debut, IterAlea fin, Compare comp);
```

Action : tri des éléments de la séquence [**debut, fin** [

♦ **stable_sort**

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

template<class IterAlea, class Compare>
    void stable_sort(IterAlea first, IterAlea fin, Compare comp);
```

Action : tri des éléments de la séquence [**debut, fin** [. Après le tri, les éléments équivalents sont placés, les uns par rapport aux autres, comme ils l'étaient avant le tri.

♦ **partial_sort**

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

template<class IterAlea, class Compare>
    void partial_sort(IterAlea debut, IterAlea milieu, IterAlea fin,
                                                        Compare comp);
```

Action : les **(milieu** – **debut)** plus petits éléments de la séquence [**debut, fin** [sont placés, triés, dans la séquence [**debut, milieu** [; les autres éléments de [**debut, fin** [sont placés, dans un ordre indéterminé, dans [**milieu, fin** [.

Exemple :

```
int A[] = { 7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
const int N = sizeof(A) / sizeof(int);

partial_sort(A, A + 5, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
```

Affichage obtenu :

```
1 2 3 4 5 11 12 10 9 8 7 6
```

♦ **partial_sort_copy**

```
template<class IterIn, class IterAlea >
    IterAlea partial_sort_copy(IterIn debut, IterIn fin,
                               IterAlea debut_result, IterAlea fin_result);

template<class IterIn, class IterAlea, class Compare>
    IterAlea partial_sort_copy(IterIn debut, IterIn fin,
                    IterAlea debut_result, IterAlea fin_result, Compare comp);
```

Action : les *m* = *min*(**fin** – **debut**, **fin_result** – **debut_result**) plus petits éléments de la séquence [**debut, fin** [sont placés, triés, dans la séquence [**debut_result, debut_result** + *m* [.

♦ **nth_element**

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

template<class IterAlea, class Compare>
    void nth_element(IterAlea debut, IterAlea nth, IterAlea fin,
                                                          Compare comp);
```

Action : après l'exécution de cet algorithme, l'élément à la position **nth** est celui qui se trouverait à cet endroit si la séquence [**debut, fin** [était triée. De plus, aucun élément de [**nth, fin** [n'est inférieur à aucun élément de [**debut, nth** [.

Recherche binaire (dichotomique)

Les opérations **lower_bound**, **upper_bound**, **equal_range** et **binary_search** concernent l'insertion et la recherche d'éléments dans les séquences ordonnées.

♦ **lower_bound**

```
template<class IterAv, class T>
    IterAv lower_bound(IterAv debut, IterAv fin, const T& valeur);

template<class IterAv, class T, class Compare>
    IterAv lower_bound IterAv debut, IterAv fin, const T& valeur,
                                                          Compare comp);
```

Retour : un itérateur pointant la position la plus à gauche, dans la séquence triée [**debut, fin** [, à laquelle on peut placer la **valeur** indiquée sans violer l'ordre.

Contrainte : la séquence [**debut, fin** [doit être triée.

♦ **upper_bound**

```
template<class IterAv, class T>
    IterAv upper_bound(IterAv debut, IterAv fin, const T& valeur);

template<class IterAv, class T, class Compare>
    IterAv upper_bound(IterAv debut, IterAv fin, const T& valeur,
                                                          Compare comp);
```

Retour : un itérateur pointant la position la plus à droite, dans la séquence triée [**debut, fin** [, à laquelle on peut placer la **valeur** indiquée sans violer l'ordre.

Contrainte : la séquence [**debut, fin** [doit être triée.

♦ **equal_range**

```
template<class IterAv, class T>
    pair<IterAv, IterAv> equal_range(IterAv debut, IterAv fin,
                                                          const T& valeur);

template<class IterAv, class T, class Compare>
    pair<IterAv, IterAv> equal_range(IterAv debut, IterAv fin,
                                          const T& valeur, Compare comp);
```

Retour : un couple d'itérateurs (*i*, *j*) représentant le plus grand intervalle [*i*, *j* [inclus dans la séquence triée [**debut, fin** [tel que l'on puisse placer la **valeur** indiquée à n'importe quelle position dans [*i*, *j* [sans violer l'ordre.

Contrainte : la séquence [**debut, fin** [doit être triée.

Exemple : création incrémentale d'un vecteur trié :

```
const int nMax = 100;
int n = 0, x;

vector<int> v;
vector<int>::iterator r, s;
v.reserve(nMax);

for (;;)
    {
    cout << "\n? "; cin >> x; if (x < 0) break;

    r = equal_range(v.begin(), v.end(), x).first;
    s = v.end();
    v.resize(v.size() + 1);
    copy_backward (r, s, v.end());
    *r = x;
    }
```

- **binary_search**

```
template<class IterAv, class T>
    bool binary_search(IterAv debut, IterAv fin, const T& valeur);

template<class IterAv, class T, class Compare>
    bool binary_search(IterAv debut, IterAv fin, const T& valeur,
                                                        Compare comp);
```

Retour : la réponse à la question « existe-t-il dans la séquence [**debut, fin** [un élément dont la valeur est équivalente à la **valeur** indiquée ? »

NOTE. Si on recherche la place de l'élément il vaut mieux utiliser **equal_range**.

Contrainte : la séquence [**debut, fin** [doit être triée.

- **merge**

```
template<class IterIn1, class IterIn2, class IterOut>
    IterOut merge(IterIn1 debut1, IterIn1 fin1,
                        IterIn2 debut2, IterIn2 fin2, IterOut result);

template<class IterIn1, class IterIn2, class IterOut, class Compare>
    IterOut merge(IterIn1 debut1, IterIn1 fin1,
            IterIn2 debut2, IterIn2 fin2, IterOut result, Compare comp);
```

Action : fusion des deux séquences triées [**debut1, fin1** [et [**debut2, fin2** [en une unique séquence triée [**result, result + (fin1 – debut1) + (fin2 – debut2)** [.

Retour : la valeur **result + (fin1 – debut1) + (fin2 – debut2)**

Contrainte : les deux séquences [**debut1, fin1** [et [**debut2, fin2** [doivent être triées.

- **inplace_merge**

```
template<class IterBin>
    void inplace_merge(IterBin debut, IterBin milieu, IterBin fin);

template<class IterBin, class Compare>
    void inplace_merge(IterBin debut, IterBin milieu, IterBin fin,
                                                        Compare comp);
```

Action : fusion des deux sous-séquences triées [**debut, milieu** [et [**milieu, fin** [en une unique séquence triée placée en [**debut, fin** [.

Contrainte : les deux sous-séquences [**debut, milieu** [et [**milieu, fin** [doivent être triées.

Opérations ensemblistes

Les algorithmes **includes**, **set_union**, **set_intersection**, **set_difference**, **set_symmetric_difference** implémentent des opérations sur des ensembles représentés par des séquences triées.

- **includes**

```
template<class IterIn1, class IterIn2>
    bool includes(IterIn1 debut1, IterIn1 fin1,
                            IterIn2 debut2, IterIn2 fin2);

template<class IterIn1, class IterIn2, class Compare>
    bool includes(IterIn1 debut1, IterIn1 fin1,
                            IterIn2 debut2, IterIn2 fin2, Compare comp);
```

Retour : la réponse à la question « les éléments de la séquence [**debut1, fin1** [apparaissent-ils tous dans la séquence [**debut2, fin2** [? »

Contrainte : les deux séquences [**debut1, fin1** [et [**debut2, fin2** [doivent être triées.

- **set_union**

```
template<class IterIn1, class IterIn2, class IterOut>
    IterOut set_union(IterIn1 debut1, IterIn1 fin1,
                    IterIn2 debut2, IterIn2 fin2, IterOut result);

template<class IterIn1, class IterIn2, class IterOut, class Compare>
    IterOut set_union(IterIn1 debut1, IterIn1 fin1,
                   IterIn2 debut2, IterIn2 fin2, IterOut result, Compare comp);
```

Action : construit la réunion des séquences triées [**debut1, fin1** [et [**debut2, fin2** [, sous la forme d'une séquence triée rangée à partir de la position **result**.

Retour : la fin de la séquence résultante.

Contrainte : les séquences [**debut1, fin1** [et [**debut2, fin2** [doivent être triées.

- **set_intersection**

```
template<class IterIn1, class IterIn2, class IterOut>
    IterOut set_intersection(IterIn1 debut1, IterIn1 fin1,
                    IterIn2 debut2, IterIn2 fin2, OutputIterator result);

template<class IterIn1, class IterIn2, class IterOut, class Compare>
    IterOut set_intersection(IterIn1 debut1, IterIn1 fin1,
                   IterIn2 debut2, IterIn2 fin2, IterOut result, Compare comp);
```

Action : construit l'intersection des séquences triées [**debut1, fin1** [et [**debut2, fin2** [, sous la forme d'une séquence triée rangée à partir de la position **result**.

Retour : la fin de la séquence résultante.

Contrainte : les séquences [**debut1, fin1** [et [**debut2, fin2** [doivent être triées.

Exemple :

```
int dA[] = { 2, 4, 6,  8, 10, 12, 14, 16, 18, 20 };
int dB[] = { 3, 6, 9, 12, 15, 18, 21, 24, 27, 30 };
int dC[50];
int *fA = dA + 10, *fB = dB + 10;

int *fC = set_intersection(dA, fA, dB, fB, dC);

for (int *p = dC; p != fC; p++)
    cout << *p << ' ';
```

L'affichage obtenu est : **6 12 18**

♦ **set_difference**

```
template<class IterIn1, class IterIn2, class IterOut>
    IterOut set_difference(IterIn1 debut1, IterIn1 fin1,
                           IterIn2 debut2, IterIn2 fin2, IterOut result);

template<class IterIn1, class IterIn2, class IterOut, class Compare>
    IterOut set_difference(IterIn1 debut1, IterIn1 fin1,
            IterIn2 debut2, IterIn2 fin2, IterOut result, Compare comp);
```

Action : construit la différence des séquences triées [**debut1, fin1** [et [**debut2, fin2** [(c'est-à-dire l'ensemble des éléments qui sont dans [**debut1, fin1** [sans être dans [**debut2, fin2** [) sous la forme d'une séquence triée rangée à partir de la position **result**.

Retour : la fin de la séquence résultante.

Contrainte : les séquences [**debut1, fin1** [et [**debut2, fin2** [doivent être triées.

♦ **set_symmetric_difference**

```
template<class IterIn1, class IterIn2, class IterOut>
    IterOut set_symmetric_difference(IterIn1 debut1, IterIn1 fin1,
                           IterIn2 debut2, IterIn2 fin2, IterOut result);

template<class IterIn1, class IterIn2, class IterOut, class Compare>
    OutputIterator set_symmetric_difference(IterIn1 debut1, IterIn1 fin1,
            IterIn2 debut2, IterIn2 fin2, IterOut result, Compare comp);
```

Action : construit la différence symétrique des séquences triées [**debut1, fin1** [et [**debut2, fin2** [(c'est-à-dire l'ensemble des éléments qui appartiennent à l'une ou l'autre de ces deux séquences sans appartenir aux deux) sous la forme d'une séquence triée rangée à partir de la position **result**.

Retour : la fin de la séquence résultante.

Contrainte : les séquences [**debut1, fin1** [et [**debut2, fin2** [doivent être triées.

Opérations sur les tas

Les opérations suivantes concernent des séquences organisées comme des tas. Un *tas*, ou *maximier*, est la structure idéale pour implémenter une file de priorité. Il a les deux propriétés suivantes :

- le premier élément est maximal (il n'y en a pas de plus grand),
- l'extraction du premier élément et l'insertion d'un nouvel élément prennent un temps $O(\log n)$.

♦ **push_heap**

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

template<class IterAlea, class Compare>
    void push_heap(IterAlea debut, IterAlea fin, Compare comp);
```

Action : réarrange la séquence [**debut, fin** [(qui est un tas sauf pour ce qui concerne l'élément ***(fin – 1)**) pour en faire un tas.

Contrainte : la séquence [**debut, fin - 1** [doit représenter un tas valide.

♦ **pop_heap**

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

template<class IterAlea, class Compare>
    void pop_heap(IterAlea debut, IterAlea fin, Compare comp);
```

Action : échange les éléments ***debut** et ***(fin – 1)** et réarrange la séquence [**debut, fin - 1** [pour en faire un tas.

Contrainte : la séquence [**debut, fin** [doit représenter un tas valide.

- **make_heap**

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

template<class IterAlea, class Compare>
    void make_heap(IterAlea debut, IterAlea fin, Compare comp);
```

Action : réarrange la séquence [**debut, fin** [pour en faire un tas.

- **sort_heap**

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

template<class IterAlea, class Compare>
    void sort_heap(IterAlea debut, IterAlea fin, Compare comp);
```

Action : trie la séquence [**debut, fin** [, qui cesse ainsi d'être un tas.

Contrainte : la séquence [**debut, fin** [doit représenter un tas valide.

- **min**

```
template<class T>
    const T& min(const T& a, const T& b);

template<class T, class Compare>
    const T& min(const T& a, const T& b, Compare comp);
```

Retour : le plus petit des deux éléments (ou, s'ils sont équivalents, le premier).

- **max**

```
template<class T>
    const T& max(const T& a, const T& b);

template<class T, class Compare>
    const T& max(const T& a, const T& b, Compare comp);
```

Retour : le plus grand des deux éléments (ou, s'ils sont équivalents, le premier).

- **min_element**

```
template<class IterAv>
    IterAv min_element(IterAv debut, IterAv fin);

template<class IterAv, class Compare>
    IterAv min_element(IterAv debut, IterAv fin, Compare comp);
```

Retour : l'itérateur *i* le plus à gauche tel qu'aucun élément de la séquence ne soit strictement inférieur à ***i*.

- **max_element**

```
template<class IterAv>
    IterAv max_element(IterAv debut, IterAv fin);

template<class IterAv, class Compare>
    IterAv max_element(IterAv debut, IterAv fin, Compare comp);
```

Retour : l'itérateur *i* le plus à gauche tel qu'aucun élément de la séquence ne soit strictement supérieur à ***i*.

- **lexicographical_compare**

```
template<class IterIn1, class IterIn2>
    bool lexicographical_compare(IterIn1 debut1, IterIn1 fin1,
                                 IterIn2 debut2, IterIn2 fin2);
```



```
template<class IterIn1, class IterIn2, class Compare>
    bool lexicographical_compare(IterIn1 debut1, IterIn1 fin1,
                                 IterIn2 debut2, IterIn2 fin2, Compare comp);
```

Retour : le résultat de la comparaison des séquences [**debut1, fin1** [et [**debut2, fin2** [selon l'ordre lexicographique. Cet algorithme peur s'exprimer ainsi :

```
i = debut1, j = debut2;
while (i != fin1 && j != fin2 && !(*i < *j) && !(*j < *i))
    ++i, ++j;
return j == last2 ? false : (i == last1 || *i < *j);
```

- **next_permutation**

```
template<class IterBid>
    bool next_permutation(IterBid debut, IterBid fin);

template<class IterBid, class Compare>
    bool next_permutation(IterBid debut, IterBid fin, Compare comp);
```

Action : la séquence [**debut, fin** [, considérée comme représentant une permutation de ses valeurs, est remplacée par la séquence qui représente la permutation suivante dans l'ordre lexicographique des permutations. On considère que la première permutation (dans l'ordre lexicographique) est la suivante de la dernière.

Retour : **false** si la permutation était la dernière ; **true** sinon.

Exemple :

```
void main()
    {
    char s[] = "abc";

    cout << s << ' ';
    while (next_permutation(s, s + strlen(s)))
        cout << s << ' ';
    }
```

Affichage obtenu :

```
abc acb bac bca cab cba
```

Notez que cet algorithme gère assez bien le cas où la séquence n'est pas une vraie permutation : avec

```
char s[] = "aab";
```

l'affichage est

```
aab aba baa
```

- **prev_permutation**

```
template<class IterBid>
    bool prev_permutation(IterBid debut, IterBid fin);

template<class IterBid, class Compare>
    bool prev_permutation(IterBid debut, IterBid fin, Compare comp);
```

Action : la séquence [**debut, fin** [, considérée comme représentant une permutation de ses valeurs, est remplacée par la séquence qui représente la permutation précédente dans l'ordre lexicographique des permutations. On considère que la dernière permutation (dans l'ordre lexicographique) est la orécédente de la dernière.

Retour : **false** si la permutation était la première ; **true** sinon.

☺☹☹☺

Références

Livres sur C++ :

[1] B. Stroustrup, *Le langage C++ (3ème édition)*, CampusPress 1999

[2] J. Charbonnel, *Langage C++, la proposition de standard ANSI / ISO expliquée*, Masson 1996

[3] A. Géron, F. Tawbi, *Pour mieux développer avec C++*, InterEditions, 1999

[4] S. Meyers, *Le C++ efficace*, Addison-Wesley, 1994

[5] S. Dewhurst, K. Stark, *Programmer en C++*, Masson, 1990

[6] S. Lippman, *L'essentiel du C++ (2ème édition)*, Addison-Wesley, 1992

[7] M. Ellis, B. Stroustrup,
The Annotated C++ Reference Manual, Addison-Wesley, 1992

Sites web, avec de la documentation (principalement sur la bibliothèque standard) :

http://www.sgi.com/Technology/STL/

La documentation de la bibliothèque développée par Silicon Graphics et Hewlett-Packard, la principale ancêtre de la STL normalisée.

http://www.ccd.bnl.gov/bcf/cluster/pgi/pgC++_lib/stdlib.htm

Documentation de la bibliothèque développée par Rogue Wave, une des plus repandues.

http://msdn.microsoft.com/default.asp

Toute la documentation de Visual C++ et de l'implantation Microsoft de la STL

http://www.suse.de/doku/gnu/iostream/Top.html

Documentation de la bibliothèque des entrées-sorties du C++ du projet GNU

Adresse de l'auteur

Henri Garreta
Faculté des Sciences de Luminy, Département d'Informatique
163, avenue de Luminy
13288 Marseille Cedex 9
Mél. henri.garreta@lim.univ-mrs.fr

