Vincent Gouvernelle

LE GUIDE DE SURVIE

C++

L'ESSENTIEL DU CODE ET DES COMMANDES

PEARSON

C++

Vincent Gouvernelle

Publié par Pearson Education France

47 bis, rue des Vinaigriers

75010 PARIS

Tél. : 01 72 74 90 00

www.pearson.fr

Relecteurs techniques : Philippe Georges et Yves Mettier

Collaboration éditoriale : Jean-Philippe Moreux

Réalisation PAO : Léa B.

ISBN : 978-2-7440-4011-5

Table des matières

À propos de l'auteur

Vincent Gouvernelle est ingénieur en informatique, diplômé de l'ESIL à Marseille, et titulaire d'un DEA en informatique. Après une période consacrée à l'enseignement et à la recherche, au cours de laquelle il a participé à des travaux sur la paramétrisation de surfaces au sein du CNRS et du BRGM, il s'est tourné vers le monde de l'industrie. Chez Gemplus, il a travaillé sur la personnalisation des cartes à puce et l'écriture de drivers optimisés pour le chiffrement de données. Puis au sein de Coretech International, maintenant filiale de Think3, il revient à sa première spécialité, la CAO, dans le domaine de la conversion et la correction de modèles. Il travaille aujourd'hui chez Sescoi R&D, société éditrice de logiciels de CFAO.

Introduction

Il y a longtemps que je souhaitais réunir toutes les informations relatives aux algorithmes et à l'utilisation de la bibliothèque standard (STL) et de BOOST. Je me suis dit qu'il était temps de passer à l'acte lorsque l'occasion m'a été donnée d'écrire ce livre. L'objectif de la collection Guide de survie auquel ce dernier appartient est de fournir au monde informatique l'équivalent des aide-mémoire linguistiques que l'on emmène avec soi à l'étranger. Ainsi, cet ouvrage n'est pas un simple dictionnaire de fonctions ou de mots-clés : un effort particulier a été fait pour mettre en situation chacun d'eux afin de vous permettre d'en exploiter tout le potentiel dans le contexte qui est le vôtre.

Les séquences de codes (relatives aux bibliothèques standard et à BOOST) fournies dans ce livre fonctionnent avec les compilateurs mentionnés dans le tableau ci-dessous. La liste n'est bien entendu pas exhaustive. Pour ce qui est des parties employant wxWidget et QT, il suffit de recourir à une plate-forme disposant de ces bibliothèques.

Je voudrais remercier les personnes sans qui cet ouvrage ne serait pas. Pour tout le temps que je n'ai pas pu leur consacrer pendant la rédaction de ces pages, je remercie mon épouse Audrey et mes trois enfants. Merci également à Patricia Moncorgé de la confiance qu'elle m'a accordée pour la réalisation de ce projet, aux relecteurs techniques Philippe Georges et Yves Mettier et au correcteur Jean-Philippe Moreux. Merci enfin à Yves Bailly qui fut le premier à m'encourager dans cette aventure.

J'espère que cet ouvrage comblera vos attentes de programmeur, que vous soyez débutant ou chevronné.

Compilateurs

Plate-forme	Compilateurs
Windows	Visual C++ (7.1 avec SP1 *aka* 2003, 8.0 *aka* 2005, 9.0 *aka* 2008), Intel C++ (10.1), Comeau C++ (4.3), MingGW, Cygwin
Linux	GCC (3.4, 4.0, 4.1, 4.2, 4.3), Intel C++ (8.1, 9.0, 9.1, 10.0), QLogic (3.1), Sun Compiler (5.9, 5.10 avec stdcxx)
Mac OS X	GCC 4 (pour PowerPC ou Intel)
HP-UX	GCC 4.2, HP C/aC++, HP aCC
IBM AIX	IBM XL C/C++ (10.1)
True64	Compaq C++ (7.1)
Sun Solaris	Sun C++ (5.7, 5.8, 5.9), GCC (3.4)

1

Bases héritées du langage C

Le langage C++ est une évolution du langage C. De ce fait, une partie de la syntaxe est commune à ces deux langages. Ce chapitre résume rapidement ces points communs quelque peu améliorés au passage. Pour plus de renseignements sur le langage C, vous pouvez vous référer au *Guide de survie – Langage C* d'Yves Mettier (Pearson, 2007).

Hello world en C

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    printf("Hello world!");
    return 0;
}
```

La première ligne est une inclusion d'un fichier d'en-tête de la bibliothèque C standard. Cela permet d'utiliser la fonction `printf()` pour écrire le message « Hello world! »

sur la console. La deuxième ligne correspond au point d'entrée standard de tout programme C (et C++), c'est-à-dire que cette fonction est automatiquement appelée lors du lancement du programme. Notez toutefois que le nommage et les paramètres de ce point d'entrée peuvent varier suivant les systèmes. Par exemple, Windows a ajouté le point d'entrée suivant :

```
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPWSTR    lpCmdLine,
                   int nShowCmd);
```

Enfin, le `return 0 ;` indique que le programme s'est déroulé sans erreur.

Compiler avec gcc

Pour compiler cet exemple, vous pouvez utiliser gcc en ligne de commande. Par exemple avec mingw ou cygwin sous Windows, ou toute autre version de gcc (disponibles sous GNU/Linux, Unix, Mac OS X, etc.) :

```
gcc helloworld.c -o helloworld.exe
```

L'option `-c` permet de compiler un fichier source et de créer un fichier objet, réutilisable par la suite.

L'option `-I` permet d'ajouter des chemins de recherche supplémentaires pour les fichiers d'en-tête.

L'option `-lbibliotheque` permet de *linker* avec la `bibliotheque` donnée.

L'option `-L` permet d'ajouter des chemins de recherche supplémentaires pour les bibliothèques.

Pour plus de simplicité, il existe aussi des environnements intégrés pour le compilateur GNU. Je vous invite à jeter un œil sur l'excellent DevCPP (**www.bloodshed.net/devcpp.html** disponible pour Windows), CodeBlocks (www.codeblocks.org, disponible pour Windows, Linux et Mac OS X) ou encore Anjuta (**anjuta.sourceforge.net** disponible pour GNU/Linux).

Compiler avec Microsoft Visual C++

Ouvrez l'environnement de développement et laissez-vous guider par les assistants de création de projet. Pour cet exemple, choisissez une application en mode console.

Commentaires

```
/* commentaire C
   possible sur plusieurs lignes */
// commentaire C++ sur une seule ligne
```

Les commentaires C commencent avec une barre oblique et un astérisque `/*` et finissent par un astérisque et une barre oblique `*/`. Leur contenu peut s'étaler sur plusieurs lignes et commencer ou finir en milieu de ligne.

Le commentaire C++ commence dès l'apparition de deux barres obliques `//` et finit automatiquement à la fin de la ligne.

Types fondamentaux

```
char caractere;
short entier_court;
int entier;
long entier_long;
float flottant_simple_precision;
double flottant_double_precision;
```

```
signed Type v;
unsigned Type v;
```

```
void
```

Le langage C (et donc C++) comprend de nombreux types de nombres entiers, occupant plus ou moins de bits. La taille des types n'est que partiellement standardisée : le standard fixe uniquement une taille minimale et une magnitude minimale. Les magnitudes minimales sont compatibles avec d'autres représentations binaires que le complément à deux, bien que cette représentation soit presque toujours utilisée en pratique. Cette souplesse permet au langage d'être efficacement adaptés à des processeurs très variés, mais elle complique la portabilité des programmes écrits en C/C++.

Chaque type d'entier a une forme signée (pouvant représenter un nombre positif ou négatif, communément appelés « nombres relatifs ») et une forme non signée (ne pouvant représenter que des nombres positifs communément appelés « nombres naturels »). Par défaut, les types entiers sont signés, rendant le qualificateur `signed` optionnel. Le type `char` est un type entier comme les autres, mis à part le fait que le standard ne précise pas s'il est signé ou non par défaut.

Modèle de données et taille des entiers (en bits)

Modèle	short	int	long	pointeur
LP32	16	16	32	32
ILP32	16	32	32	32
LP64	16	32	64	64
ILP64	16	64	64	64

Limite des types fondamentaux (modèle ILP32)

Type	Taille (octets)	min	max
`char`	1	–128	127
`unsigned char`	1	0	255
`short`	2	–32 768	32 767
`unsigned short`	2	0	65 535
`int`	4	–2 147 483 648	2 147 483 647
`unsigned int`	4	0	4 294 967 295
`long`	4	–2 147 483 648	2 147 483 647
`unsigned long`	4	0	4 294 967 295
`long long`	8	–92 233 720 368 547 758 079	223 372 036 854 775 807
`unsigned long long`	8	0	18 446 744 073 709 551 615
`float`	4	-10^{255}	10^{255}
`double`	8	-10^{2047}	10^{2047}

Info

Le mot-clé `register` permet d'indiquer au compilateur d'utiliser un registre du processeur. Ceci n'est qu'une indication fournie au compilateur et non une garantie.

Le type `void` est utile pour le type pointeur sur type inconnu `void*` et pour la déclaration de procédure (une fonction retournant `void` est une procédure).

Types élaborés

```
struct NomStructure { … };
union NomUnion { … };
enum NomEnum { … };
Type *pointeur;
Type tableau[taille];
Type fonction(parametres);
```

```
typedef Type nouveauNom;
typedef Type (*MonTypeDeFonction)(…);
```

Le C++ a apporté son lot d'amélioration quant à la manière de déclarer de nouveaux types `struct`, `union` et `enum`. Il suffit maintenant d'utiliser le nom de la structure ainsi définie sans devoir systématiquement répéter le mot-clé qui la précède. Il n'est plus besoin de recourir de recourir à un alias (`typedef`) pour parvenir au même résultat. C'est cette syntaxe allégée qui est présentée ici.

Les structures (`struct`) permettent de regrouper plusieurs variables dans une sorte de groupe et de les rendre ainsi indissociables, comme s'il s'agissait d'un nouveau type.

Dans ce cas, le nom utilisé pour chaque variable dans le bloc devient le moyen d'accéder à celle-ci.

```
struct Personne
{
    char *nom, *prenom;
    int age;
};
Personne personne = { "Alain", "Dupont", 54 };
personne.age = 35;
```

Les unions de type (`union`) permettent de voir une même donnée (binaire) de différentes manières. La syntaxe est la même que les `struct` mais son objectif est différent.

```
union IP
{
    unsigned int adresse;
    struct { unsigned char a,b,c,d; };
};
IP masque;
masque.adresse = 0; // masque.a, .b, .c et .d == 0
masque.a = 255;
masque.b = 255;
masque.c = 255; // masque <=> 255.255.255.0
printf("%X\n", masque.adresse); // => affiche FFFFFF00
```

Les énumérations (`enum`) permettent d'associer des noms à des valeurs entières. Seul le type `int` est supporté par le langage (la future norme C++0x lèvera cette limitation). Elles permettent ainsi de rendre votre code source beaucoup plus lisible et mieux contrôlé par le compilateur (par exemple lors d'un `switch` sur une valeur de type énumération, le compilateur génère un *warning* si aucune clause `default`

n'est présente alors que toutes les valeurs de l'énumération ne sont pas présentes).

```
enum Etat { Vrai = 1, Faux = 0, Inconnu = -1 };
```

Par défaut, toutes les valeurs se suivent dans l'ordre croissant (si aucune valeur n'est mentionnée), en commençant par la valeur zéro. L'exemple suivant illustre ce mécanisme :

```
enum Nombre { Zero, Un, Deux, Quatre = 4, Cinq, Sept = 7,
➥ Huit, Neuf };
```

Les crochets `[]` permettent de créer des tableaux (de taille fixe).

```
int vecteur[3]; // trois entiers
int matrice[3][3]; // neuf entiers
```

Le mot-clé `typedef` permet de créer des *alias* sur d'autres types afin de pouvoir les utiliser plus facilement ou bien d'en créer de nouveau en leur donnant ainsi un nouveau nom.

```
typedef double Reel;        // un réel de IR et codé
                               ➥ en double
typedef Reel Vecteur[3];    // vecteur de IR3
typedef Reel Matrice[3][3]; // matrice de transformation
Vecteur v,v2;
Matrice m;
v[0] = 3.0; v[1] = 5.0; v[2] = 1.0;
m[0][0] = 1.0 ; /*…*/ m[2][2] = 1.0;
```

Structures conditionnelles

```
if (test)
    instruction;
```

```
if (test)
    instruction;
else
    instruction;
```

```
switch (instruction)
{
case valeur1:
   instruction;
   [ break; ]
case valeur2:
   …
[ default:
    [ instruction;
     [ break; ]
    ]
]
}
```

Les structures conditionnelles permettent d'exécuter une instruction ou une autre en fonction d'un test. Les parenthèses entourant le test sont obligatoires. Un test est une instruction dont la valeur est entière ou booléenne. Dans le cas d'une valeur entière, toute valeur non nulle est considérée comme étant vraie.

Lorsque l'instruction à exécuter (pas celle du test mais celle choisie en fonction du test) est un bloc de code (entre accolade { … }), il ne faut pas mettre de point-virgule après celui-ci. Le code suivant l'illustre sur un `if` :

```
if (test)
{
    // bloc1
}
else
{
    // bloc 2
}
```

La structure conditionnelle `switch` est une structure de branchement. Les valeurs utilisées pour les points d'entrée des branchements (`case`) doivent être des constantes et ne peuvent pas être des chaînes. Lorsqu'un point d'entrée est choisi par le test, l'exécution se poursuit à cet endroit et ne sort du `switch` que lorsqu'un `break` est rencontré (et pas lors du prochain `case`).

Attention

Il est interdit d'effectuer une déclaration de variable dans un case. Si vous n'avez pas d'autre choix (pour une variable temporaire), faites-le alors dans un bloc d'instruction. Mais ce dernier ne peut pas contenir de case.

```
int i = …;
switch (i)
{
case 1:
case 2: // si i==1 ou i==2, on se retrouve ici
    {
```

```
        // on utilise un bloc si besoin de déclarer des
          ➥ variables temporaires
        int j = …;
        i += 3 * j;
    }
    break;
case 3: // si i==3, on se retrouve là
    i += 4;
    if ( une_fonction_test(i) )
        break; // si une_fonction_test(i) est vraie, on
        ➥ finit le switch
default: // si i<1 ou i>3
         // ou que le test dans le 'cas 3' était faux
         // on se retrouve ici
    i /= 3;
}
```

Opérateurs de comparaison

Symbole	Signification
`a == b`	Vrai si `a` est égal à `b`
`a != b`	Vrai si `a` est différent de `b`
`a < b`	Vrai si `a` est inférieur à `b`
`a > b`	Vrai si `a` est supérieur à `b`
`a <= b`	Vrai si `a` est inférieur ou égal à `b`
`a <= b`	Vrai si `a` est supérieur ou égal à `b`

Opérateurs logiques

Symbole	Signification
`a && b`	Vrai si `a` et `b` sont vrais
`a || b`	Vrai si `a` ou `b` est vrai (l'un, l'autre ou les deux)
`! a`	Vrai si `a` est faux

Opérateurs binaires

Symbole	Signification
`a & b`	ET binaire
`a | b`	OU binaire
`a ^ b`	OU EXCLUSIF binaire

Structures de boucle

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

```
for ( initialisation ; test; incrément )
    instruction;
```

```
do
{
    instruction;
}
while (test);
```

```
break;
continue;
```

Dans tous les cas, l'*instruction* est exécutée tant que le *test* est vrai. Il est possible de modifier le cours de l'exécution de l'*instruction* lorsque celle-ci est un bloc, à l'aide des mots-clés `continue` et `break` :

- `continue` interrompt l'exécution du bloc et revient au test de la structure de boucle ;
- `break` interrompt l'exécution du bloc et sort de la structure de boucle ;
- lorsque plusieurs structures de boucle sont imbriquées, ces sauts se réfèrent à la boucle dans laquelle ils se trouvent, et pas aux niveaux supérieurs.

Le code ci-après montre comment écrire une boucle `for` avec un `while`. La définition de bloc entourant le code est nécessaire pour l'écriture de son équivalence avec `while`, car en C++ l'initialisation est locale à la structure de boucle `for`.

```
{
    initialisation;
    while (test)
        instruction;
}
```

La structure `do … while` permet de garantir que l'instruction est exécutée au moins une fois. C'est un peu comme si la même instruction apparaissait avant et dans un `while` traditionnel, comme dans l'exemple suivant :

```
instruction;
while (test)
    instruction;
```

Sauts

```
étiquette:
goto étiquette;
```

Les sauts permettent de sauter d'un endroit à un autre dans le fil d'exécution des instructions d'un programme. Les étiquettes peuvent être dotées de tout nom respectant les mêmes contraintes syntaxiques que celles d'un identificateur.

Il n'est pas possible d'effectuer des sauts en dehors d'une fonction. En revanche, il est possible d'effectuer des sauts en dehors et à l'intérieur des blocs d'instructions sous certaines conditions (voir l'avertissement ci-après). Si au terme d'un saut, on sort de la portée d'une variable, celle-ci est détruite. Enfin, il est impossible de faire un saut dans un bloc `try {}` (voir le Chapitre 7 consacré aux exceptions).

Attention

Les sauts sont fortement déconseillés. Vous avez certainement déjà entendu ou entendrez certainement que l'on peut toujours s'en passer. Cela est vrai, à quelques exceptions près. Toutefois, leur utilisation rend parfois le code plus lisible. Si vous pensez qu'il est légitime de les utiliser, ne vous en privez pas. Mais n'en abusez pas pour autant, car ils cachent certains pièges, surtout avec le C++ ! En effet, si la déclaration de saut se trouve après une déclaration, cette déclaration ne doit pas contenir d'initialisations et doit être un type simple (comme les variables, les structures ou les tableaux). Vous l'aurez donc compris, l'utilisation de sauts avec des classes peut se révéler délicate.

Fonctions

```
type identificateur(paramètres)
{
    … // instructions
}
```

```
return [ instruction ];
```

Les paramètres d'une fonction sont de la forme *type* `variable` ou *type* `variable = valeurParDefaut`, séparés par une virgule. Attribuer une valeur par défaut n'est possible que pour le (ou les) dernier(s) paramètre(s consécutifs). Si le type de retour d'une fonction est `void`, alors il s'agit d'une procédure.

La valeur de retour d'une fonction est donnée par l'instruction `return`. Cette instruction fait sortir immédiatement de la fonction (ou de la procédure).

Info

Il est possible d'écrire des fonctions acceptant un nombre variable de paramètres grâce au fichier d'en-tête `stdarg.h`. Le code ci-après montre un exemple. Sachez toutefois qu'un tel code n'est pas toujours portable et que l'implémentation de cette fonctionnalité varie suivant les compilateurs.

```
#include <stdarg.h>
double somme(int quantite, ...)
{
    double res = 0.0;
    va_list varg;
    va_start(varg, quantite);
    while (quantite-- != 0)
        res += va_arg(varg, double);
```

```
    va_end(varg);
    return res;
}
```

La fonction `printf()` utilise cette technique. Le nombre, l'ordre et le type des arguments sont indiqués par le premier paramètre correspondant à la chaîne de formatage des données.

Préprocesseur

```
// instructions
#include «fichier»
#include <fichier>

#define NOM code
#define NOM(a [, …]) code
#undef NOM

#if test
#ifdef NOM    // équivalent à #if defined NOM
#ifndef NOM   // équivalent à #if not defined NOM
#elif test // C89
#else
#endif

#pragma … // C89
#error "Message"
#warning "Message"

#line numéro [fichier]
```

```
// constantes
__FILE__
__LINE__
__FUNCTION__ //  alternative: __func__
__DATE__
__TIME__
__cpluplus
```

La directive `#include` permet d'inclure le fichier mentionné à cet endroit. Lorsque le nom est entre guillemets, le fichier spécifié est recherché dans le répertoire courant d'abord, puis de la même manière qu'avec les crochets. Si le nom de fichier est entre crochets, le fichier est recherché dans les répertoires (ou dossiers) spécifiés dans les options d'inclusion (`-I` avec g++) puis dans les chemins de recherche des en-têtes du système. Le fichier inclus est lui aussi traité par le préprocesseur.

Le préprocesseur définit un certain nombre de constantes :

- `__LINE__` donne le numéro de la ligne courante ;
- `__FILE__` donne le nom du fichier courant ;
- `__DATE__` renvoie la date du traitement du fichier par le pré-processeur ;
- `__TIME__` renvoie l'heure du traitement du fichier par le pré-processeur ;
- `__cpluplus` est défini lors d'une compilation en C++. Il permet de distinguer les parties de code écrites en C++ de celles écrites en C. En principe, la valeur définie correspond à la norme du langage supportée par le compilateur. Par exemple, pour la norme de novembre 1997, la valeur sera `199711L`.

Astuce

Pour convertir un paramètre de macro, il est parfois nécessaire d'imbriquer deux macros pour obtenir un résultat fonctionnant correctement. C'est le cas par exemple lors de la transformation d'un paramètre en sa chaîne de caractères correspondante (par l'instruction #). Le code suivant montre comment s'en sortir dans ce cas :

```
#define TO_STR1(x) #x
#define TO_STR(x) TO_STR1(x)
```

La syntaxe ## permet de concaténer deux paramètres. Le code suivant permet d'obtenir un nom unique. Cette macro, utilisée plusieurs fois sur une même ligne, générera le même nom.

```
#define UNIQUE_NAME2(name,line)  name##line
#define UNIQUE_NAME1(name,line)  UNIQUE_NAME2(name,line)
#define UNIQUE_NAME(name)        UNIQUE_NAME1(name,__LINE__)
```

Les directives de compilation sont couramment utilisées pour la protection des fichiers d'en-tête contre les inclusions multiples :

```
#ifndef MON_HEADER
#define MON_HEADER
// inclus une seule fois
#endif
```

Opérateurs et priorité (C et C++)

Il est parfois difficile de se souvenir de l'ordre de priorité des opérateurs entre eux. Voici donc une liste les rassemblant de la plus haute priorité (on commence par évaluer ceux-ci) à la plus basse (le compilateur finira par ceux-là).

Si toutefois vous aviez un doute, ou tout simplement parce que vous trouvez cela plus lisible, n'hésitez pas à utiliser les parenthèses (…) pour forcer l'ordre d'évaluation.

Un groupe rassemble les opérateurs ayant même priorité, c'est-à-dire que leur évaluation se fait dans l'ordre de lecture (sauf mention spéciale).

J'y ai également inclus les opérateurs ajoutés par le C++ afin de tous les réunir en un seul endroit.

Opérateurs par ordre de priorité

Opérateur	Signification
Groupe 1 (pas d'associativité)	
`::`	Opérateur de résolution de porté (C++)
Groupe 2	
`()`	Modification de la priorité des opérateurs ou appel de fonction
`[]`	Élément d'un tableau
`.`	Champ de structure ou d'union (ou de fonction membre en C++)
`->`	Champ désigné par pointeur (sélection de membre par déréférencement)
`++`	Incrémentation (post-fixe, par exemple `++i`)
`--`	Décrémentation (post-fixe, par exemple `--i`)
`type(...)`	Transtypage explicite (C++)
`new`	Création dynamique d'objets (C++)
`new[]`	Création dynamique de tableaux (C++)
`delete`	Destruction des objets créés dynamiquement (C++)
`delete[]`	Destruction des tableaux créés dynamiquement (C++)

Opérateurs par ordre de priorité *(suite)*

Opérateur	Signification
Groupe 3 (associativité de droite à gauche)	
`!`	Négation booléenne
`~`	Complément binaire
`+`	Plus unaire
`-`	Opposé (appelé aussi moins unaire)
`++`	Incrémentation (préfixe, par exemple `i++`)
`--`	Décrémentation (préfixe, par exemple `i--`)
`&`	Adresse (pas le « et » binaire)
`*`	Accès aux données indiquées par un pointeur (déréférencement)
`sizeof(...)`	Taille en nombre d'octets de l'expression
`typeid(...)`	Identification d'un type (C++), pas toujours implémenté
`(type)`	Transtypage (*cast*)
`const_cast`	Transtypage de constante (C++)
`dynamic_cast`	Transtypage dynamique (C++)
`reinterpret_cast`	Réinterprétation (C++)
`static_cast`	Transtypage statique (C++)
Groupe 4	
`.*`	Sélection de membre par pointeur sur membre (C++)
`->*`	Sélection de membre par pointeur sur membre par déréférencement

Opérateurs par ordre de priorité *(suite)*

Opérateur	Signification
Groupe 5	
*	Multiplication
/	Division
%	Modulo (reste de la division euclidienne)
Groupe 6	
+	Addition
-	Soustraction
Groupe 7	
<<	Décalage à gauche
>>	Décalage à droite
Groupe 8	
<	Test strictement inférieur à
<=	Test inférieur ou égal à
>	Test supérieur à
>=	Test supérieur ou égal à
Groupe 9	
==	Test égal à
!=	Test différent de
Groupe 10	
&	ET binaire
Groupe 11	
^	OU eXclusif (XOR) binaire

Opérateurs par ordre de priorité *(suite)*

Opérateur	Signification
Groupe 12	
|	OU binaire
Groupe 13	
&&	ET logique (ou booléen)
Groupe 14	
||	OU logique (ou booléen)
Groupe 15	
… ? … : …	Opérateur conditionnel
Groupe 16 (associativité de droite à gauche)	
=	Affectation
+=	Incrémentation et affectation
-=	Décrémentation et affectation
*=	Multiplication et affectation
/=	Division et affectation
%=	Modulo et affectation
<<=	Décalage à gauche et affectation
>>=	Décalage à droite et affectation
&=	ET binaire et affectation
|=	OU binaire et affectation
^=	XOR binaire (OU exclusif) et affectation
Groupe 17	
, (virgule)	Séparateur dans une liste d'expressions

2

Bases du langage C++

Le C++ est un langage de programmation permettant la programmation sous de multiples paradigmes, comme par exemple la programmation procédurale (héritée du langage C), la programmation orientée objet (voir le Chapitre 4) et la programmation générique (voir le Chapitre 5). Depuis 1995 environ, C++ est le langage le plus utilisé au monde.

Avant d'aborder des notions plus complexes, ce chapitre se concentre sur les bases du C++, en plus de celles héritées du langage C.

Hello world en C++

```
#include <iostream>
int main(int argc, char* argv[])
{
    std::cout << "Hello world !" << std::endl;
    return 0;
}
```

Voici le pendant C++ du traditionnel *HelloWorld* du langage C, que nous avons vu dans le chapitre précédent.

Notez que par habitude, l'extension du fichier n'est plus « .c » mais « .cpp » (on rencontre aussi « .cxx » ou encore « .C » surtout sous Unix et GNU-Linux, qui distinguent minuscules et majuscules dans les noms de fichiers). Pour les fichiers d'en-têtes C++, la STL (bibliothèque standard du C++) a simplement supprimé l'extension (plus de « .h ») ; mais aujourd'hui, l'extension standard est « .hpp » (que l'on retrouve dans la bibliothèque BOOST).

Si vous avez réussi à compiler ce programme en C, vous y parviendrez sans difficulté en C++. Par contre, il ne faut plus invoquer `gcc` mais `g++`.

Les mots-clés

Voici la liste des mots réservés du C++ qui ne sont pas déjà présents dans le langage C. Elle vous sera utile pour retrouver facilement la section associée.

Les mots-clés du C++

Mot-clé	Page	Mot-clé	Page
`bool`	29	`inline`	46
`catch`	123	`mutable`	72
`class`	62	`namespace`	40
`const_cast`	34	`new`	113
`delete`	113	`operator`	37
`dynamic_cast`	36	`private`	75
`explicit`	71	`protected`	75
`false`	29	`public`	75
`friend`	66	`reinterpret_cast`	35

Les mots-clés du C++ *(suite)*

Mot-clé	Page	Mot-clé	Page
`static_cast`	33	`typeid`	86
`template`	96	`typename`	101
`this`	67	`using`	41
`throw`	123	`virtual`	79
`true`	29	`wchar_t`	31
`try`	123		

Les constantes

Une constante est une valeur qui ne change pas au cours de l'exécution d'un programme.

Une constante ressemble à une macro par différents aspects :

- sa portée est réduite au fichier où elle est déclarée ;
- les expressions l'utilisant sont évaluées à la compilation ;
- elle peut être utilisée pour définir la taille d'un tableau de type C.

Toutefois, contrairement aux macros, le compilateur peut allouer un emplacement mémoire où stocker la constante lorsque cela est requis. C'est par exemple le cas lorsque l'on utilise son adresse :

```
const int constante_entiere = 345;
const int *ptr_sur_constante = &constante_entiere;
```

Si les macros C `#define` sont toujours disponibles, il est préférable d'utiliser le concept de constantes qu'offre le C++. Cela permet une vérification de type à la compilation et diminue le risque d'erreur.

```
const int a = 2;
int const b = 7;       // équivalent à la ligne précédente
const Objet unobjet;   // un objet peut être une constante
const int const c = 7; // ERREUR : const dupliqué
const double d;        // ERREUR : initialisation manquante
```

Pour les pointeurs, le qualificatif const peut aussi bien s'appliquer au pointeur qu'à l'élément pointé :

```
const int entier1;
int* ptr = &entier1;   // ERREUR : conversion invalide

// Pointeur sur une constante
const int *a = &constante_entiere;
int const *b = &constante_entiere; // équivalent
                                   // à la ligne précédente
a = &constante_entiere;
*a = 4;          // ERREUR : on ne modifie pas une constante

// Pointeur constant sur une variable
int *const c = &entier;
int *const c;   // ERREUR : initialisation manquante
c = &entier1;   // ERREUR : on ne modifie pas une constante
*c = 5;

// Pointeur constant sur une constante
const int *const d = &constante_entiere;
int const *const e = &constante_entiere;  // équivalent
                                  // à la ligne précédente
const int *const f;   // ERREUR : initialisation manquante
d = &entier1;     // ERREUR : on ne modifie pas une constante
*d = 5;           // ERREUR : on ne modifie pas une constante
```

Déclarations de variables

En C++, la déclaration de variables est considérée comme une instruction. Il n'est pas nécessaire de regrouper toutes les déclarations de variables en début de bloc comme en C.

```
int a;       // déclaration d'une variable
int b = 0;   // déclaration et initialisation
Objet o;     // déclaration et appel du constructeur
```

Attention

La déclaration de variables peut entraîner l'exécution de beaucoup de code, notamment lors de l'initialisation des objets (classes).

Une variable déclarée dans une boucle `for` implique que sa portée reste limitée à cette dernière :

```
for (int i=0 ; i<n ; ++i)
    // …
for (int i=k ; i>=0 ; --k)
    // …
```

Attention

Certains vieux compilateurs C++, comme Visual C++ 6, ne gèrent pas cette spécificité et transforment toute déclaration normalement locale à la boucle `for` en une déclaration précédant celle-ci. Ainsi, le code ci-dessus ne compilerait pas avec un tel compilateur : un message d'erreur prétextant que la variable `i` est déjà déclarée serait généré pour la deuxième boucle.

Pour vous assurer qu'une variable a une portée limitée à un fichier, utilisez un espace de nom anonyme plutôt que de recourir au mot-clé `static` :

```
static int n;          // correcte mais de style C
namespace              // on préférera le style C++
{
    int n;
}
```

L'utilisation de `static` pour une déclaration de variable dans un bloc a le même effet qu'en C : l'initialisation n'a lieu qu'une fois et son comportement est semblable à celui d'une variable globale uniquement visible dans ce bloc.

```
int f(int x)
{
    static int a = x;
    return a;
}
int main(void)
{
    cout << f(10) << endl ;
    cout << f(12) << endl ;
}
```

Produira la sortie suivante :

```
10
10
```

Les nouveaux types de variables du C++

bool

Le type `bool` accepte deux états :

- `true` (vrai) : correspond à la valeur 1 ;
- `false` (faux) : correspond à la valeur 0.

Ce type est codé sur le même nombre de bits que le type `int`. Lorsque l'on convertit un type numéraire en `bool`, toute valeur non nulle est considérée comme `true`.

```
bool vrai = true;
bool faux = false;
```

Les références

Une référence est une sorte de pointeur masqué. Pour déclarer un pointeur, il suffit de rajouter un et commercial (`&`) après le type : si `T` est un type, le *type référence* sur `T` est `T&`.

```
int a = 3;
int& r = a ; // r est une référence sur a
```

Il est obligatoire d'initialiser une référence lors de sa déclaration, sinon vous obtiendrez un message d'erreur. En effet, `r = b;` ne transformera pas `r` en une référence sur `b` mais copiera la valeur de `b` dans `r`.

Astuce

Les références permettent de définir des raccourcis (ou *alias*) sur des objets.

```
int& element = tableau[indice];
element = entier;
```

est équivalent à :

```
tableau[indice] = entier;
```

Attention

Une fonction ne doit jamais renvoyer une référence sur un objet qui lui est local.

```
int& fonction(…)
{
    int res;
    // …
    return res; // retourne une référence sur un objet
                // qui sera aussitôt détruit !
}
```

De la même manière, il faut se méfier. Dans certains cas, une référence retournée par une fonction peut s'avérer peu stable.

```
class  Pile
{
    // …
public:
    double& sommet() const { return m_valeurs[m_niveau_
    ➥ courant - 1] ; }
      double depiler() { return m_valeurs[--m_niveau_
      ➥ courant] ; }
};
Pile pile;
//…
pile.sommet() = pile.depiler() + 10.0; // Le résultat
est imprévisible !
```

enum, struct et union

```
enum MonType { mtValeur1, mtValeur2, mtValeur3 };
MonType uneInstance = mtValeur2;
```

```
struct MaStructure { /* … */ };
MaStructure a;
```

```
union MonUnion { /* … */ };
MonUnion u;
```

Déjà connu en C, la déclaration d'un `enum`, d'une `struct` ou d'une `union` se trouve simplifiée en C++. En effet, il n'y a plus besoin de répéter le mot-clé lors d'une instanciation ; l'identifiant choisi lors de la déclaration suffit.

Nous ne détaillerons pas ici la syntaxe de leur contenu, qui reste équivalente au langage C.

Info

C'est un peu comme si en C, toute déclaration était automatiquement suivie de :

```
typedef enum MonType MonType;
```

Il convient par là de faire attention au masquage local de toute autre déclaration de niveau supérieur.

wchar_t

`wchat_t` est le type utilisé pour les jeux de caractères étendus tel Unicode. Contrairement au C ou ce type est défini par un `typedef` (*via* l'inclusion du fichier d'en-tête `<stddef.h>`), en C++ `wchar_t` est bel et bien un mot-clé du langage.

Conversion de type C

À l'origine, le comité de standardisation C++ prévoyait de supprimer la conversion de type C. Cette dernière n'a été conservée que par soucis de réutilisation de code ancien et de compatibilité (un compilateur C++ doit pouvoir compiler du C). C'est pourquoi tout programmeur C++ est encouragé à la bannir et à plutôt utiliser les nouveaux opérateurs de conversion.

Pourquoi les conversions de type C sont-elles maintenant considérées comme obsolètes ? Voyez plutôt :

```
void *p = &x;
int n = (int) p;   // conversion  de type C
```

Cette conversion de type C, inoffensive de premier abord, recèle en fait plusieurs dangers. Premièrement, elle effectue des opérations différentes selon le contexte. Par exemple, elle peut transformer de manière sûre un `int` en `double`, mais elle peut aussi effectuer des opérations intrinsèquement dangereuses comme la conversion d'un `void*` en valeur numérique (voir l'exemple ci-dessus). En relisant un code source contenant une telle conversion, un programmeur ne pourra pas toujours déterminer si celle-ci est sûre ou non, si le programmeur d'origine a fait une erreur ou non.

Pire, une conversion de type C peut effectuer plusieurs opérations en une. Dans l'exemple suivant, non seulement un `char*` est converti en `unsigned char*`, mais en plus le qualificateur `const` est éliminé en même temps :

```
const char *msg = "une chaine constante";
unsigned  char *ptr = (unsigned char*) msg;
                              ➥// est-ce intentionnel ?
```

Encore une fois, il est impossible de dire si c'est la volonté du programmeur d'origine ou un oubli. Ces problèmes

relatifs à ce type de conversion sont connus depuis des années. C++ offre une meilleure solution avec les opérateurs de conversion. Ils rendent explicite l'intention du programmeur et préservent la capacité du compilateur à signaler les bogues potentiels précités.

Conversion avec static_cast

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

C'est la conversion la plus proche de l'ancienne conversion C. Elle reste éventuellement dangereuse, mais *rend explicite l'intention du programmeur.* Sont principalement autorisées, en plus des conversions standard, les conversions :

- d'un type entier vers un type énumération ;
- de `B*` vers `D*`, où `B` est une classe de base accessible de `D` (la réciproque est une conversion standard).

Elle peut être utilisée même lorsqu'une conversion implicite existe :

```
bool b = true;
int n = static_cast<int>(b);
```

Dans d'autres cas, l'utilisation de `static_cast` est obligatoire, par exemple lors de la conversion à partir d'un `void*` :

```
int n = 0;
void *ptr = &n;
int *i = static_cast<int*>(ptr); // obligatoire
```

`static_cast` utilise l'information disponible à la compilation pour effectuer la conversion de type requise. Ainsi, la

source et la destination peuvent ne pas avoir la même représentation binaire, comme lors de la conversion d'un `double` vers un `int` où l'opérateur de conversion effectue le travail nécessaire à une conversion correcte.

Utiliser `static_cast` permet d'éviter certains écueils comme :

```
const char *msg = "une chaine constante";
unsigned  char *ptr = static_cast<unsigned char*>( msg);
➥// erreur
```

Cette fois, le compilateur signale une erreur indiquant qu'il est impossible d'enlever le qualificateur `const` avec ce type de conversion. Il en est de même avec le qualificateur `volatile`.

Conversion avec `const_cast`

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

Enlever ou ajouter une qualification `const` ou `volatile` requiert l'opérateur de conversion `const_cast`. Notez que le type et le type de l'expression doivent être les mêmes, aux qualificatifs `const` et `volatile` près, sinon le compilateur générera un message d'erreur.

```
struct A
{
    void fonction(); // fonction membre non const
};
void ma_fonction(const A& a)
{
    a.func();   // erreur : appel à une fonction non const
```

De manière évidente, il s'agit d'une erreur de conception. La fonction `fonction()` aurait dû être déclarée `const`.

Néanmoins, un tel code existe parfois dans des bibliothèques existantes mal conçues. Un programmeur inexpérimenté sera tenté d'utiliser une conversion brute à la C. Pour régler un tel problème, il est préférable de supprimer le qualificateur `const` ainsi :

```
A &ref = const_cast<A&>(a) ; // enlève const
ref.fonction();  // fonctionne maintenant correctement
```

Attention

Souvenez-vous toutefois que si `const_cast` permet de supprimer le qualificateur `const`, vous ne devez pas pour autant vous autoriser à modifier l'objet. Sinon, attendez-vous à de mauvaises surprises...

Conversion avec `reinterpret_cast`

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

À l'opposé de `static_cast`, `reinterpret_cast` effectue une opération relativement dangereuse ou non portable. `reinterpret_cast` ne change pas la représentation binaire de l'objet source. Toutefois, il est souvent utilisé dans des applications bas niveau qui convertissent des objets en d'autres données transmises à un flux d'octets (et *vice versa*). Dans l'exemple suivant, `reinterpret_cast` est utilisé pour *tromper* le compilateur, permettant au programmeur d'examiner les octets à l'intérieur d'une variable de type `float` :

```
float f=123;
unsigned char *ptr = reinterpret_cast<unsigned char*>(&f);
for (int i=0 ; i<4 ; ++i)
    cout << ptr[i] << endl;
```

Utilisez l'opérateur `reinterpret_cast` pour expliciter toute conversion potentiellement dangereuse (et probablement non portable).

Info

En quoi l'exemple ci-dessus peut-il être non portable ? Imaginez une sauvegarde/lecture fichier d'un `float` ou d'un `int` avec un tel code. Sauvez votre valeur sur une machine *little endian* puis rechargez le fichier de sauvegarde sur une machine de type *big endian*. Vous ne retrouverez probablement pas la valeur d'origine...

Conversion avec `dynamic_cast`

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

`dynamic_cast` diffère des trois autres opérateurs. Il utilise les informations de type d'un objet pendant l'exécution du programme plutôt que celles connues à la compilation (pour plus d'information à ce sujet, voir la section « Obtenir des informations de type dynamiquement » du Chapitre 4). Deux scénarios requièrent l'utilisation de `dynamic_cast` :

- la conversion de spécialisation (ou *downcast*) lors de la conversion d'une référence ou d'un pointeur de classe vers une référence ou un pointeur d'une classe dérivant de la classe que l'on veut downcaster ;
- la conversion transversale (ou *crosscast*) lors de la conversion d'un objet d'héritage multiple vers une de ses classes de base.

Surcharge

La surcharge de fonctions (les opérateurs sont des fonctions) est une nouveauté apportée par le C++. Ce mécanisme, apparemment fort simple, suit des règles strictes dont le résultat n'est pas toujours intuitif. Mais avant d'aller plus loin, voici ce mécanisme :

- repérer les fonctions candidates ;
- les filtrer en ne gardant que les viables ;
- les filtrer de nouveau selon le critère de la meilleure fonction viable.

À la suite de ce mécanisme, s'il reste plus d'une fonction, on aboutit à une erreur de compilation du type « appel ambigu » ; si la fonction trouvée est inaccessible (comme une fonction membre privée ou protégée) ou est virtuelle pure, on aboutit également à une erreur.

Info

Les signatures des fonctions ne tiennent pas compte du type de la valeur de retour. Par conséquent, le mécanisme de surcharge ne peut pas en tenir compte non plus.

Les fonctions candidates :

- ont le même nom que la fonction appelée ;
- appartiennent toutes à la même région déclarative (la recherche commence dans le même niveau que l'appel, puis remonte au niveau supérieur jusqu'à en trouver une, puis recense toutes celles de ce même niveau).

L'exemple suivant illustre ce principe.

```
void print(double);
void afficher()
{
    void print(int);
    print(1.2);       // fonctions candidates : print(int)
}

class Reel
{
public:
    void print(double);
};

class Entier : public Reel
{
public:
    void print(int);
};
Entier e;
e.print(1.2);     // fonctions candidates :
                     ➥ Entier::print(int)
```

Attention

Une exception est faite pour les opérateurs où l'ensemble des fonctions candidates s'étend à l'union de ces trois domaines de recherche :

- **les fonctions membres candidates (si le premier opérande est un objet) ;**
- **les fonctions non membres candidates ;**
- **les opérateurs redéfinis.**

Une fonction candidate est viable si :

- elle possède autant de paramètres que l'appel (en tenant compte des valeurs par défaut) ;
- le type de chaque paramètre correspond (à une conversion implicite près).

Info

Les fonctions membres sont traitées comme des fonctions standard en leur adjoignant comme premier paramètre l'objet du type de la classe. Par exemple :

```
class MaClasse
{
    void fonction_1(int);
    void fonction_2(double) const;
};
```

sera vu comme :

```
void fonction_1(MaClasse*, int);
void fonction_2(const MaClasse*, double);
```

et l'appel :

```
monObject.fonction_1(3);
```

comme :

```
fonction_1(&monObjet,3);
```

La meilleure fonction viable est déterminée en fonction de la *qualité* des conversions utilisées pour déterminer sa viabilité. Une conversion est meilleure si (dans l'ordre) :

- elle ne nécessite aucune conversion, ou est une conversion triviale comme `Type[]` vers `Type*` (et sa réciproque), `Type` vers `const Type` (uniquement dans ce sens), `f()` vers `(*f)()` (et sa réciproque) ;
- elle est une promotion de `char` vers `int`, `short` vers `int` ou `float` vers `double` ;
- elle est une conversion standard (par exemple `int` vers `float`) ;
- elle est une conversion utilisateur.

```
void f(int, double);
void f(double, int);
f(1, 1);                // appel ambigu
f(1.0, 1.0);            // appel ambigu
```

Exemple de résolution de surcharge ambiguë

Attention

Nous avons dit que la résolution pouvait parfois être déroutante. En voici un exemple :

```
class Object
{
public:
    operator int();
};
void fonction(double, int);
void fonction(int, Objet);
Objet obj;
fonction(0.99, obj);
```

Dans ce cas, la meilleure fonction viable sera `fonction(int,Obj)` malgré la présence de l'opérateur de conversion.

Les espaces de noms

```
namespace nom
{
    // déclarations / implémentations
}
```

Les *espaces de noms* permettent de ranger du code dans des boîtes virtuelles. Cela permet par exemple de faire cohabiter

des fonctions ayant le même nom et les mêmes types de paramètres. Le même principe est applicable aux définitions de classes et aux variables.

Si vous disposez de deux (ou plus) entités (données ou méthodes) de même nom, en C standard seule la plus locale est accessible. C++ permet de préciser de quel nom il est question grâce à la syntaxe `où::nom` (`::nom` signifie que l'on désire accéder à l'espace *global*).

```
namespace perso
{
    void f();
}
void f();
```

Tout ce qui est défini dans `perso` (donc entre les accolades) est différent de ce qui est défini ailleurs. À l'extérieur de `perso`, on peut néanmoins accéder à sa fonction `f()` grâce au code `perso::f()`.

Il est possible d'utiliser une préférence d'appel grâce à la déclaration `using`. Ainsi, dans notre exemple précédent, `using perso;` permet d'accéder à toutes les composantes de `perso` (dans la portée de la déclaration, évidemment). Utilisez-la au niveau global et vous obtiendrez une préférence par défaut pour le code qui la suit. Utilisez-la dans un bloc et la préférence sera locale au bloc.

Info

Vous pouvez imbriquer les espaces de noms, mais vous ne pouvez pas en créer dans une classe ou un bloc de code (c'est-à-dire dans le corps d'une fonction).

Astuce

Utilisez les espaces de noms anonymes pour limiter la portée de vos déclarations plutôt que de les déclarer `static`. Par exemple :

```
namespace A
{
    namespace // anonyme car pas de nom
    {
        void fonction()
        {
            //…
        }
    }
    fonction(); // OK : visible
}
fonction(); // ERREUR : n'est plus visible
```

La `fonction()` est visible dans la portée du `namespace` anonyme.

Incompatibilités avec le C

Pointeurs de type void (C90 et C99)

En C, il est possible de réaliser une conversion implicite d'un type donné vers un pointeur générique de type `void*`, et *vice versa*.

```
void *ptr_void;
int *ptr_int;
// …
ptr_void = ptr_int;
ptr_int = ptr_void; // en C++ : génère un message d'erreur
```

En C++, la conversion d'un pointeur générique `void*` vers un pointeur d'un type donné doit être explicitée de la manière suivante (voir la section « Conversion avec `reinterpret_cast` ») :

```
ptr_int = reinterpret_cast<int*>(ptr_void);
```

Instruction goto (C90 et C99)

```
    int i = 0;
goto start:
    int j = 1; // ERREUR en C++
    //…
start:
    //…
```

En C++, l'instruction `goto` ne peut pas être utilisée pour sauter une déclaration comportant une initialisation, sauf si le bloc qui contient cette déclaration est entièrement sauté.

Type de caractères et surcharge (C90 et C99)

Les types utilisés pour représenter les caractères ne sont pas les mêmes en C et en C++. Le C utilise un entier (`int`) alors que le C++ utilise le type caractère (`char`). Cela peut avoir son importance si vous surchargez des fonctions (notamment de la bibliothèque C standard), comme dans cet exemple :

```
int putchar(int c); // présent dans <stdio.h>
int putchar(char c) // ma surcharge
{
    printf("%c\n", c);
}
```

Dans ce cas, le code `putchar('a');` appellera la seconde fonction et non la première.

Initialisation de tableaux de caractères

```
char tableau[5] = "12345";
```

En C, il est possible d'initialiser un tableau de caractères avec une chaîne ayant la même longueur, sans compter son zéro (`'\0'`) terminal. En C++, une telle initialisation générera un message d'erreur.

Type de retour des fonctions

```
fonction();
```

En C, omettre le type de retour d'une fonction lors de sa déclaration équivaut implicitement à un retour de type `int`. Ceci n'est pas permis en C++.

Type booléen

```
typedef int bool;
```

En C, il était possible de définir un type `bool`. En C++, ce n'est pas permis et génère un message d'erreur du type : « error: redeclaration of C++ built-in type 'bool' ».

Lier du code C et C++

```
extern "C"
```

Pour gérer la surcharge de fonctions (voir la section « Surcharge »), le compilateur génère un symbole de noms

plus long que celui de la fonction elle-même. Ce procédé, appelé *name mangling*, adjoint au nom de la fonction des informations permettant de connaître le nombre et le type de ses arguments. C'est ce nom long qu'utilise l'éditeur de liens.

Les compilateurs C ne disposent pas de ce mécanisme de signature des fonctions. Et pour cause : il n'y a pas de surcharge en C. Du coup, il est nécessaire d'encadrer toutes les fonctions C ainsi :

```
extern "C"
{
    // déclarations de variables et fonctions C
}
```

Vous pouvez également faire précéder chaque déclaration par cette même mention :

```
extern "C" void fonctionC(int);
extern "C" double autreFonction(double);
```

Astuce

N'hésitez pas dans vos fichiers d'en-têtes C à utiliser la macro `__cplusplus` qui n'est prédéfinie qu'en C++. Vous rendrez ainsi ceux-ci portables.

```
#ifdef __cpluplus
extern «C» {
#endif
    // vos déclarations C
#if __cpluplus
}
#endif
```

Embarquer une fonction

```
inline Type fonction(...) { ... }
```

Le langage C++ introduit le concept de fonctions embarquées en ajoutant le mot-clé `inline`, qui permet de définir des fonctions dont l'appel dans le programme sera remplacé par le code de la fonction elle-même. Elles doivent être réservées de préférence aux petites fonctions (membres de classe ou globales) fréquemment utilisées.

```
inline int doubler(int i)
{
   return 2 * i;
}
```

Attention

Ce mot-clé est une *indication* donnée au compilateur. Il ne garantit pas que le code sera effectivement embarqué.

Constantes usuelles

```
#include <limits>
#include <climits>
#include  <cfloat>
```

Vous trouverez toutes les constantes usuelles dans ces trois fichiers d'en-tête. L'exemple suivant montre comment connaître la valeur maximale du type `double` ; il utilise `<limits>` :

```
std::numeric_limits<double>::max();
```

3

Pointeurs et références

Un *pointeur* est simplement une variable contenant une adresse mémoire, indiquant où commence le stockage mémoire d'une donnée d'un certain type. Une *référence* est un pointeur masqué, permettant de manipuler l'objet « pointé » par un pointeur comme s'il s'agissait d'une variable ordinaire.

Les pointeurs sont couramment utilisés en C et C++.

Attention

Il est essentiel de bien comprendre les pointeurs et les références. Une erreur de manipulation est vite arrivée. Gardez en tête que la mémoire d'un ordinateur n'est qu'une suite ordonnée de 0 et de 1, groupés par blocs. Historiquement, il s'agissait de bloc de 8 bits, c'est-à-dire un octet. Ces blocs sont implicitement numérotés. Le numéro d'un bloc correspond à son *adresse*. L'architecture des ordinateurs ayant évolué, certaines données doivent débuter à des adresses multiples de 4 (sur une architecture 32 bits) ou de 8 (sur une architecture 64 bits). C'est ce que l'on appelle « l'alignement ».

Pour en savoir plus sur les problèmes d'alignement mémoire, vous pouvez vous référer à la page web http://fr.wikipedia.org/wiki/Alignement_de_données.

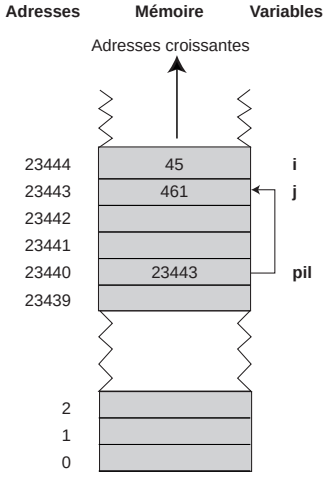


Figure 3.1 : Notions de pointeur et d'adresse

Créer et initialiser un pointeur

```
Type* pointeur;
```

Pour connaître l'adresse d'une variable, on utilise une *indirection* avec l'opérateur `&`. Lorsque l'on veut accéder au contenu d'une variable pointée, on utilise un *déréférencement* avec l'opérateur `*`.

Le code suivant illustre ces manipulations :

```
int a;    // déclaration d'une variable, ici de type entier
int *pa;  // déclaration d'un pointeur sur entier
 pa = &i; // indirection : on récupère l'adresse de a
*pa = 0;  // déréférencement : on utilise la variable
          //                   précédemment déréférencée
*pa = *pa + 1;   // encore : ici pour ajouter 1 au contenu
                 // de la variable pointée (autrement dit a)
```

Attention

Le * dans la déclaration d'un pointeur ne se rapporte qu'à la variable immédiatement à sa droite. Ainsi :

```
int *a, b;
```

déclare a comme pointeur sur un entier MAIS b comme un entier et NON un pointeur sur entier. Pour que a et b soient tous deux des pointeurs, il faut répéter le * devant chaque nom de variable (du moins celles que nous souhaitons être des pointeurs), comme suit :

```
int *a, *b;
```

Accéder aux données ou fonctions membres

```
pointeur->membre
pointeur->fonction(…)
(*pointeur)->membre
(*pointeur)->fonction(…)
```

Lorsque l'on utilise des pointeurs sur des structures ou des classes, accéder aux données ou fonctions membres se réalise de deux façons :

```
struct Copain
{
    int m_age;
};
Copain toto;
Copain *pCopain = &toto;
(*pCopain).age = 5; // 1ère façon
pCopain->age = 7;   // 2e façon, plus pratique
```

Info

Historiquement, la bibliothèque standard du langage C avait introduit la constante `NULL` (par l'intermédiaire d'une macro). La valeur de cette dernière, bien que souvent `0`, pouvait valoir n'importe quelle valeur (certains compilateurs la définissaient à `-1`). Le C++ a unifié cette valeur à `0`, facilitant ainsi le portage des programmes.

Certains encouragent l'utilisation systématique de `0`, d'autres celle de la macro `NULL` quitte à la définir soi-même à `0` si elle n'existe pas. Réjouissez-vous, la future norme C++0x (voir l'annexe qui lui est consacrée) réconciliera certainement tout le monde en introduisant le nouveau mot-clé `nullptr`.

Créer et utiliser une référence

```
Type& reference;
```

Les références, ajoutées par le C++, masquent le système d'indirection et de déréférencement des pointeurs.

L'exemple suivant montre à quel point leur utilisation est simple :

```
int I;
int &rI = I; // référence sur la variable I
rI = rI + 1; // ajoute 1 à rI et donc aussi a I
```

Déclarer un pointeur sur un tableau

```
type (*tableau)[N];
```

Ce code déclare un pointeur sur un tableau de N éléments.

Attention

`type* tableau[N];`
déclare un tableau de N pointeurs.

Pointeurs et tableaux

Différence de type

Du point de vue du compilateur, il existe une différence entre :

- `T var[]` qui définit une variable de type *tableau de type T* et ;
- `T* var` qui définit une variable de type *pointeur sur type T.*

Ainsi, si vous définissez par exemple `char a[6]` dans un fichier source, pour le rendre public et accessible par d'autres fichiers sources, vous devez le déclarer (par exemple dans un fichier d'en-tête) comme `extern char a[6]` ou `extern char a[]` et *non* `extern char *a`.

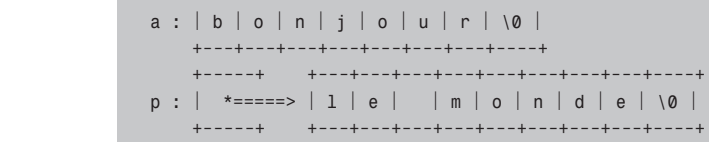
Différence mémoire

On entend souvent dire que `T a[]` et `T* a` sont équivalents. Cela est vrai lors de leur passage en tant que paramètre de fonction. Cela est faux vis-à-vis de la structure mémoire lors de la déclaration.

Un exemple valant mieux que mille mots, considérons les deux déclarations suivantes :

```
char  a[] = "bonjour";
char *p = "le monde";
```

Les données correspondantes en mémoire peuvent être représentées ainsi :



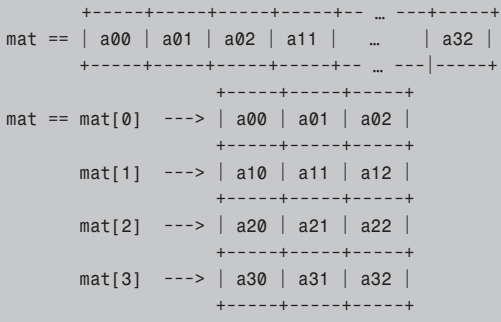
Il est important de réaliser cette différence pour comprendre que le code généré par la suite peut influer sur les performances. En effet, dans le deuxième cas une opération sur l'arithmétique des pointeurs se cache derrière l'instruction `p[3]`.

Équivalence lors de l'accès

Même s'il existe une différence subtile entre tableau C et pointeur, tout accès à un élément de tableau a un équivalent avec un appel par pointeur.

Par exemple `int t[5];` réserve 5 entiers consécutifs en mémoire, et `t` correspond à un pointeur sur le début de cette mémoire. Ainsi, si `int* pi = t;` alors `*(pi+3)` ou `pi[3]` est équivalent à `t[3]`.

Pour les tableaux multidimensionnels, les choses sont un peu plus complexes. Tout d'abord, il faut comprendre comment un tableau multidimensionnel est organisé en mémoire. Prenons le cas d'un tableau à deux dimensions. Par exemple `T mat[4][3]` peut être représenté ainsi :



Le tableau d'éléments est stocké en mémoire ligne après ligne ; pour un tableau de taille $m \times n$ d'éléments de type `T`, l'adresse de l'élément *(i,j)* peut s'obtenir ainsi :

```
adresse(mat[i][j]) == adresse(mat[0][0]) + (i*n +j) * size(T)
adresse(mat[i][j]) == adresse(mat[0][0]) +
                      i * n * size(T) +
                      j * size(T)
adresse(mat[i][j]) == adresse(mat[0][0]) +
                      i * size(une ligne de T) +
                      j * size(T)
```

D'une manière générale si on a `T tableau[D0][D1][D2]…[DN]`, on peut accéder à l'élément `tableau[i0][i1][i2]…[iN]` par la formule suivante :

```
*(tableau + i0 + D1* (i1 + (D2* (i2 + D3* (… + DN*iN))))
```

Cette formule omet la taille d'un élément car le compilateur la déduit automatiquement en fonction du type du pointeur. Du coup, prenez garde au fait que si le type de pointeur est différent, l'adresse obtenue n'est pas la même. Si l'on a :

```
int *pi = 0;
char* pc = 0;
```

alors `(pi + 1) != (pc + 1)`.

Équivalence en tant que paramètre

Lors du passage d'un tableau à une fonction, seul le pointeur sur le début du tableau est transmis. Du coup, on a une équivalence stricte entre les deux représentations. Par contre, s'il s'agit d'un tableau multidimensionnel, on a tout intérêt à garder la déclaration sous forme de tableau pour éviter de se tromper dans le calcul de la conversion des indices.

Déclarer un pointeur sur une fonction

```
Type (*pointeur_sur_fonction)(paramètres);
```

```
typedef Type (*type_pointeur_sur_
fonction)(paramètres);
type_pointeur_sur_fonction pointeur_sur_fonction;
```

Les parenthèses autour de `*pointeur_sur_fonction` sont obligatoires, sinon le compilateur pensera qu'il s'agit de la déclaration d'une fonction renvoyant un pointeur sur le `Type` donné. L'exemple suivant montre comment utiliser les pointeurs sur fonction et donne un aperçu de leur intérêt. Cet exemple est écrit en C, mais fonctionne parfaitement en C++.

```
#include <stdio.h>
#include <string.h>

#define MAX_BUF 256

long arr[10] = { 3,6,1,2,3,8,4,1,7,2};
char arr2[5][20] = {  "Mickey Mouse",
                      "Donald Duck",
                      "Minnie Mouse",
                      "Goofy",
                      "Ted Jensen" };

// Déclaration du type de la fonction de comparaison
// utilisé par le tri bulle
typedef int (*FnComparaison)(const void *, const void *);
// Fonction de tri à bulle générique
void tri_bulle(void *p, int width, int N, FnComparaison fptr);
```

```
// Deux fonctions de comparaison
int compare_chaine_c(const void *m, const void *n);
int compare_long(const void *m, const void *n);

int main(void)
{
    int i;
    puts("\nAvant le tri :\n");

    for (i = 0; i < 10; i++)   // Affiche les ints de arr
        printf("%ld ",arr[i]);
    puts("\n");

    for (i = 0; i < 5; i++)    // Affiche les chaînes de arr2
        printf("%s\n", arr2[i]);

    tri_bulle(arr, 4, 10, compare_long);   // Trie les longs
    tri_bulle(arr2, 20,5, compare_chaine_c); // Trie les chaînes
    puts("\n\nAprès le tri :\n");

    for (i = 0; i < 10; i++)   // Affiche les longs triés
        printf("%d ",arr[i]);
    puts("\n");

    for (i = 0; i < 5; i++)    // Affiche les chaînes triées
        printf("%s\n", arr2[i]);
     return 0;
}

// Implémentation de la fonction de tri à bulle générique
void tri_bulle(void *p, int width, int N,
               FnComparaison fptr)
{
    int i, j, k;
    unsigned char buf[MAX_BUF];
    unsigned char *bp = (unsigned char*)p;

    for (i = N-1; i >= 0; i--)
```

```
    {
        for (j = 1; j <= i; j++)
        {
            k = fptr((void *)(bp + width*(j-1)),
                     (void *)(bp + j*width));
            if (k > 0)
            {
                memcpy(buf, bp + width*(j-1), width);
                memcpy(bp+width*(j-1), bp+j*width, width);
                memcpy(bp+j*width, buf, width);
            }
        }
    }
}

int compare_chaine_c(const void *m, const void *n)
{
    char *m1 = (char *)m;
    char *n1 = (char *)n;
    return (strcmp(m1,n1));
}

int compare_long(const void *m, const void *n)
{
    long *m1, *n1;
    m1 = (long *)m;
    n1 = (long *)n;
    return (*m1 > *n1);
}
```

Dans cet exemple, la fonction `tri_bulle` est généralisée en passant en paramètre la méthode permettant de comparer les valeurs contenues dans le tableau. Avec `compare_long`, on considère que le tableau contient des longs ; avec `compare_chaine_c` on considère que le tableau contient des pointeurs sur des chaines de type C. Dans tous les cas, il est nécessaire que la taille d'un élément du tableau soit égale à la taille d'un pointeur. Il est impossible d'utiliser ce `tri_bulle` avec un tableau de caracères ASCII.

Passer un objet en paramètre par pointeur/référence

```
Type1 fonction(…, Type2* ptr, …); // par pointeur
```

```
Type1 fonction(…, Type2& ref, …); // par référence
```

Passer un paramètre par pointeur ou par référence évite de copier des objets complexes lors de leur passage à des fonctions. Cela permet d'économiser de la mémoire et du temps.

Attention

Il faut absolument éviter de recopier un objet « lourd » (du fait de sa taille ou du temps que prendrait sa recopie) si cela n'est pas nécessaire. Il faut être conscient que cela se fait de manière implicite lorsqu'on le transmet à une fonction (passage de paramètre).

Toutefois, il peut exister un intérêt à créer une copie temporaire. On peut ainsi la modifier à souhait, sans toucher à l'original.

L'exemple 1 passe un paramètre par *recopie*.

```
MaClasse A;
// …
void fonction(MaClasse B)
{
    // B est une copie de A
    B.methode();
   // B est détruit à la fin de la fonction
}
// …
fonction(A);
```

L'exemple 2 utilise un paramètre par *référence*. On ne recopie plus l'objet, par contre une modification de `B` entraîne une modification de `A`. Il est possible de déclarer un paramètre constant (à l'aide du mot-clé `const`) ; dans ce cas, tout appel à une fonction membre non `const` ou toute tentative de modification d'une variable membre de l'objet (à condition qu'elle ne soit pas qualifiée de `mutable`) aboutira à une erreur de compilation.

```
MaClasse A;
// …
void fonction(MaClasse& B)
{
    B.methode();
}
void fonction(const MaClasse& B)
{
    B.methode(); // erreur si MaClasse::methode non const
    B.var = 0;   // erreur
    Type v = B.var; // ok
}
```

Enfin, l'exemple 3 illustre un passage de paramètre par pointeur. Toute modification de `B` entraîne une modification de `A`.

```
MaClasse* A = new MaClasse();
void fonction(MaClasse* B)
{
    B->methode();
}
```

Info

La signature de ces trois fonctions est identique :

```
void f(int a[10]);
void f(int* a);
void f(int a[]);
```

4

Classes et objets

La programmation orientée objet (POO), ou programmation par objet, est un paradigme de programmation informatique qui consiste en la définition et l'assemblage de briques logicielles appelées « objets » ; un objet représente un concept, une idée ou toute entité du monde physique : voiture, personne ou encore page d'un livre. La programmation orientée objet utilise des techniques comme l'*encapsulation*, la *modularité*, le *polymorphisme* et l'*héritage*. Ce chapitre montre comment le langage C++ les met en œuvre.

Attention

Ne perdez jamais de vue que le C++ est un langage *orienté* objet. Beaucoup semblent l'oublier lorsqu'ils découvrent les possibilités orientées objet du C++. Ils s'emballent et, sous prétexte de faire de l'objet, définissent des méthodes et encore des méthodes... C'est plus un défaut qu'une bonne pratique. Il est facile d'inclure dans ses classes des choses qui n'ont rien à y faire. Pour vous aider, posez-vous cette question : la fonctionnalité que j'écris fait-elle partie de l'objet ou bien agit-elle dessus ? En d'autres termes : ne négligez pas l'utilité et le bien-fondé des fonctions.

Ajouter des données à des objets

```
class MaClasse
{
    type donnee;
};
```

Ajouter une donnée à un objet se fait de la même manière qu'en C avec les structures `struct`.

L'exemple suivant permet de définir l'âge d'une personne en ajoutant la donnée membre `m_age` à l'objet `Personne`.

```
class Personne
{
    int m_age;
};
```

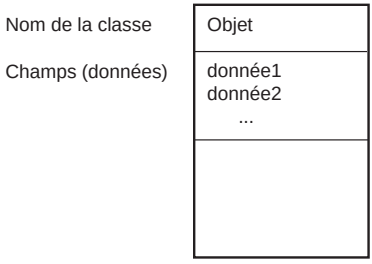


Figure 4.1 : Représentation UML des données d'un objet

On distingue deux types de données :

- Celles que l'objet possède. C'est la *composition*. Dans ce cas, la donnée naît et meurt avec l'objet.

```
struct Voiture
{
    Carburateur m_carbu;
};
```

Exemple de composition (trivial)

```
class Voiture
{
    Carburateur* m_carbu;
public:
    Voiture() { m_carbu = new Carburateur; }
    ~Voiture() { delete m_carbu; }
};
```

Exemple de composition (avec pointeur)

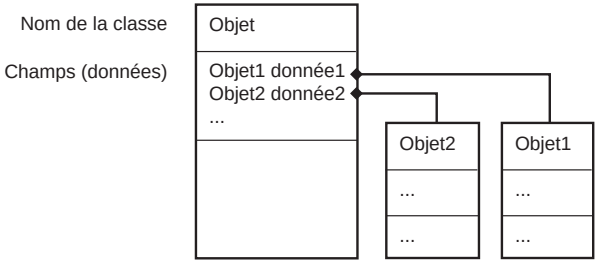


Figure 4.2 : Représentation UML d'une composition

- Celles qui sont des liens vers un autre objet. C'est l'*agrégation*. Dans ce cas, la durée de vie de l'objet et de la donnée liée sont indépendantes. Dans l'exemple suivant, la durée de vie du carburateur ne dépend pas de celui d'une voiture.

```
{
    Carburateur* m_carbu;
public:
    Voiture() { m_carbu = 0; }
    ~Voiture() {}
    void set_carburateur(Carburateur* c) { m_carbu = c; }
};
```

Exemple d'agrégation

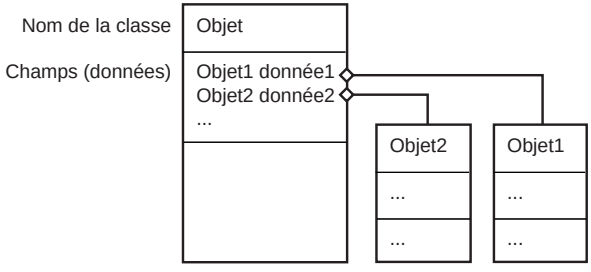


Figure 4.3 : Représentation UML d'une agrégation

Lier des fonctions à des objets

```
class MaClasse
{
    type fonction_membre(arguments);
};
```

Pour lier une fonction à un objet, ou plus exactement, à une instance d'objet, vous devez utiliser une *fonction membre*. Vous pouvez l'appeler avec l'opérateur . (point) ou -> (flèche) comme indiqué ci-après, selon qu'il s'agit d'une instance de l'objet considéré ou d'un pointeur sur celle-ci.

```
Type instance, *instance_ptr;
instance.fonction_membre(arguments);
instance_ptr->fonction_membre(arguments);
```

Attention

La validité du pointeur sur une instance d'une classe est sous la responsabilité du programmeur. Un appel de fonction membre avec un pointeur invalide peut provoquer un comportement inattendu, mais pas forcément un plantage du programme. Un appel avec un pointeur nul provoque le plus souvent un plantage direct et plus facile à détecter.

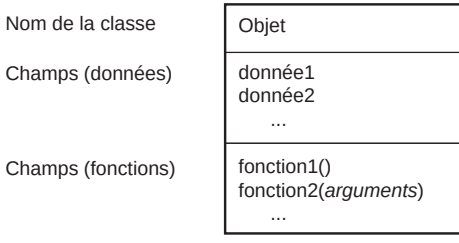


Figure 4.4 : Représentation UML des fonctions d'un objet

L'exemple ci-après crée une simple classe représentant une personne avec son âge et son nom (reportez-vous au Chapitre 10 pour en savoir plus sur le type chaîne de caractères utilisé pour cette variable).

```
class Personne
{
    std::string m_prenom, m_nom ;
    int m_age ;
public :
    //…
    int setNom(const std::string& n) ;
    //…
    int getAge() const { return m_age ; }
    void setAge(int a) const { m_age = a ; }
    //…
    void affiche(std::ostream& o) const
    {
        o << " Nom    : "  << m_nom << std::endl
          << " Prenom : "  << m_prenom << std::endl
          << " Age    : "  << m_age << std::endl ;
    }
} ;
// dans le fichier source
void Personne::setNom(const std::string& n)
{
    m_nom = n ;
}
```

Voici maintenant un exemple d'utilisation de cette classe :

```
Personne moi ;
moi.setAge( jeux_devinez_le() ) ;
moi.affiche( std::cout ) ;
```

Déterminer la visibilité de fonctions ou de données membres

```
class Classe
{
public :
    // …
protected :
    // …
private :
    // …
} ;
```

Les fonctions et données membres `public` sont visibles (ou accessibles) par n'importe quelle partie du programme utilisant la classe. Elles fournissent l'interface publique de la classe.

Les fonctions et données membres `private` ne sont visibles que par les fonctions membres de la classe. Elles sont utilisées pour cacher les détails d'implémentation.

Les fonctions et données membres `protected` sont visibles par les fonctions membres de la classe ou de ses classes filles directes (par héritage). Elles ne sont pas visibles du reste du programme.

Info

Les `class` sont `private` par défaut (tant pour l'héritage que pour les membres précédant la première spécification de visibilité), alors que les `struct` sont `public` par défaut.

```
class MaClasse : /* private */ ClasseMere { … } ;
struct MaClasse : /* public */ ClasseMere { … } ;
```

Il est possible de représenter sous forme d'une *hiérarchie de classes*, parfois appelée *arborescence de classes*, la relation de parenté qui existe entre les différentes classes. L'arborescence commence par une classe générale appelée *superclasse* (parfois *classe de base*, *classe parent*, *classe ancêtre*, *classe mère* ou *classe père*, les métaphores généalogiques sont nombreuses). Puis les classes *dérivées* (*classe fille* ou *sous-classe*) deviennent de plus en plus spécialisées. Ainsi, on peut généralement exprimer la relation qui lie une classe fille à sa mère par la phrase « est un » (de l'anglais « *is a* »).

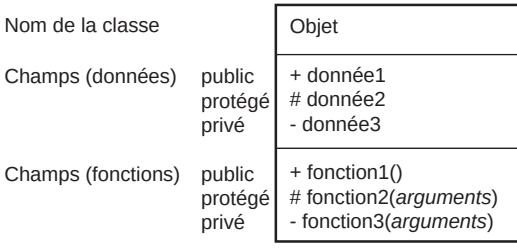


Figure 4.5 : Représentation UML de la visibilité des membres

Astuce

Grâce au mot-clé `friend` (ami), il est possible de rendre accessible les données et fonctions privées d'une classe à :

- une *fonction particulière*, définie n'importe où dans le programme ;
- une *classe extérieure* déterminée ;
- une *fonction particulière d'une classe extérieure* déterminée.

Le mot-clé `friend` est par exemple nécessaire lorsque l'on veut implémenter des opérateurs globaux qui utilisent des éléments de ladite classe.

Pour donner l'accès à une fonction, vous pouvez procéder comme suit :

```
class X
{
    int a;
    friend void f ( X* );
};

void f (X * p)
{
    p->a = 0;
}

void main ()
{
    X * ptr = new X;
    f (ptr);
}
```

Pour donner l'accès à une classe, vous pouvez procéder comme suit :

```
class B;

class A
{
private:
    int a;
    f();
    friend B;
};
```

```
class B
{
    void h (A*p) { p->a = 0; p->f(); }
};
```

Pour donner l'accès à une fonction membre d'une autre classe, vous pouvez procéder comme suit :

```
class B;

class A
{
private:
    int a;
    friend void B::f();
};

class B
{
    void f(A * p) { p->a = o; }
};
```

Expliciter une instance avec le pointeur `this`

```
type MaClasse::fonction_membre(arguments)
{
    …
    this->variable_membre = … ;
    …
    return valeur;
}
```

Le pointeur `this` correspond à l'adresse mémoire de l'instance ayant servi pour l'appel de la fonction membre. Son utilisation n'est pas obligatoire, sauf pour lever une ambiguïté (par exemple lorsqu'une variable membre de la

classe porte le même nom qu'un argument de la fonction membre).

Le pointeur `this` n'est disponible que dans l'implémentation d'une fonction membre.

Astuce

L'utilisation systématique de `this` pour accéder à des membres de la classe se révèle être une bonne pratique à l'usage. Elle rend évidente l'accès à ces derniers ; la lourdeur apparente de cette syntaxe a pour contrepartie une compréhension accrue, facilitée, du code.

Définir un constructeur/destructeur

```
class Objet
{
public:
    Objet();              // constructeur par défaut
    Objet(const Objet&); // constructeur par copie
    Objet(paramètres);    // constructeur
    ~Objet();             // destructeur
};
```

Les *constructeurs* et le *destructeur* d'un objet ressemblent, à quelques détails près, à des fonctions membres :

- ils n'ont pas de type de retour ;
- ils portent le même nom que l'objet.

Le destructeur possède en plus les caractéristiques suivantes :

- son nom commence par un ~ (tilde) ;
- il n'a pas d'argument.

Pour toute instanciation de classe, un constructeur est obligatoirement appelé. Si aucun constructeur n'existe, le compilateur en génère un automatiquement. Ce dernier consiste à appeler le créateur par défaut pour chacun des membres de la classe, lorsqu'il existe ou qu'il peut être généré.

Attention

L'héritage est censé garantir l'ordre d'appel des destructeurs. Par exemple, si `C` hérite de `B` qui hérite elle-même de `A`, lors de la destruction de d'une instance de `c` les destructeurs doivent être appelés dans l'ordre inverse, soit `:C::~C`, `B::~B` puis `A::~A`. Pourtant, ce n'est pas toujours le cas, notamment avec certains vieux compilateurs. Prenez donc l'habitude d'être prudent et partez du postulat que ce n'est généralement pas le cas.

Ajoutons maintenant un constructeur et un destructeur à notre classe `Personne`.

```
class Personne
{
    //…
public:
    Personne(std::string prenom, std::string nom, int age=0)
      : m_prenom(prenom), m_nom(nom), m_age(age)
    {
        std::cout << m_prenom << " " << m_nom
                  << " vient de naître\n";
    }
    ~Personne()
    {
        std::cout << m_prenom << " " << m_nom
                  << " vient de mourir\n";
    }
    //…
};
```

Et utilisons-le :

```
//…
{
    Personne reMoi("Vincent", "Gouvernelle");
    //…
}
//…
```

Lors de l'exécution du code précédent, nous obtiendrons cet affichage sur la console :

```
Vincent Gouvernelle vient de naître
Vincent Gouvernelle vient de mourir
```

La première ligne apparaît lors de l'instanciation de la classe. La deuxième apparaît lors de la destruction de celle-ci à la fin du bloc.

Empêcher le compilateur de convertir une donnée en une autre

```
class Objet
{
public:
    explicit Objet(paramètres);   // constructeur
};
```

La surcharge de fonction implique de manière sous-jacente des tentatives de conversion des paramètres. Cela peut être indésirable, voire dangereux. Pour interdire au

compilateur de tenter de convertir une donnée en une autre par l'intermédiaire d'un constructeur, vous avez la possibilité de rendre ce dernier `explicit`.

```
struct Age
{
    int valeur;
    Age(int a) : valeur(a) {}
};
void fonction(Age a);
//…
fonction(34); // OK : 34 implicitement converti en
                ➥ Age(34)
```

```
struct Age
{
    int valeur;
    explicit Age(int a) : valeur(a) {}
};
void fonction(Age a);

//…
fonction(34);      // Erreur
fonction(Age(34)); // OK : l'utilisateur rend
                     ➥"explicite" sa volonté
```

Astuce

Le mot-clé `explicit` permet également de simuler avec le C++, dans une certaine mesure, un langage à typage fort comme Pascal ou Ada.

Spécifier qu'une fonction membre ne modifie pas l'objet lié

```
class Objet
{
    // …
    type fonction(arguments) const;
    // …
    mutable type variable;
};
```

Une fonction spécifiée comme `const` ne peut ni appeler d'autres fonctions membres non `const`, ni modifier ses variables membres (sauf si elles sont `mutable`). Mentionnez dès que possible ce spécificateur, car un objet passé en référence constante (`const type &`) ne peut appeler que des fonctions membres `const`.

Info

N'abusez pas du spécificateur `mutable` pour détourner `const`. D'une manière générale, cela traduit un défaut de conception de votre architecture logicielle.

Rendre une fonction/donnée membre indépendante de l'objet lié

```
class  Objet
{
    static type variable;
public:
    static type fonction(arguments);
};
```

Les membres `static` d'une classe ne sont pas liés à une instance de cette dernière. Ils agissent comme des variables globales ou des fonctions traditionnelles, mais sont conceptuellement liés à la classe.

Pour illustrer ceci, examinons rapidement une implémentation simple du *pattern* Singleton. L'objet du singleton est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement). Il est utilisé par exemple lorsque l'on a besoin d'exactement un objet pour coordonner des opérations dans un système.

```
class Singleton
{
    static Singleton* unique_instance;
    int status;
    Singleton();
public:
    static Singleton* get_instance() const;
    // Autres déclarations
    int get_status() const;
};
```

Pattern Singleton (déclaration)

```
Singleton* Singleton::unique_instance = 0;
static Singleton* Singleton::get_instance() const
{
    if (unique_instance == 0)
        unique_instance = new Singleton;
    return unique_instance;
}
int Singleton::get_status() const
{
    return status;
}
```

Pattern Singleton (implémentation)

```
int s = Singleton::get_instance()->get_status();
```

Pattern Singleton (utilisation)

Comprendre le changement de visibilité lors de l'héritage

```
class Fille : public Parente { … };
class Fille : private Parente { … };
class Fille : protected Parente { … };
```

Info

L'héritage est une notion essentielle dans la programmation objet. L'héritage consiste à réunir les définitions d'une (héritage simple) ou plusieurs classes (héritage multiple) et de leur adjoindre un ensemble de membres spécifiques. Cette nouvelle classe est communément appelée « classe dérivée ».

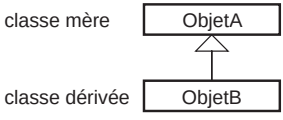


Figure 4.6 : Représentation UML de l'héritage

La dérivation peut être de trois types : `public`, `protected` (protégé) ou `private` (privé). Selon le type de dérivation, la visibilité des membres de la classe parente change. Le tableau suivant résume les différents cas.

Visibilité des membres parents en fonction du type de dérivation

Classe parente	Héritage privé	Héritage protégé	Héritage public
inaccessible	inaccessible	inaccessible	inaccessible
privée	inaccessible	inaccessible	inaccessible
protégée	privée	protégé	protégé
publique	privée	protégé	public

Pour bien comprendre cela, considérons la classe mère suivante :

```
class A
{
public:
    void func_publique() {}
protected:
    void func_protegee() {}
private:
    void func_privee() {}
};
```

L'héritage public engendre les visibilités suivantes :

```
class B : public A
{
public:
    void test_call()
    {
        A::func_publique(); // vue comme publique
        A::func_protegee(); // vue comme protégée
        //A::func_privee(); // erreur : est inaccessible
    }
};

void B_test_use()
{
    B b;
    b.func_publique();   // vue comme publique
    //b.func_protegee(); // vue comme protégée
    //b.func_privee();   // est inaccessible
}
```

L'héritage protégé engendre les visibilités suivantes :

```
class C : protected A
{
public:
    void test_call()
    {
        A::func_publique(); // vue comme protégée
        A::func_protegee(); // vue comme protégée
        //A::func_privee(); // est inaccessible
    }
};

void C_test_use()
{
    C c;
    //c.func_publique(); // vue comme protégée
    //c.func_protegee(); // vue comme protégée
    //c.func_privee();   // est inaccessible
}
```

L'héritage privé engendre les visibilités suivantes :

```
class D : private A
{
public:
    void test_call()
    {
        A::func_publique(); // vue comme privée
        A::func_protegee(); // vue comme privée
        //A::func_privee(); // est inaccessible
    }
};

void D_test_use()
{
    D d;
    //d.func_publique(); // vue comme privée
    //d.func_protegee(); // vue comme privée
    //d.func_privee();   // est inaccessible
}
```

Étendons maintenant notre class `Personne` pour en faire un employé. Un employé reçoit un salaire et peut pointer le matin et le soir.

```
class Employe : public Personne
{
    double m_salaire;
public:
    Employe() : Personne("",""), m_salaire(0) {}
    void pointer_matin(Heure) { … }
    void pointer_soir(Heure) { … }
};
```

Comprendre les subtilités de l'héritage multiple

```
class Fille : derivation Parent1, derivation
➥ Parent2, … { … };
```

Comme nous l'avons dit à la section « Comprendre le changement de visibilité lors de l'héritage », l'héritage peut être multiple. Il en résulte certaines subtilités qu'il est essentiel de connaître pour maîtriser cette technique. Pour utiliser sainement l'héritage multiple, il vaut mieux le voir comme un moyen d'ajouter à une classe fille un ensemble de méthodes et d'attributs de telle sorte que cela ne remette pas en cause ce dont elle héritait déjà par ailleurs.

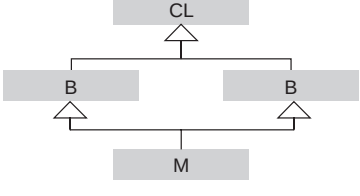


Figure 4.7 : Représentation UML de l'héritage multiple.

Le premier problème de l'héritage multiple est le clonage de donnée, comme le montre l'exemple suivant.

```
class CL { … };
class A : public CL { … };
class B : public CL { … };
class M : public A, public B
{
    …
    M(…) : A(…), B(…) … { … }
    …
};
```

Dans ce cas, la classe `CL` est dupliquée en mémoire. Si l'on veut accéder à l'une ou l'autre des données d'une instance de `CL`, il est alors nécessaire de préciser par quelle branche d'héritage il faut passer. Cela ce fait grâce à un transtypage multiple. Dans le cadre de notre exemple, on a les deux possibilités suivantes :

- `CL* clPtr = (CL*) (A*) mPtr ;`
- `CL* clPtr = (CL*) (B*) mPtr.`

Empêcher la duplication de données avec l'héritage virtuel

```
class Fille : type_de_derivation virtual Parent1,
➥… { … };
```

Dans certains cas, l'héritage multiple engendre la duplication de classes en mémoire. C'est le cas lorsqu'en remontant dans l'héritage, un ou plusieurs types de classe apparaissent plusieurs fois. Le mot-clé `virtual` doit être répété si nécessaire, comme pour le type de dérivation. L'exemple ci-après montre comment empêcher cette duplication mémoire, grâce à l'*héritage virtuel*.

```
class U { … };
class A : public virtual U { … };
class B : public virtual U { … };
class M : public A, public B
{
    …
    M(…) : U(…), A(…), B(…) … { … }
    …
};
```

Comme nous venons de le dire, le principe de l'héritage virtuel est d'éviter la duplication des données d'une classe parente apparaissant plusieurs fois dans la hiérarchie de l'héritage. Cela est pratique mais implique de devoir préciser la construction de toutes les classes dont on hérite virtuellement, et ce à chaque nouvel héritage.

C'est pourquoi, bien souvent, les programmeurs C++ se limitent à l'héritage simple autant que possible. Ils n'utilisent l'héritage multiple qu'avec parcimonie, en prenant soin d'éviter les cas où une classe apparaît plusieurs fois dans une hiérarchie. L'héritage virtuel n'est en effet que très peu utilisé, car dans une hiérarchie complexe, il devient vite très complexe.

Une méthode virtuelle de la classe commune `U` peut être redéfinie dans `A` ou `B`. Trois cas se présentent alors :

- si elle n'est redéfinie ni dans `A` ni dans `B`, alors celle de `U` sera utilisée ;
- si elle est redéfinie seulement dans `A` ou `B`, alors cette dernière sera utilisée et pas celle de `U` ;
- si elle est redéfinie dans `A` et dans `B`, alors le compilateur ne peut choisir.

Bien entendu, si elle est redéfinie aussi dans `M`, la redéfinition de `M` aura priorité sur les autres.

Simuler un constructeur virtuel

```
class Objet
{
public:
    virtual ~Objet() {}
    virtual Objet* clone() = 0;
    virtual Objet* create() = 0;
```

```
};
class Type : public Objet
{
public:
    virtual Type* clone() { return new Type(*this);
}
    virtual Type* create() { return new Type(); }
};
```

Le langage C++ ne supporte pas directement les constructeurs virtuels. Il est néanmoins possible de les simuler par un idiome. Il est ainsi possible d'obtenir le même effet qu'un constructeur virtuel grâce à l'utilisation d'une fonction membre virtuelle `clone()` pour le constructeur par copie, ou d'une fonction membre virtuelle `create()` pour le constructeur par défaut. Vous pouvez bien sûr les appeler comme bon vous semble, mais les noms employés ici sont ceux que l'on retrouve traditionnellement dans une telle situation.

Le principe est simple. La fonction membre `clone()` appelle le constructeur par copie, ou tout autre code plus complexe, dans le but de copier l'état de l'instance actuelle dans un nouvel objet de même type : c'est un clone. La fonction membre `create()` appelle simplement le constructeur par défaut de la classe concernée : elle crée une nouvelle instance neuve de même type.

```
void fonction(Objet& objet)
{
    Objet* obj1 = objet.clone();
    Objet* obj2 = objet.create();
    // …
    delete obj1; // vous comprenez ici la nécessité
                 // du destructeur virtuel
    delete obj2;
}
```

Le code ci-avant fonctionne quel que soit le sous-type d'objet utilisé. Il permet de créer une copie ou une nouvelle instance de même type que l'objet fourni en paramètre sans le connaître à l'avance.

Info

Le type de retour des fonctions membres `clone()` ou `create()` est intentionnellement différent du type de retour de la classe de base. Ce mécanisme s'appelle *covariant return type* ou, en français, « type de retour covariant ». Il n'est pas supporté par tous les compilateurs, surtout s'ils ne sont pas récents. Si c'est le cas du vôtre, vous n'aurez pas d'autres moyens que de conserver le type de retour de la classe de base, `Objet*` dans notre cas.

Créer un type abstrait à l'aide du polymorphisme

```
class Classe
{
    virtual Type Nom(arguments) = 0;
    virtual Type Nom(arguments) const = 0;
};
```

Une classe devient *abstraite* à partir du moment où elle contient au moins une *fonction virtuelle pure*. Il devient dans ce cas impossible de l'instancier. On appelle parfois les classes abstraites des ADT (*abstract data types*) ou, en français, « types de données abstraits ». L'idée est d'imposer aux concepteurs des classes dérivées de redéfinir et d'implémenter ces fonctions. Il est ainsi possible d'utiliser ces fonctionnalités avant même leur implémentation.

```
class Forme
{
public:
   virtual float Aire() = 0;
};

class Carre : public Forme
{
public:
   virtual float Aire() { return m_cote * m_cote; }
private:
   float m_cote;
};

class Cercle : public Forme
{
public:
   virtual float Aire() { return 3.1415926535*
   ➥m_rayon*m_rayon; }
private:
   float m_rayon;
};
```

Grâce aux fonctions virtuelles, on peut créer un algorithme en n'utilisant que la classe de base qui va automatiquement appeler les fonctions des classes dérivées.

En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, le *polymorphisme* permet une programmation beaucoup plus générique. Le développeur n'a pas à savoir, lorsqu'il programme une méthode, le type précis de l'objet sur lequel la méthode va s'appliquer. Il lui suffit de savoir que cet objet implémentera la méthode.

Ainsi, un logiciel de calcul d'intérêt pour des comptes bancaires se présenterait de la façon suivante (en pseudo-code) dans le cadre d'une programmation classique :

```
si type de <MonCompteBancaire> est un:
    PEA     => MonCompteBancaire->calculeInteretPEA()
    PEL     => MonCompteBancaire->calculeInteretPEL()
    LivretA => MonCompteBancaire->calculeInteretLivretA()
fin du choix
```

Si un nouveau type de compte bancaire PERP apparaît (et avec lui un nouveau calcul), il sera nécessaire d'une part d'écrire la nouvelle méthode `calculeInteretPERP()`, mais aussi de modifier tous les appels du calcul donné ci-dessus. Dans le meilleur des cas, celui-ci sera isolé et mutualisé de sorte qu'une seule modification sera nécessaire. Dans le pire des cas, il peut y avoir des centaines d'appels à modifier.

Avec le polymorphisme, toutes les méthodes porteront le même nom, par exemple `calculeInteret()`, mais auront des implémentations différentes : une par type de compte. L'appel sera de la forme :

```
MonCompteBancaire->calculeInteret()
```

Lors de l'arrivée du nouveau compte, aucune modification de ce code ne sera nécessaire. Le choix de la méthode réelle à utiliser sera fait automatiquement à l'exécution par le langage, alors que dans le cas précédent c'est le développeur qui devait programmer ce choix.

Info

Lorsqu'une classe ne contient que des fonctions membres virtuelles pures, on parle alors de classe interface. Dans certains langages, comme Java, cette notion est directement définie.

Utiliser l'encapsulation pour sécuriser un objet

```
class MaClasse
{
    // …
private: // ou protected:
    Type mon_objet_encapsule;
    Type* mon_objet_encapsule2;
    // …
public:
    Type2 methode(…);
};
```

L'*encapsulation* consiste à masquer le contenu d'un objet et à ne mettre à disposition que des méthodes permettant de manipuler cet objet. En forçant l'utilisateur de la classe à utiliser des fonctions membres (plutôt que directement les données), l'encapsulation permet d'assurer la cohésion interne d'un objet.

L'objet peut être ainsi vu comme une boîte noire, à laquelle sont associés des propriétés et/ou des comportements. L'implémentation est cachée et peut être changée sans impact sur le reste du code, à condition bien sûr que le changement d'implémentation ne change pas le comportement de l'objet. L'encapsulation permet donc de séparer la spécification, ou définition, d'un objet de son implémentation.

En troisième lieu, l'encapsulation peut permettre de cacher totalement la manière dont un objet est implémenté. Cela peut s'avérer particulièrement utile si vous voulez créer une bibliothèque sans laisser filtrer vos secrets de fabrication.

L'exemple suivant vous montre comment :

```
// Fichier en-tête
class ClassePrivee;
class ClasseVendue
{
    ClassPrivee *implementation;
public:
    Type methode(…);
};

// Fichier source
class ClassePrivee
{
    // données
public:
    Type methode(…);
};
Type ClasseVendue::methode(…)
{
    implementation->methode(…);
}
```

Obtenir des informations de types dynamiquement

```
#include <typeinfo>
class type_info
{
    …
public:
    virtual ~type_info();
    bool operator==(const type_info&) const;
    bool operator!=(const type_info&) const;
    int before(const type_info&) const;
    const char* name() const;
};
class bad_cast : public exception;
class bad_typeid : public exception;
```

```
type_info typeid(…);
```

Nous avons déjà parlé de `dynamic_cast<>` dans la section sur les conversions. Le C++ définit également la fonction `typeid()`. Elle permet d'obtenir – en temps constant, c'est-à-dire indépendamment de la taille ou de la complexité de l'objet –, des informations sur une classe ou une instance de celle-ci grâce aux *informations dynamiques de type* (de l'anglais *runtime type information,* RTTI). Cette classe diffère souvent selon les compilateurs, mais fort heureusement une trame commune se retrouve dans ces diverses implémentations.

Tout d'abord, il faut savoir qu'il n'est pas possible de créer soi-même des instances de classe `type_info`. La seule manière est de passer par la fonction `typeid()`, disponible à travers les fichiers d'en-tête `typeinfo`. Il est également impossible de copier ou d'affecter des objets de type `type_info`.

Les opérateurs `==` et `!=` permettent de tester l'égalité et l'inégalité de type à travers les objets `type_info`. Contrairement à un `dynamic_cast`, qui effectue un test du type « est compatible », ces opérateurs effectuent bien un test du type « est un ». Cela n'a pas la même vocation mais est très utile dans certains cas.

Il n'y a aucun lien entre l'ordre de collecte des types et les relations d'héritage. Vous pouvez utiliser la fonction membre `before()` pour déterminer l'ordre de collecte des types. Il n'y a aucune garantie que cette fonction retourne le même résultat selon le programme, voire même selon l'exécution d'un même programme. Sous ce rapport, cette fonction a le même genre de comportement que l'opérateur `&`.

La fonction membre `name()` renvoie le nom du type de l'objet. Parfois, ce nom est partiellement décoré ; cela dépend principalement du compilateur utilisé. Je vous déconseille donc fortement d'utiliser cette valeur comme appui dans vos programmes.

Info

Le type de retour de la fonction membre `before()` peut varier selon les compilateurs. Par exemple, Visual Studio 2008 retourne un `int`, alors que g++ 3.4.2 renvoie un `bool`.

Le fichier d'en-tête `typeinfo` fournit également la définition de deux exceptions `bad_cast` et `bad_typeid`. La première peut être levée lors d'un `dynamic_cast` invalide. La deuxième le sera lors d'un appel à `typeid()` sur une expression invalide, telle un pointeur nul.

Transformer un objet en fonction

```
struct
{
    type operator()(arguments) const;
};
```

Les *foncteurs* sont une abstraction des fonctions sous forme d'objet-fonction à l'aide de l'opérateur parenthèses. Les foncteurs sont largement utilisés par la bibliothèque standard STL.

À quoi bon les foncteurs puisqu'on dispose des fonctions ? Ils sont en fait un peu plus puissants que les fonctions. Ils permettent par exemple d'embarquer des paramètres supplémentaires en plus du code de la fonction elle-même. Ils sont aussi utilisables comme paramètre de patron d'algorithme (*algorithm templates*), le plus souvent sous forme d'objet temporaire.

L'exemple ci-après illustre l'addition élément par élément du contenu de deux tableaux `a` et `b`. Le résultat est stocké dans le premier tableau. Les deux tableaux contiennent des `double` et sont de même taille.

```
std::transform(a.begin(), a.end(), b.begin(),
               ➥ a.begin(), std::plus<double>());
```

Cet autre exemple montre comment inverser (`x = -x`) chaque élément :

```
std::transform(a.begin(), a.end(), a.begin(),
               ➥std::negate<double>());
```

Les fonctions d'addition et de négation sont *inlinées* implicitement. Ce qui rend le code au moins aussi performant que si vous l'aviez écrit vous-même, mais surtout plus synthétique et comportant moins de risque d'erreur lors de l'écriture.

Voici encore quelques exemples utiles :

```
std::vector<int> V(100);
std::generate(V.begin(), V.end(), rand);
```

Remplir un tableau avec des nombres aléatoires

```
struct plus_petit_abs :
    public std::binary_function<double, double, bool>
{
    bool operator() (double x, double y)
    {
        return fabs(x) < fabs(y);
    }
};
std::vector<double> V;
//…
std::sort(V.begin(), V.end(), plus_petit_abs());
```

Trier les éléments d'un tableau en ignorant leur signe

```
struct Somme : public std::unary_function<double, void>
{
    Somme() : somme(0) {}
    void operator()(double valeur) { somme+=valeur; }
    double somme;
};
std::vector<double> V;
//…
Somme resultat = std::for_each(V.begin(), V.end(), Somme());
std::cout << "Somme = " << resultat.somme << std::endl;
```

Somme des éléments d'un tableau illustrant l'avantage des foncteurs sur les fonctions

```
std::list<int> L;
//
std::list<int>::iterator it;
it = std::remove_if(L.begin(), L.end(), std::compose2(
    std::logical_and<bool>(),
    std::bind2nd(std::greater<int>(), 100),
    std::bind2nd(std::less<int>(), 1000) ));
L.erase(it, L.end());
```

Suppression des éléments (compris entre 100 et 1 000) d'une liste illustrant l'imbrication de foncteurs

```
struct Objet
{
    void affiche() { cout << /* … */ ; }
};
std::vector<Objet> V(100);
std::for_each(V.begin(), V.end(),
            ➥ std::mem_func<void,Objet>(&Objet::affiche));
```

Afficher tous les éléments d'un tableau via la fonction membre de l'objet utilisé

Astuce

Tous les foncteurs standard dérivent des deux classes, `unary_function` et `binary_function`. Elles ne contiennent que des `typedef`. Cela peut paraître totalement inutile, mais facilite grandement la programmation générique. Utilisez cette technique si vous décidez d'écrire vos propres foncteurs.

Voici pour mémoire et à titre d'exemple le contenu de ces deux classes :

```
template <class _Arg, class _Result>
struct unary_function
{
    typedef _Arg argument_type;  // le type de l'argument
    typedef _Result result_type; // le type de retour
};
template <class _Arg1, class _Arg2, class _Result>
struct binary_function
{
    typedef _Arg1 first_argument_type; // le type du premier
                                       ➥ argument
    typedef _Arg2 second_argument_type; // le type du
                                       ➥ deuxième argument
    typedef _Result result_type;   // le type de retour
};
```

Info

Si vous regardez les fichiers d'en-tête de la bibliothèque STL, vous rencontrez certainement la notion de générateur (*generator*), notamment dans le nommage de certains paramètres des patrons (*templates*). Cette notion correspond simplement à un foncteur ne prenant pas d'argument. On pourrait la représenter par cette classe de base :

```
template <class _Result>
struct generator
{
    typedef _Result result_type; // le type de retour
};
```

Les générateurs qui en hériteraient n'auraient plus qu'à définir l'`operator()()` sans argument.

La STL prédéfinit plusieurs types de foncteurs dans le fichier d'en-tête <functional>. Voici un récapitulatif de son contenu.

Foncteur sur les opérateurs standard

Foncteurs arithmétiques	Opérateur associé	Foncteurs logiques	Opérateur associé
`plus`	`+`	`equal_to`	`==`
`minus`	`-`	`not_equal_to`	`!=`
`multiplies`	`*`	`greater`	`>`
`divides`	`/`	`less`	`<`
`modulus`	`%`	`greater_equal`	`>=`
`negate`	`-` (unaire)	`less_equal`	`<=`
		`logical_and`	`&&`
		`logical_or`	`||`
		`logical_not`	`!` (unaire)

Inverseurs logiques

Foncteur	Helper	Code équivalent à l'exécution
`unary_negate`	`not1`[1]	`not1(pred)(a)` équivaut à `! pred(a)`
`binary_negate`	`not2`[1]	`not2(pred)(a,b)` équivaut à `! pred(a,b)`

1. Les fonctions `not1` et `not2` prennent un prédicat en argument et retournent, respectivement, une instance de `unary_negate` ou une instance de `binary_negate`.

Binders : pour fixer des paramètres

Foncteur	Helper	Code équivalent à l'exécution
`binder1st`	`bind1st`[1]	`bind1st(std::minus<float>,1.3)(a)` équivaut à `1.3 - a`
`binder2d`	`bind2nd`[1]	`bind2nd(std::minus<float>,1.3)(a)` équivaut à `a - 1.3`

1. Les fonctions `bind1st` et `bind2nd` prennent un prédicat en argument et retournent, respectivement, une instance de `binder1st` ou une instance de `binder2nd`.

Adaptateur de pointeur de fonctions : transformer des fonctions en foncteurs

Foncteur	Helper
`pointer_to_unary_function`	`ptr_fun`
`pointer_to_binary_function`	`ptr_fun`

Adaptateur de pointeur de fonctions membres : transformer des fonctions en foncteurs

Type	Helper	Description
`mem_fun_t`	`mem_fun`	Sert à appeler une fonction membre. `mem_func_t(p_func)(arg)` équivaut à `(*arg->*p_func)()`.
`const_mem_fun_t`	`mem_fun`	Idem pour une fonction membre `const`
`mem_fun_ref_t`	`mem_fun`	Sert à appeler une fonction membre par référence. `mem_func_t(p_func)(arg)` équivaut à `(*arg.*p_func)()`.
`const_mem_fun_ref_t`	`const_mem _fun_ref`	Idem pour une fonction membre `const`.
`mem_fun1_t`	`mem_fun`	Sert à appeler une fonction membre à un paramètre. `mem_func_t(p_func)(arg1,arg2)` équivaut à `(*arg1->* p_func)(arg2)`.
`const_mem_fun1_t`	`mem_fun`	Idem pour une fonction membre `const`
`mem_fun1_ref_t`	`mem_fun`	Sert à appeler une fonction membre à un paramètre, par référence. `mem_func_t(p_func)(arg1,arg2)` équivaut à `(*arg1.*p_func)(arg2)` .
`const_mem`	`mem_fun_ fun1_ref_t`	Idem pour une fonction membre `const`.

Foncteurs spécifiques

Type	Plate-forme	Description	Helper
`_Identity`	1	Voir `identity`	
`_Select1st`	1	Voir `select1st`	
`_Select2nd`	1	Voir `select2nd`	
`identity_element`	2 4	Pour `std::plus<_T>` retourne `_T(0)`, pour `std::multiplies<_T>` retourne `_T(1)`	
`identity`	2 3 4	Retourne l'argument tel quel	
`select1st`	2 3 4	Prend un `std::pair<>` en argument et retourne `arg.first`	
`select2nd`	2 3 4	Prend un `std::pair<>` en argument et retourne `arg.second`	
`project1st`	2 3 4	Retourne le premier argument	
`project2nd`	2 3 4	Retourne le deuxième argument	
`unary_compose`	2 3 4	`unary_functor(F1,F2)(x)` équivaut à `F1(F2(x))`	`compose1`
`binary_compose`	2 3 4	`unary_functor(F1,F2,F3)(x)` équivaut à `F1(F2(x),F3(x))`	`compose2`
`substrative_rng`	2 4	Générateur de nombres pseudo-aléatoires basé sur la méthode soustractive employée en FORTRAN	

1. GCC 3.4
2. GCC 3.4 *via* `<ext/functional>`
3. VC++2008 avec `_HAS_TRADITIONAL_STL`
4. SGI

5

Templates et métaprogrammation

Le *template* est une technique de programmation générique favorisant l'indépendance du code par rapport au type, et éventuellement au nombre d'arguments utilisés. Ce concept est très important car il permet d'augmenter le niveau d'abstraction du langage de programmation.

La *métaprogrammation* consiste à utiliser les modèles de telle sorte que le code source soit directement exécuté par le compilateur. Ces métaprogrammes peuvent générer des constantes ou des structures de données. Cette technique est largement utilisée en C++, et on la retrouve notamment dans la bibliothèque BOOST.

Traduction

Comment traduire *template* en bon français ? Le mot le plus proche dans notre langue se trouve dans le domaine de la couture : « patron ». Vous rencontrerez peut-être aussi le terme « modèle ». En couture, un patron s'utilise un peu comme un calque pour découper des pièces de tissus. On est ainsi sûr d'obtenir la même forme pour chaque vêtement produit. Mais il est bien sûr possible de changer le type d'étoffe. Je trouve l'analogie

particulièrement bien choisie, mais pour obtenir une traduction vraiment compréhensible, il serait préférable d'utiliser une expression comme *patron de classe* – bien qu'un peu longue à mon goût.

Dans le présent ouvrage, j'ai préféré garder l'anglicisme pour la concision. Deux raisons s'ajoutent à celle précitée. La première est qu'il correspond au mot-clé du langage lui-même. La deuxième raison est que l'usage de cet anglicisme est largement accepté.

Créer un modèle de classe réutilisable

```
template <(class   typename) T1 [, …]>
class Objet [: … ]
{
};
Objet<Type1 [, …]> objet;
```

```
template <(class   typename) T1 [, …]>
Type1 fonction(Type2 param [, …]);
Type1 res = fonction[<Type,[,…]>](…, t1, …);
```

La notion de « modèle » (de l'anglais *template*) est largement utilisée dans la STL et dans BOOST. Il est nécessaire de bien la comprendre pour tirer pleinement partie du langage C++. Les templates permettent de séparer la forme du fond, en laissant à la charge du compilateur la réécriture d'un même algorithme (le fond) à d'autres types de données (la forme). On peut donc voir les paramètres templates comme un moyen de paramétrer les types d'une structure ou d'une fonction, rendant ces dernières indépendantes du ou des types de données manipulés.

Pour mettre en œuvre un template, il faut impérativement préciser avec quels types on l'utilise. La seule exception où l'on peut omettre cette qualification a lieu lors de l'appel à une fonction template pour laquelle il n'y a aucune ambiguïté de résolution d'appel. Un exemple bien connu de fonction template sont les fonctions `min` et `max` fournit avec la STL (attention, Microsoft persiste à ne pas les fournir mais les remplace par ses vieilles macros `min(a,b)` et `max(a,b)`). Le code suivant est une version simplifiée de celle fournie par la STL de g++ :

```
template <typename _Tp>
inline const _Tp& min(const _Tp& __a, const _Tp& __b)
{
    if (__b < __a)
        return __b;
    return __a;
}

template<typename _Tp>
inline const _Tp& max(const _Tp& __a, const _Tp& __b)
{
    if (__a < __b)
        return __b;
    return __a;
}
```

Ces fonctions sont plus puissantes que des macros, car elles assurent de ne pas calculer deux fois des valeurs (avant comparaison et retour du `min` ou du `max`). Et elles illustrent parfaitement le paramétrage d'une fonction avec des types, grâce au template.

L'instanciation d'une fonction template, comme nous l'avons dit, peut se faire explicitement, mais aussi implicitement. Par exemple, un appel à `std::min(1,2)` est suffisant. Dans ce cas, `1` et `2` sont toutes deux des constantes entières et le compilateur comprend aisément qu'il faut instancier `std::min<int>()`. Par contre, lors d'un appel à `std::min(1,2.0)`, le

compilateur ne sait plus quoi faire (sauf s'il existe une fonction `min(int,double)` dans l'espace de nom `std`). Ici, il sera nécessaire d'employer une instanciation explicite : `std::min<int>(1,2.0)` pour transformer `2.0` en valeur entière, ou `std:min<double>(1,2.0)` pour transformer `1` en réel double précision.

Le compilateur utilise la loi du moindre effort pour instancier un template. Par exemple, utiliser un pointeur sur un objet template ne suffit à provoquer son instanciation. Celle-ci n'aura lieu que lors du déréférencement de celui-ci. De même, seules les fonctionnalités utilisées d'une classe template sont, par définition, instanciées.

Par exemple, dans l'exemple ci-après, seule la méthode `affiche()` sera instanciée. Ce programme compile donc parfaitement, même si la méthode `non_Codee()` n'est pas implémentée.

```
#include <iostream>
template <typename T>
class Exemple
{
public:
    void affiche();
    void non_Codee();
};
template <typename T>
void Exemple<T>::affiche()
{
    std::cout << "Je m'affiche()." << std::endl;
}
// la méthode Exemple<T>::non_Codee() n'est pas implémentée
int main(int,char**)
{
    // instanciation de Exemple<char>
    Exemple<char> ex;

    // instanciation de Exemple<char>affiche()
    ex.affiche();
}
```

class ou typename ?

Dans des paramètres templates, les mots-clés `class` et `typename` sont strictement équivalents. Ils permettent simplement de spécifier que le paramètre template est un type (ce qui n'est pas obligatoire). Certains programmeurs préfèrent utiliser systématiquement `typename` plutôt que `class`, car ils trouvent cela plus clair et conceptuellement plus beau. D'autres, par habitude, donnent une sémantique différente à ces mots-clés : `typename` pour les cas totalement génériques, comme les conteneurs STL, et `class` pour les cas particuliers réservés à certaines classes. D'autres encore utilisent exclusivement `class`. Vous l'aurez compris, c'est une affaire de choix personnel. J'espère simplement que ces quelques lignes vous auront permis d'y voir plus clair dans les habitudes de programmation que vous rencontrerez.

Créer une bibliothèque avec des templates

```
template [class struct] Objet<valeur [, …]>;
```

L'*instanciation explicite complète* permet de forcer l'instanciation d'un template. Cela est particulièrement utile pour les bibliothèques : ce faisant, il devient possible de coupler l'intérêt des templates et le fait de ne pas livrer votre implémentation. Par contre, cela peut également empêcher l'utilisateur de ladite bibliothèque d'utiliser vos objets templates sur d'autres types que ceux que vous aurez pré-instanciés, à moins de lui livrer aussi votre implémentation.

Pour réaliser une instanciation explicite, il suffit de préciser les paramètres templates et de faire précéder ce type explicité par le mot-clé `template`. Par exemple, si vous

vouliez forcer l'instanciation d'un modèle `Pile` pour des entiers, vous pourriez écrire :

```
template class Pile<int>;
```

Cette syntaxe peut être simplifiée pour les fonctions templates. Comme pour leur utilisation, il suffit que le compilateur puisse déterminer les types des paramètres utilisés. Le code `template int min(int, int);` suffit donc à forcer l'instanciation de la fonction `min()` citée précédemment.

Lorsqu'une fonction ou une classe template possède des valeurs par défaut pour tous ses paramètres templates, il n'est pas nécessaire de préciser à nouveau ces paramètres correspondants. Par exemple, le code suivant suffirait :

```
template class MaClasse<>;
```

Pour une instanciation concernant une bibliothèque dynamique, on utilise conjointement l'instanciation explicite et la spécification d'exportation du code. Pour simplifier l'exemple ci-après, qui contient très peu de fichiers, j'ai laissé la déclaration du `define` de l'implémentation dans le fichier source de la DLL.

```
#ifdef MADLL_IMPLEMENTATION
#define MADLL_API __declspec(dllexport)
#else
#define MADLL_EXPORT __declspec(dllimport)
#endif

template <…>
class MADLL_API MaClasse
{
    // …
   MADLL_API Type fonction(parametres);
};
```

Fichier en-tête de la bibliothèque

```
#define MADLL_IMPLEMENTATION2
#include "mon_header.hpp"
template <…>
MADLL_API Type MaClasse<…>::fonction(parametres)
{
    // …
}
// Instanciations explicites
template class MADLL_API MaClasse<double>;
template class MADLL_API MaClasse<int>;
```

Fichier source de la bibliothèque

```
#include "mon_header.hpp"
MaClasse<double> mc_d;
MaClasse<int>    mc_i;
MaClasse<float>  mc_f;
//…
mc_d.fonction(…); // OK
mc_i.fonction(…); // OK
mc_f.fonction(…); // ERREUR de link : fonction non définie
```

Exemple d'utilisation de la bibliothèque

Utiliser un type indirect dans un template

```
typename identificateur
```

Le mot-clé `typename` révèle tout son intérêt dans l'utilisation d'un type défini par une classe ou une structure lorsque cette

dernière est elle-même utilisée comme paramètre template. Considérez le code suivant :

```
class A
{
public:
    typedef int Type;
};
template <class T>
class B
{
    typename T::Type m_valeur;
};
B<A> b;
```

Si vous omettez le `typename`, le code ne compile plus : le compilateur n'arrive pas à comprendre que `T::Type` est bien un type. Et pour cause, il ne sait pas encore quel sera le type de `T`.

Changer l'implémentation par défaut fournie par un template

```
template<> Type fonction<…>(…) { … }
template<> class<…> … { … };
```

La *spécialisation* permet de changer l'implémentation par défaut fournie par un template. Pourquoi ? Le but du template est de ne pas dupliquer le code. Mais de ce fait, les algorithmes implémentés ainsi sont souvent génériques, parfois au détriment des performances. C'est là qu'intervient la spécialisation, partielle ou totale, des fonctions, classes ou membres de classes templates. Ce mécanisme

permet, comme son nom l'indique, de spécialiser ces templates pour certains paramètres donnés dans le but de fournir des algorithmes plus performants adaptés à ces cas particuliers.

Par exemple, une implémentation générique d'un mécanisme de pile travaillant sur des pointeurs sur objets peut avoir intérêt à être spécialisée pour des objets de petite taille ou des types fondamentaux du langage. En évitant ainsi un niveau d'indirection, les performances peuvent être améliorées pour ces types.

La *spécialisation totale* implique de fournir tous les paramètres templates de la fonction ou de la classe mais de les omettre dans la déclaration template. L'exemple ci-après montre comment spécialiser la fonction `std::min()` sur un type de données particulier.

```
struct  Structure
{
    int id;
    // …
};
namespace std
{
    template<>
    const Structure& min(const Structure& s1,
                         const Structure& s2)
    {
        if (s1.id < s2.id)
            return s1;
        return s2;
    }
}
```

Spécialiser partiellement l'implémentation d'un template

```
template<…> Type fonction<…>(…) { … }
template<…> class<…> … { … };
```

La spécialisation totale implique la réécriture de tout le code concerné par la spécification. Pour les classes, cela peut vite s'avérer fastidieux, surtout si vous n'avez besoin d'en spécialiser qu'une partie. Pour remédier à cela, il existe la *spécialisation partielle*. Elle peut être partielle sous deux aspects :

- soit elle ne spécialise qu'une partie des paramètres templates ;
- soit elle ne spécialise qu'une méthode de la classe.

Ces deux aspects peuvent être cumulés lorsqu'il s'agit d'une fonction membre.

La spécialisation partielle des paramètres templates permet de garder certains paramètres du templates comme indéfinis. Il est possible de changer la nature d'un paramètre template (passer d'un pointeur à un non pointeur et *vice versa*). L'exemple suivant montre comment faire des spécialisations partielles :

```
template<int i, class A, class B> class Objet           {…};
template<int i, class A>          class Objet<i,A,A*>   {…};
template<int i, class A, class B> class Objet<i,A*,B>   {…};
template<class A>                 class Objet<2,A,char> {…};
```

Ne vous emballez pas, vous constaterez à l'usage qu'il existe aussi certaines restrictions :

- une expression spécialisant un argument template ne doit pas utiliser un paramètre template de la spécialisation ;

```
template<int I, int J> class Obj {…};
template<int K> class Obj<K, 5*K> {…}; // ERREUR
```

- le type d'un paramètre spécialisant un argument template ne doit pas dépendre d'un paramètre de la spécialisation ;

```
template<class T, T t> struct Obj {…};
template<class T> struct Obj<T, 1> {…}; // ERREUR
```

- la liste des arguments de la spécialisation ne doit pas être implicitement identique au template d'origine.

```
template<class A, class B> class Obj {…};
template<class B, class A> class Obj<B,A> {…}; // ERREUR
```

Spécialiser une fonction membre

```
template<> Type Objet<…>::methode(…) { … }
template<…> Type Objet<…>::methode(…) { … }
```

Le dernier type de spécialisation concerne une méthode d'une classe template. Nul besoin de spécialiser toute une classe si spécialiser une ou plusieurs méthodes de celle-ci suffit. La technique est simple : il suffit de faire comme si la méthode était une fonction normale et spécialiser les paramètres templates souhaités. L'exemple suivant vous rappelle comment :

```
template <typename T>
class Objet
{
    T m_valeur;
```

```
public:
    Objet(T t)
    {
         m_valeur = t;
    }

    void set(T)
    {
        m_valeur = t;
    }

    T get() const
    {
        return m_valeur;
    }

    void afficher() const
    {
        std::cout << m_valeur << std::endl;
    }
};

// spécialisation d'une fonction membre
template <>
void Objet<int*>::print() const
{
    cout << *item << endl;
}
```

Exécuter du code à la compilation

La *métaprogrammation par templates* consiste à faire exécuter du code par le compilateur à l'aide des templates. Elle n'est pas l'apanage du C++ et se retrouve dans d'autres langages comme Curl, D, Eiffel, Haskell, ML ou XL. Pour être mise en œuvre en C++, cette technique repose sur deux opérations : la définition d'un template (ou plusieurs) et sa (ou leurs) instanciation.

Certains diront peut-être que les macros sont aussi exécutées à la compilation. Mais les macros ne peuvent effectuer des contrôles de type et manquent de sémantique : elles ne sont qu'un système de manipulation et de substitution de texte.

Les métaprogrammes templates n'ont, à proprement parler, pas de variables, et ne peuvent manipuler que des constantes. Les valeurs changent de par la récursion. La métaprogrammation peut du coup être vue comme un langage fonctionnel : c'est en fin de compte une sorte de machine de Turing.

```
// La métafonction factorielle
template <int n>
struct Factorielle
{
   enum { Result = n * Factorielle<n-1>::Result };
};

// On atteint la fin de la récursion grâce à une
// spécialisation partielle
template <>
struct Factorielle<0>
{
    enum { Result = 1 };
};

// Lancement du calcul par l'instanciation
Factorielle<5>::Result; // vaut 120 à la compilation

// Il faut utiliser des constantes uniquement
int i=5;
Factorielle<i>::Result; // ERREUR
const int j=5;
Factorielle<j>::Result; // OK, vaudra 120 à la compilation
```

Les métafonctions templates peuvent bien entendu être utilisées pour en construire d'autres, comme en programmation classique. L'exemple ci-après illustre l'utilisation de

la métafonction `Factorielle<>` pour construire la fonction mathématique.

$$C_n^i = C(n,i) = \frac{n!}{i!\,(n-i)!}$$

Figure 5.1 : Formule factorielle

```
template <int n, int i>
struct Comb
{
    enum
    {
        Result = Factorielle<n>::Result / ( Factorielle
        ➥<i>::Result * Factorielle<n-i>::Result)
    };
};
//
Comb<5,3>::Result;
```

Créer des méta-opérateurs/ métabranchements

```
template <bool g, class THEN, class ELSE>
struct IF
{
    typedef THEN Result;
};
template <class THEN, class ELSE>
struct IF<false, THEN, ELSE>
{
    typedef ELSE Result;
};
```

La métaprogrammation template peut même utiliser des tests conditionnels du type `if … then … else …` Cela est certes moins lisible mais fonctionne parfaitement. Le code suivant montre comment utiliser ce genre de métaopérateurs :

```
class Allocateur1 {…};
class Allocateur2 {…};

IF <UseAllocatorNumber==1,
    Allocateur1, // THEN type
    Allocateur2> // ELSE type
::Result allocator;
```

De la même manière, il est possible de faire comme si l'on disposait de valeurs booléennes. Le code ci-après illustre comment les simuler. Pour cela, nous réutilisons l'exemple précédant en adaptant la métaclasse de test `IF`, et lui adjoignions deux classes `FAUX` et `VRAI` qui feront office de valeurs booléennes. Ensuite, la métaclasse `isPointer<>` permet de déterminer si son paramètre template est ou non de type pointeur en renvoyant `VRAI` ou `FAUX`. Ce résultat peut enfin être utilisé lors d'un test `IF`.

```
struct FAUX {};
struct VRAI {};
template <class BOOL, class THEN, class ELSE>
struct IF
{
    typedef THEN Result;
};
template <class THEN, class ELSE>
struct IF<FAUX, THEN, ELSE>
{
    typedef ELSE Result;
};
template <class T> struct isPointer { typedef FAUX Result; };
```

```
template <class T> struct isPointer<T*> { typedef VRAI
➥ Result; };

template <class T>
class MonTableau
{
    IF< isPointeur<T>::Result, T, T*>::Result
    ➥ type_interne;
    // …
};
```

Si vous testez ce dernier exemple, vous remarquerez certainement que votre compilateur n'accepte pas le `IF<…>` présent dans la classe `MonTableau`. Cela est frustrant, mais vous montre les limites de l'interprétation des templates par le compilateur. C'est pour cette raison, entre autre, que les types *traits* ou d'autres biais sont utilisés.

Curiosité

Après les macros, après les fonctions inline, après les templates, voici la version métaprogrammée des fonction `min()` et `max()`. Encore ? Quel intérêt ? Deux raisons à cette curiosité : elles illustrent un aspect supplémentaire de la métaprogrammation et pourront vous être utiles pour « métaprogrammer ». Voici donc une implémentation possible de ces dernières pour des entiers :

```
template <int a, int b> struct Min { enum { Result =
➥ (a < b) ? a : b; }; };
template <int a, int b> struct Max { enum { Result =
➥ (a > b) ? a : b; }; };
```

Avantages et inconvénients de la métaprogrammation

- **Compromis entre temps de compilation et temps d'exécution.** Tout code template est analysé, évalué et déployé pendant la compilation. Elle peut donc prendre beaucoup de temps mais cela permet d'obtenir un code très rapide à l'exécution. Ce surcoût est généralement faible mais peut devenir non négligeable pour de gros projets ou des projets utilisant intensément la métaprogrammation template.
- **Programmation générique.** La métaprogrammation template permet de se concentrer sur l'architecture tout en laissant au compilateur la charge de générer le code nécessaire. La métaprogrammation fournit donc un moyen d'écrire du code générique et d'écrire moins de code. La maintenance du code en est facilitée.
- **Lisibilité.** En métaprogrammation C++, la syntaxe et les idiomes du langage sont souvent ardus à lire par rapport à du C++ classique. Ce code peut même rapidement devenir très difficile à déchiffrer. Un très bon niveau et surtout beaucoup d'expérience peuvent devenir indispensables. La maintenance d'un tel code, qui est censée être facilitée, peut s'en trouver réservée à un petit nombre de programmeurs qualifiés.
- **Portabilité.** Tous les compilateurs ne se valent pas. Surtout en ce qui concerne l'évaluation des templates. Un code reposant massivement sur cette notion peut devenir très difficile à porter, voir impossible si vous ne disposez pas d'un compilateur récent.

6

Gestion de la mémoire

Réserver et libérer la mémoire

```
Type* variable = new Type;
delete variable
```

```
double* variable = new double[n];
delete [] variable;
```

À l'instar de la fonction `malloc()`, l'opérateur `new` permet d'allouer dynamiquement de la mémoire. Notez toutefois qu'invoquer `new` provoquera l'initialisation des données d'un objet à l'aide de son constructeur, s'il existe. `new Type` alloue un objet, tandis que `new Type[n]` alloue un tableau de *n* objets contigus.

Attention

Utiliser `delete` au lieu de `delete[]`, ou inversement, peut entraîner un comportement imprévisible de votre application. D'une manière générale, utilisez `delete` pour un `new` et `delete[]` pour un `new[]`.

Faites toutefois très attention : dans certains cas, il n'est pas toujours facile de déterminer lequel utiliser. Pour bien comprendre la règle présentée, examinez cet exemple :

```
struct Livre { … };
typedef Bibliotheque Livre[8];
Livre* livre = new Livre;
//Bibliotheque *biblio = new Bibliotheque;   => ERREUR
Livre* biblio = new Bibliotheque;            // OK
delete livre;
// delete biblio;  => ERREUR : comportement imprévisible
delete[] biblio;   // OK
```

Si vous avez compris le principe, vous aurez remarqué que le `new Bibliotheque` est en fait un `new Livre[8]` caché qui nécessite bien une destruction par `delete[]`.

Redéfinir le système d'allocation mémoire

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

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

Vous pouvez redéfinir votre propre système d'allocation mémoire en redéfinissant les opérateurs globaux `new` et `delete`. Vous pouvez ainsi optimiser l'allocation mémoire de toute une application en utilisant un ou plusieurs pools de mémoire, ou utiliser des ressources système particulières de manière transparente pour le reste de votre code.

Attention

Redéfinir ces opérateurs globaux aura un effet sur toutes les allocations de votre programme.

D'autres solutions existent si vous souhaitez optimiser l'allocation mémoire de vos programmes :

- l'opérateur de placement ;
- une surcharge des opérateurs `new` et `delete` au niveau de vos classes ;
- la technique des allocateurs (elle est largement utilisée au sein de la STL).

Simuler une allocation d'objet à une adresse connue

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

Les opérateurs de placement simulent une allocation d'objet à une adresse connue. Ces opérateurs ne font que renvoyer l'adresse fournie.

Attention

Vous ne devez jamais utiliser `delete` sur des objets ainsi créés. Au besoin, appelez explicitement le destructeur pour appliquer les traitements qui y sont associés :

```
Obj* o = new(&quelqueChose) Obj;
//…
o->~Obj(); // appel explicite du destructeur sans
           ➥ détruire l'objet lui-même
```

Info

Cet opérateur utilise la même technique de surcharge que l'opérateur `new` sans exception (voir la section suivante).

Traiter un échec de réservation mémoire

```
try
{
    Type *objet = new Objet;
}
catch (std::bad_alloc)
{
    //
}
```

Un échec d'allocation mémoire se traduit par le lancement de l'exception `std::bad_alloc`. Si vous souhaitez que votre application ne se termine pas inopinément, vous avez intérêt à traiter un tel cas d'erreur en capturant cette dernière. Vous pourrez ainsi lancer un message d'erreur ou lancer un système de récupération de mémoire (comme un système de sauvegarde, libération, traitement, réallocation de la sauvegarde) à votre gré.

Une autre technique pour intercepter un échec d'allocation consiste à mettre en place un *handler* dédié. Lorsque celui-ci se termine normalement, l'opérateur `new` tente une nouvelle allocation. Si celle-ci échoue à nouveau, le handler est à nouveau invoqué. Voici à quoi pourrait ressembler un tel handler :

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

void newHandlerPerso()
{
    static int numEssai = 0;
    switch( numEssai )
    {
    case 0:
        // compacter le tas
        std::cout << "Compacter le tas\n";
        numEssai++;
        break;
    case 1:
        // nettoyer la mémoire du code inutile
        std::cout << "nettoyer la mémoire du code inutile\n";
        numEssai++;
        break;
    case 2:
        // utiliser un fichier swap
        std::cout << "utiliser un fichier swap\n";
        numEssai++;
        break;
    default:
        std::cerr << "Pas assez de mémoire\n";
        numEssai++;
        throw std::bad_alloc();
    }
}
```

Vous pourrez activer ce handler à volonté ainsi :

```
typedef void (*handler)();
handler ancienHandler = 0;

void test_handler()
{
    handler ancienHandler =
    ➥ set_new_handler( newHandlerPerso );
    // code protégé
    try
    {
        while ( 1 )
        {
            new int[5000000];
            cout << "Allocation de 5000000 ints." << endl;
        }
    }
    catch ( exception e )
    {
        cout << e.what( ) << " xxx" << endl;
    }
    //
    set_new_handler( ancienHandler );
}

int main(int,char**)
{
    test_handler();
    return 0;
}
```

Désactiver le système d'exception lors de l'allocation

```
#include <new>
Type *obj = new(std::nothrow) Type;
int  *a   = new(std::nothrow) int[1000000000];
```

Le fichier d'en-tête standard <new> définit une structure appelée std::nothrow_t dans l'espace de nom std. Cette structure vide sert de directive en forçant l'appel de new à être redirigé sur ces surcharges des opérateurs :

```
void* operator new(size_t size, const std::nothrow_t&);
void* operator new(void* v, const std::nothrow_t& nt)
```

Déclaration apparaissant dans <new>

<new> définit également une instance extern const nothrow_t nothrow; permettant d'utiliser de manière uniforme ladite syntaxe.

```
Objet* objets = new(std::nothrow) Objet[1000000000];
if (objets == 0)
{
    std::cout << "Erreur allocation memoire…" << std::endl;
}
```

Optimiser l'allocation avec un pool mémoire

```
boost::pool<> pool(tailleEnOctets);
boost::object_pool<Type> pool;
boost::singleton_pool<PoolTag, TailleEnOctets> pool;
boost::pool_allocator<Type>
```

Plutôt que de développer vous-même un système de *pools* (ou bacs) de mémoire, utilisez ceux fournis par BOOST. Pour en savoir plus sur cet ensemble de bibliothèques, reportez-vous au Chapitre 13, consacré à BOOST.

Pour les pools de type objet, détruire le pool libère implicitement tous les blocs mémoire alloués par celui-ci. Pour les pools de type singleton, les blocs mémoire ont une durée de vie égale au programme. Ils peuvent donc être aisément partagés. Du coup, ils ont été codés pour être *thread-safe* (avec pour corollaire un coût à l'exécution). Pour libérer les ressources mémoire d'un pool de type singleton, il faut l'instancier avec les fonctions membres `release_memory()` ou `purge_memory()`.

`purge_memory()` libère tous les blocs mémoire alloués. `release_memory()` libère tous les blocs internes non utilisés.

Attention

Certaines implémentations préfèrent retourner 0 (NULL) plutôt que de déclencher une exception lorsqu'il n'y a plus de mémoire. Soyez donc bien attentif à ce point lors de leur utilisation.

Les `boost::pool<>` sont de type objet et retournent NULL en cas d'échec de réservation mémoire. Dans l'exemple

ci-après, nul besoin de se préoccuper de la désallocation mémoire, puisque le pool s'en chargera lors de sa destruction à la fin de la fonction.

```
#include <boost/pool/pool.hpp>
void fonction()
{
    boost::pool<> p(sizeof(int));
    for (int i=0; i < 100000; ++i)
    {
        int* const i_ptr = (int*)p.malloc();
        // …
    }
}
```

Les `boost::object_pool<>` sont de type objet et retournent NULL en cas d'échec d'allocation mémoire. Ils tiennent compte du type d'objet alloué. Là encore, tous les objets concernés sont détruits avec le pool.

```
#include <boost/pool/object_pool.hpp>
class Objet { … };
void fonction()
{
    boost::object_pool<Objet> p;
    for (int i=0; i < 100000; ++i)
    {
        Objet* const i_ptr = p.malloc();
        // …
    }
}
```

Les `boost::singleton_pool` sont de type singleton. Ils retournent NULL en cas d'échec d'allocation mémoire (encore). Par contre, la mémoire n'est libérée qu'explicitement. Le *tag* permet de créer plusieurs instances différentes de ce pool de mémoire.

```
#include <boost/pool/singleton_pool.hpp>
struct MonPool_Tag{};
typedef boost::singleton_pool<MonPool_Tag,
sizeof(int)> MonPool;
void fonction()
{
    for (int i=0; i < 100000; ++i)
    {
        int* const i_ptr = (int*)MonPool::malloc();
        // …
    }
    // Appel explicite pour libérer la mémoire
    MonPool::purge_memory();
}
```

Les `boost::pool_alloc` sont de type singleton. Attention, ceux-ci déclenchent une exception en cas d'échec d'allocation mémoire. Ils sont conçus sur les `boost:singleton_pool` et sont compatibles avec les allocateurs STL. Dans l'exemple ci-après, la mémoire ne sera pas libérée à la fin de la fonction. Pour le forcer, il faut appeler `boost::singleton_pool<boos::pool_allocator_tag, sizeof(int)>::release_memory()`.

```
#include <boost/pool/pool_alloc.hpp>
#include <vector>
void fonction()
{
    std::vector<int, boost::pool_allocator<int> > V;
    for (int i=0; i < 100000,  ++i)
         V.push_back(33);
}
```

7

Exceptions

Obtenir un code fiable est certainement une des grandes préoccupations des programmeurs. De manière générale, deux possibilités de gestion des erreurs existent : l'utilisation de codes de retour ou les exceptions. Libre à vous d'utiliser l'une ou l'autre de ces possibilités. Ce chapitre vous permettra d'appréhender de manière sereine les exceptions en C++ afin de les utiliser judicieusement.

Principe

```
#include <exception>
#include <stdexcept>
```

```
try
{
    // code pouvant déclencher directement
    // ou indirectement une exception
}
catch(type1)
{
    // code du gestionnaire traitant les exceptions
    // correspondant au type1
}
catch(type2)
```

```
{
    // code du gestionnaire traitant les exceptions
    // correspondant au type2 (et non traitées par un
    // des gestionnaires précédents)
}
// …
catch(...)
{
    // code du gestionnaire traitant toute exception
    // non gérée par un des gestionnaires précédents
}
```

```
throw expression;
```

Les exceptions permettent une forme de communication directe entre les fonctions bas niveau et les fonctions haut niveau qui les utilisent. L'utilité de cette communication est d'autant plus évidente lorsque la différence de niveau entre le déclencheur et le gestionnaire est importante. Le principe est le suivant.

Une fonction détecte un comportement exceptionnel et déclenche une exception à l'aide de `throw` *`expression`*.

L'exception est construite grâce à l'*`expression`* fournie à l'instruction `throw`. Elle peut être un nombre entier ou flottant, une chaîne ou un objet. Le plus souvent, il s'agit d'un objet dont la classe dérive de `std::exception`, contenant les informations relatives aux caractéristiques de l'événement survenu.

L'exception stoppe le fonctionnement normal du programme. Elle détruit tous les objets construits localement en remontant bloc par bloc, fonction par fonction (dans la pile d'appel) jusqu'à rencontrer un gestionnaire (`catch`) qui la gère. Si aucun gestionnaire approprié n'est trouvé,

la fonction `terminate()` est appelée ; elle provoque la fin du programme. Si un gestionnaire est utilisé, le programme poursuit son cours normal à l'instruction qui suit le dernier gestionnaire de son groupe de catch.

Info

La destruction des objets locaux est un des principes fondamentaux les plus utiles du système des exceptions en C++.

Voici un exemple simple montrant comment garantir que l'on n'accède pas à un élément en dehors des limites d'un tableau :

```
class Tableau3
{
    int valeur[3];
public:
    int get(int i) const
    {
        if (i<0 || i>3)
            throw std::out_of_range();
        return valeur[i];
    }
    // …
};
// …
    int v = tab.get(4);
```

Sélection du gestionnaire de l'exception

Le premier gestionnaire éligible, c'est-à-dire dont le type est compatible avec l'exception, est utilisé. Le test d'éligibilité d'un gestionnaire `catch` est soumis à des règles de conversion proche de celles de la surcharge de fonction (voir le Chapitre 2, section « Surcharge »).

Un gestionnaire `catch(T)`, `catch(const T)`, `catch(T&)` ou `catch (const T&)` est compatible avec une exception de type `E` si :

- `T` et `E` désignent le même type ou ;
- `T` est une classe de base visible (accessible) de `E` ou ;
- `T` et `E` sont des types pointeurs et `E` peut être converti en `T` par une conversion standard.

Contrairement à la surcharge de fonction, si `T` et `E` ne sont pas de type pointeur, aucune conversion standard n'est tentée.

Voici un exemple :

```
#include <stdexcept>
//…
try
{
    throw std::bad_alloc();
}
catch(std::exception& e)
{
    // ce gestionnaire correspond à l'exception
       ➥ déclenchée
    // car std::bad_alloc hérite de std::exception.
}
catch(std::bad_alloc&)
{
    // jamais atteint,
    // car le gestionnaire précédent est utilisé.
}
```

Notez qu'un bon compilateur vous signalera votre étourderie (g++ le fait), par un message de ce type :

```
excep.cpp: In function `int f(bool)':
excep.cpp:22: warning: exception of type
   'std::bad_alloc' will be caught
excep.cpp:13: warning:    by earlier handler for
   'std::exception'
```

Attention

Les débogueurs modernes permettent de s'arrêter automatiquement sur le déclenchement d'une exception. Souvent, cela permet de trouver très rapidement l'endroit d'un bogue. Toutefois, si votre programme fait un usage abusif des exceptions, le débogueur se mettra très souvent en pause, rendant la recherche du bogue au mieux fastidieuse, au pire impraticable.

Peut-être alors tenterez-vous d'utiliser l'option « S'arrêter sur les exceptions non gérées ». Mais si votre programme met en œuvre des gestionnaires très génériques, le débogueur ne s'arrêtera jamais sur le lieu du bogue.

Vous l'aurez compris : n'utilisez pas abusivement le mécanisme des exceptions comme système de traitement d'erreur « normale » ! Comme son nom l'indique, *une exception doit rester une exception !* Il est prudent de ne les utiliser que pour les réels cas d'anomalies.

Attention

L'appel d'un gestionnaire `catch` est traité comme un appel de fonction. Il commence donc, selon le type de paramètre transmis, par faire une copie de l'argument ! Soyez donc vigilants et prenez plutôt l'habitude de déclarer le type par référence, surtout avec les objets. Par exemple utilisez `catch(const MonException&)` plutôt que `catch(MonException)`.

Transmettre une exception

```
throw;
```

Dans un gestionnaire `catch`, vous pouvez utiliser `throw;` pour redéclencher l'exception en cours de traitement comme si elle n'avait pas été attrapée. Le code qui pourrait suivre cette instruction dans ledit gestionnaire ne sera pas exécuté.

Le programme suivant illustre le comportement de la transmission d'exception :

```
#include <stdexcept>
#include <iostream>

int f(bool transmettre)
{

    try
    {
        throw std::bad_alloc();
    }
    catch(std::exception& e)
    {
        // ce gestionnaire correspond à l'exception
           ➥ déclenchée
        // car std::bad_alloc hérite de std::exception.
        std::cout << "traitement 1" << std::endl;
        if (transmettre)
            throw;
        std::cout << "traitement 1 bis" << std::endl;
    }
    catch(std::bad_alloc&)
    {
        // jamais atteint,
        // car le gestionnaire précédent est utilisé.
        std::cout << "traitement 2" << std::endl;
    }
}

int main(void)
{
    std::cout << "--- essai A ---" << std::endl;
    try
    {
        f(false);
    }
    catch(...)
    {
        std::cout << "traitement 3" << std::endl;
    }
```

```
    std::cout << "--- essai B ---" << std::endl;
    try
    {
        f(true);
    }
    catch(...)
    {
        std::cout << "traitement 3" << std::endl;
    }

    return 0;
}
```

Il produit la sortie suivante :

```
--- essai A ---
traitement 1
traitement 1 bis
--- essai B ---
traitement 1
traitement 3
```

Expliciter les exceptions

```
type fonction(parametres) throw(T,…)
type fonction(parametres) throw()
```

L'explicitation des types d'exceptions (ou exceptions compatibles) pouvant être déclenchées par une fonction ne fait pas partie de sa signature. Elle permet néanmoins au compilateur de faire certaines vérifications et de vous mettre en garde par des messages (le plus souvent des *warnings*).

```
#include <stdexcept>
#include <iostream>

void excep_bad_cast()
{
    throw std::bad_cast();
}
void excep_range_error()
{
    throw std::range_error(std::string("toto"));
}

int f(bool transmettre) throw(std::range_error,
➥ std::bad_alloc)
{
    try
    {
        // À décommenter au choix pour les tests
        // excep_bad_cast();
        // excep_range_error();
        throw std::bad_alloc();
    }
    catch(std::bad_alloc&)
    {
        std::cout << "traitement 2" << std::endl;
        throw;
    }
}

int main(void)
{
    std::cout << "--- essai A ---" << std::endl;
    try
    {
        f(false);
    }
    catch(...)
    {
        std::cout << "traitement 3" << std::endl;
    }

    return 0;
}
```

En décommentant la ligne `excep_bad_cast();` vous obtiendrez (avec g++) :

```
--- essai A ---
terminate called after throwing an instance of
➥'std::bad_cast'
  what():  St8bad_cast
Abort trap
```

En décommentant la ligne `excep_range_error();` vous obtiendrez (avec g++) :

```
--- essai A ---
traitement 3
```

Utiliser ses propres implémentations des fonctions `terminate()` et `unexpected()`

```
typedef void (*handler)();
handler set_terminate(handler);
handler set_unexpected(handler);
```

Vous pouvez utiliser vos propres implémentations des fonctions `terminate()` et `unexpected()`. Elles retournent le handler en vigueur avant l'appel. À vous de le sauvegarder pour éventuellement le remettre.

Attention

Votre fonction `terminate()` doit impérativement terminer le programme en cours. Elle ne doit ni retourner à l'appelant, ni lancer une quelconque exception.

Votre fonction `unexpected()` peut ne pas terminer le programme. Dans ce cas, elle doit impérativement déclencher une nouvelle exception.

L'exemple suivant montre comment mettre en place vos propres gestionnaires :

```
void my_stop()
{
    QMessageBox::critical(0,"Erreur irrécupérable",
         "L'application a rencontré une erreur
          ➥ inattendue"
         " et va se fermer…");
    exit(-1);
}
void fonction()
{
    std::terminate_handler orignal =
    ➥ set_terminate(my_stop);
    //…
    set_terminate(original);
}
```

Utiliser les exceptions pour la gestion des ressources

Considérez par exemple un code effectuant les opérations suivantes :

1. Réserver une ressource.
2. Faire un traitement.
3. Libérer la ressource.

Ce type de code peut paraître anodin et sans danger. Et pourtant que se passerait-il si le traitement devait déclencher une exception ? Votre ressource ne serait jamais libérée. Vous pourriez alors opter pour cette solution :

```
try
{
    reserver_ressource();
    traitement();
}
catch(...)
{
    liberer_ressource();
}
```

Cette solution peut être viable, mais devient vite impraticable dans certaines situations. Considérez le cas de plusieurs réservation/traitement avec une libération du tout à la fin, ou bien plusieurs réservations suivies d'un traitement pour enfin libérer toutes vos ressources… Vous rencontrerez ce cas régulièrement avec les allocations mémoire. Que se passerait-il en utilisant la méthode précédente ?

```
try
{
    ObjetA * objetA = new ObjetA();
    ObjetB * objetB = new ObjetB();
    // traitement
}
catch(...)
{
    delete objetA;
    delete objetB;
}
```

Si une exception `bad_alloc` est déclenchée lors de l'allocation de l'`objetB`, l'instruction `delete objetB` aura un comportement imprévisible ! Même un test sur la valeur du pointeur n'est pas suffisant.

Astuce

Utilisez des objets pour allouer des ressources et non des pointeurs.

Voici une version simpliste de `std::auto_ptr` pouvant servir de modèle pour résoudre tous vos problèmes :

```
template<class T>
struct Ptr
{
    Ptr(T* p) : ptr(p) {}
    ~Ptr() { delete p; }
    T* get() const { return ptr; }
    T* operator->() const { return ptr; }
private:
    T* ptr;
};
```

Le code précédent devient alors non seulement plus simple, mais totalement sûr. En effet, si la deuxième allocation échoue, le deuxième objet ne sera pas créé et son destructeur ne sera pas appelé. Par contre, de par le fonctionnement du C++, nous avons la garantie que le premier sera bien détruit, provoquant la libération de la ressource allouée.

```
Ptr<Objet> ptrA(new ObjetA());
Ptr<ObjetB> ptrB(new ObjetB());
// traitement identique
```

Exceptions de la STL

Vous trouverez dans le tableau suivant toutes les exceptions utilisées (ou qui le seront dans de prochaines évolutions) par la STL.

<exception> (inclus par <stdexcept>)

Type	Description
`exception`	Classe de base de toutes les exceptions pouvant être déclenchées par la bibliothèque STL ou certaines expressions du langage C++. Vous pouvez définir vos propres exceptions en héritant de celle-ci si vous le souhaitez.
+ `bad_exception`	Peut être déclenchée si une exception non répertoriée dans la liste des exceptions autorisées par une fonction est déclenchée

<stdexcept>

Type	Description
+ `logic_error`	Représente les problèmes de logique interne au programme. En théorie, ils sont prévisibles et peuvent souvent être repérés par une lecture attentive du code (comme par exemple lors de la violation d'un invariant de classe).
+ + `domain_error`	Déclenchée par la bibliothèque, ou par vous, pour signaler des erreurs de domaine (au sens mathématique du terme)
+ + `invalid_argument`	Déclenchée pour signaler le passage d'argument(s) invalide(s) à une fonction
+ + `length_error`	Déclenchée lors de la construction d'un objet dépassant une taille limite autorisée. C'est le cas de certaines implémentations de `basic_string`.
+ + `out_of_range`	Déclenchée lorsqu'une valeur d'un argument n'est pas dans les limites (bornes) attendues. La méthode `at()` des `vector` effectue un test de borne et déclenche cette exception si besoin.
+ `runtime_error`	Représente les problèmes sortant du cadre du programme. Ces problèmes sont difficilement prévisibles et surviennent lors de l'exécution du programme.
+ + `range_error`	Déclenchée pour signaler une erreur de borne issue d'un calcul interne

<stdexcept> *(suite)*

Type	Description
+ + `overflow_error`	Déclenchée pour signaler un débordement arithmétique (par le haut)
+ + `underflow_error`	Déclenchée pour signaler un débordement arithmétique (par le bas)

<type_info>

Type	Description
+ `bad_cast`	Déclenchée lors d'une conversion dynamique `dynamic_cast` invalide
+ `bad_typedid`	Déclenchée si un pointeur nul est transmis à une expression `typeid()`

<new>

Type	Description
+ `bad_alloc`	Déclenchée par `new` (dans sa forme standard, c'est-à-dire sans le `nothrow`) pour signaler un échec d'allocation mémoire

Info

Aucune exception ne peut (et ne doit, si vous écrivez vos propres classes en héritant de `exception`) déclencher pendant l'exécution d'une fonction membre de la classe `exception`.

Astuce

La classe de base `exception` possède une fonction virtuelle `const char* what()` qui retourne une chaîne de caractères la « décrivant ».

8

Itérateurs

Les *itérateurs* sont une généralisation des pointeurs. Notez que je ne parle pas ici de LA généralisation, mais bien d'UNE généralisation. D'autres types de généralisation des pointeurs existent, comme la notion de pointeur fort/pointeur faible.

Un itérateur est une interface standardisée permettant de parcourir (ou *itérer*) tous les éléments d'un conteneur. Les itérateurs présentent la même sémantique que les pointeurs par plusieurs aspects :

- ils servent d'intermédiaire pour atteindre un objet donné ;
- si un itérateur pointe sur un élément donné d'un ensemble, il est possible de l'incrémenter pour qu'il pointe alors sur l'élément suivant.

Les itérateurs sont un des éléments essentiels à la programmation générique. Ils font le lien entre les conteneurs et les algorithmes. Ils permettent d'écrire un algorithme de manière unique quel que soit le conteneur utilisé, pourvu que ce dernier fournisse un moyen d'accéder à ses éléments à travers un itérateur. Vous aurez certainement déjà compris que ce qui importe pour l'algorithme n'est plus de savoir sur quel type de conteneur mais sur quel type d'itérateur il travaille.

Il est donc possible et même nécessaire d'avoir plusieurs (ou suffisamment) catégories d'itérateurs si l'on veut pouvoir écrire un large éventail d'algorithmes génériques. Rassurez-vous, la bibliothèque standard (STL) vous a mâché le travail en définissant plusieurs itérateurs, et même mieux, une hiérarchie d'itérateurs.

Les différents concepts

Les itérateurs sont utilisés pour accéder aux membres des classes conteneurs de la STL. Ils peuvent être utilisés d'une manière analogue aux pointeurs. Par exemple, il est possible d'utiliser un itérateur pour parcourir les éléments d'un `std::vector`. Il existe plusieurs types d'itérateurs. Vous les retrouverez dans le tableau ci-après. Plus que des types, ce sont surtout des concepts.

Les différents itérateurs STL

Type	Description
InputIterator	Lire des valeurs en avançant. Ils peuvent être incrémentés, comparés et déréférencés.
OuputIterator	Écrire des valeurs en avançant. Ils peuvent être incrémentés et déréférencés.
ForwardIterator	Lire et/ou écrire des valeurs en avançant. Ils combinent les fonctionnalités des *InputIterator* et des *OutputIterator* avec la possibilité de stocker les itérateurs eux-mêmes.
BidirectionalIterator	Lire et/ou écrire des valeurs en avançant et/ou en reculant. Ils sont comme les *ForwardIterator* mais ils peuvent être incrémentés et/ou décrémentés.

Type	Description
RandomAccessIterator	Lire et/ou écrire des valeurs dans un ordre aléatoire. Aléatoire ne signifie pas ici au hasard mais dans n'importe quel ordre, selon vos besoins. On le traduit parfois par « itérateur d'accès direct ». Ce sont les itérateurs les plus puissants ; ils ajoutent aux fonctionnalités des *BidirectionalOperator* les possibilités d'effectuer des opérations semblables à l'arithmétique des pointeurs et la comparaison des pointeurs.

Chaque classe conteneur de la STL est associée à un type ou concept d'itérateur. Il en est de même pour tous les algorithmes fournis par la STL : chacun utilise un type d'itérateur particulier. Par exemple, les vecteurs (`std::vector`) sont associés aux *RandomAccessIterator* (itérateur à accès aléatoire). Il en découle qu'ils peuvent être utilisés avec tous les algorithmes basés sur ce type d'itérateur. Comme cet itérateur englobe toutes les caractéristiques des autres types d'itérateurs, les vecteurs peuvent aussi être utilisés avec tous les algorithmes conçus pour les autres types d'itérateurs.

Le code suivant crée et utilise un itérateur avec un vecteur :

```
std::vector<int> V;
std::vector<int>::iterator iterateur;
for (int i=0; i<10; i++)
    V.push_back(i);
int total = 0;
for (iterateur = V.begin(); iterateur != V.end();
iterateur++)
    total += *iterateur;
std::cout << "Total = " << total << std::endl;
```

Notez que vous pouvez accéder aux éléments du conteneur en déréférençant l'itérateur.

Comprendre les iterator_traits

```
#include <iterator>
template <class Iterator>
struct iterator_traits
{
   typedef typename Iterator::iterator_category
   ➥ iterator_category;
   typedef typename Iterator::value_type      value_type;
   typedef typename Iterator::difference
   ➥_type   difference_type;
   typedef typename Iterator::pointer         pointer;
   typedef typename Iterator::reference       reference;
};

template <class T>
struct iterator_traits<T*>
{
   typedef random_access_iterator_tag iterator_category;
   typedef T                          value_type;
   typedef ptrdiff_t                  difference_type;
   typedef T*                         pointer;
   typedef T&                         reference;
};
```

Une des caractéristiques les plus importantes des itérateurs est d'avoir des types associés. À un type d'itérateur est donc associé d'autres types, comme :

- **`value_type`**. Le type de valeur est le type d'objet sur lequel l'itérateur pointe.
- **`difference_type`**. Le type de distance (ou type de la différence) entre deux itérateurs. Ce peut être un type intégral comme `int`.

Les pointeurs, par exemple, sont des itérateurs. Par exemple, le type de valeur de `int*` est `int`. Le type de distance est `ptrdiff_t` puisque si `p1` et `p2` sont des pointeurs, l'expression `p1 - p2` est de type `ptrdiff_t`.

Les algorithmes génériques ont souvent besoin d'accéder à des types associés. Un algorithme qui prend en paramètre une séquence d'itérateurs peut avoir besoin de déclarer une variable temporaire du type de la valeur pointée. La classe `iterator_traits` fournit un mécanisme pour de telles déclarations.

Une technique de prime abord évidente pour une telle fonctionnalité serait d'imposer à tous les itérateurs d'embarquer ces déclarations de type. Ainsi, le type de valeur d'un itérateur `I` serait `I::value_type`. Néanmoins, cela ne fonctionne pas bien. Les pointeurs sont des itérateurs et ne sont pas des classes : si `I` était, par exemple, un `int*`, il serait impossible de définir `I::value_type`. Le concept de *traits template* a été créé pour remédier à ce genre de difficultés.

Info

Une classe de caractéristiques ou *traits class* est une classe utilisée en lieu et place de paramètres templates. Elle embarque des constantes et types utiles. Elle offre un nouveau niveau d'indirection pour résoudre de nombreux problèmes. À l'origine, ce concept portait le nom de « bagage » ou « valise » (*baggage*) contenant des caractéristiques (*traits*) et a fini par être appelé directement *traits*.

Si vous définissez un nouvel itérateur `I`, vous devez vous assurer que `iterator_traits<I>` soit correctement défini. Vous avez deux possibilités pour cela. Premièrement, vous définissez votre itérateur de telle sorte qu'il contienne les types associés `I::valu_type`, `I::difference_type` et ainsi de suite. Deuxièmement, vous pouvez spécialiser `iterator_traits` avec votre type. La première technique est la plupart du temps plus pratique, surtout lorsque vous pouvez créer votre itérateur en héritant d'une des classes de base `input_iterator`, `output_iterator`, `forward_iterator`, `bidirectional_iterator`, ou `random_access_iterator`.

L'exemple suivant illustre l'utilisation de cette classe *traits* :

```
template <class InputIterator>
std::iterator_traits<InputIterator>::value_type
derniere_valeur(InputIterator debut, InputIterator fin)
{
    std::iterator_traits<InputIterator>::value_type
    ➥ resultat = *debut;
    for (++debut; debut != fin ; ++debut)
        resultat = *debut;
    return resultat;
}
```

Cette fonction renvoie la valeur du dernier élément d'une séquence. Bien évidemment, ce n'est pas un exemple de « bon » code. Il existe de bien meilleures façons d'arriver au même résultat, surtout si l'itérateur est du type `bidirectional_iterator` ou `forward_iterator`. Mais vous pouvez noter qu'utiliser les `iterator_traits` est le seul moyen d'écrire une fonction générique de ce type.

Calculer la distance entre deux itérateurs

```
#include <iterator>
iterator_traits<InputIterator>::difference_type
distance(InputIterator debut, InputIterator fin);
void distance(InputIterator debut, InputIterator
➥ fin, Distance& n);
```

Ces fonctions retournent la distance `d` entre `debut` et `fin`, autrement dit `debut` doit être incrémenté `d` fois pour qu'il devienne égal à `fin`. La première version de `distance()` retourne cette distance, la deuxième l'ajoute au `n` donné.

Info

La deuxième version de `distance()` a été définie dans la STL originale, et la première version a été introduite par le comité de standardisation C++. La définition a été modifiée car l'ancienne était source d'erreur. En effet, cette version requérait l'utilisation d'une variable temporaire et son utilisation n'était pas intuitive : elle ajoutait à `n` la distance entre `debut` et `fin` plutôt que simplement lui affecter cette valeur. Du coup, oublier d'initialiser `n` à zéro était une erreur répandue.

Bien entendu, les deux versions sont implémentées par souci de compatibilité, mais vous êtes encouragé à toujours utiliser la version du comité (et qui renvoie la distance).

L'exemple ci-après montre que la fonction `distance()` fonctionne même avec les conteneurs à accès séquentiel (au sens propre du terme), mais alors sa complexité est linéaire et plus en temps constant.Vous devez donc l'utiliser en connaissance de cause.

```
std::list<int> L;
L.push_back(0);
L.push_back(1);
assert(std::distance(L.begin(), L.end()) == L.size());
```

Attention

Si votre compilateur ne supporte pas la *spécialisation partielle* (voir la section « Spécialiser partiellement l'implémentation d'un template » au Chapitre 5), vous ne pourrez pas utiliser les fonctions utilisant `std::iterator_traits<>`. Cette classe repose en effet largement sur cette spécificité du langage C++. Rassurez-vous, la plupart des compilateurs modernes supportent la spécialisation partielle.

Déplacer un itérateur vers une autre position

```
#include <iterator>
void advance(InputIterator& i, Distance n);
```

`advance()` incrémente `i` de la distance `n`. Si `n` est positif, cela est équivalent à exécuter `n` fois `++i`, et s'il est négatif `|n|` fois `--i`. Si `n` est nul, la fonction n'a aucun effet.

L'exemple suivant illustre l'utilisation de cette fonction sur les listes :

```
std::list<int> L;
L.push_back(0);
L.push_back(1);
std::list<int>::iterator it = L.begin();
std::advance(it, 2);
assert(it == L.end());
```

Astuce

`distance()` et `advance()` prennent tout leur intérêt avec la programmation générique (c'est-à-dire avec les templates). Grâce à elles, vous n'êtes plus obligé de connaître le type de conteneur sur lequel vous travaillez. Ainsi, vous pouvez passer le type de conteneur comme paramètre template, et le tour est joué : votre algorithme devient générique.

Comprendre les itérateurs sur flux d'entrée/lecture

```
#include <iterator>
class istream_iterator<T, Distance>;
```

Un `istream_iterator<>` est un *InputIterator*. Il effectue une sortie formatée d'un objet de type `T` sur un `istream<>` particulier donné. Quand la fin du flux est atteinte, l'`istream_iterator<>` renvoie une valeur d'itérateur particulière qui signifie « fin de flux ». La valeur de cet itérateur de fin peut être obtenue par le constructeur vide. Vous pourrez noter que toutes les caractéristiques d'un *InputIterator* sont applicables.

```
double somme = 0;
std::istream_iterator<double, char>  is(std::cin);
while (is != std::istream_iterator<double, char>())
{
    somme = somme + *is;
    ++is;
}
std::cout << "Somme = " << somme << std::endl;
```

Cet exemple lit des valeurs réelles sur la console (jusqu'à un Ctrl+Z) puis écrit leur somme.

L'exemple suivant mémorise une suite d'entiers, fournis à travers l'entrée standard, dans un tableau :

```
std::vector<int> V;
std::copy(std::istream_iterator<int>(std::cin),
          std::istream_iterator<int>(),
          std::back_inserter(V));
```

Comprendre les itérateurs sur flux de sortie/écriture

```
#include <iterator>
class ostream_iterator<T>;
```

Un `ostream_iterator<>` est un *OutputIterator* qui effectue une sortie formatée d'objets de type `T` sur un `ostream` (flux de sortie) particulier. Notez que toutes les restrictions d'utilisation d'un *OuputIterator* doivent être observées.

L'exemple suivant copie les éléments d'un vecteur sur la sortie standard, en en écrivant un par ligne :

```
std::vector<int> V;
// …
std::copy(V.begin(), V.end(),
        ➥ std::ostream_iterator<int>(std::cout, "\n"));
```

Utiliser les itérateurs de parcours inversé

```
#include <iterator>
class reverse_iterator<RandomAccessItertator, T,
➥ Reference, Distance>;
class reverse_bidirectional_iterator<
➥BidirectionalIterator, T, Reference, Distance>;
```

`reverse_iterator<>` et `reverse_bidirectional_iterator<>` sont des adaptateurs d'itérateur qui permettent le parcours inversé d'une séquence. Appliquer l'opérateur `++` sur un `reverse_iterator<`*`RandomAccessIterator`*`>` (ou un `reverse`

`_bidirectional_iterator<BidirectionalIterator>` revient au même qu'appliquer l'opérateur - sur un *RandomAccessIterator* (ou un *BidirectionalIterator*). Les deux versions diffèrent uniquement sur le concept d'itérateur sur lequel elles s'appliquent.

```
template <class T>
void afficher_en_avant(const std::list<T>& L)
{
    std::list<T>::iterator debut = L.begin();
    std::list<T>::iterator fin = L.end();
    while (debut != fin)
        std::cout << *debut << std::endl;
}
template <class T>
void afficher_en_arriere(const std::list<T>& L)
{
    typedef std::reverse_bidirectional_iterator<
            std::list<T>::iterator,
            T,
            std::list<T>::reference_type,
            std::list<T>::difference_type>
        reverse_iterator;
    reverse_iterator rdebut( L.begin() );
    reverse_iterator rfin( L.end() );
    while (rdebut != rfin)
        std::cout << *rdebut << std::endl;
}
```

Dans la fonction `afficher_en_avant()`, les éléments sont affichés dans l'ordre : `*debut`, `*(debut+1)`, …, `*(fin-1)`. Dans la fonction `afficher_en_arriere()` les éléments sont affichés du dernier au premier : `*(fin-1)`, `*(fin-1)`, …, `*debut`.

Si l'on devait écrire la deuxième fonction sur un `std::vector`, il faudrait utiliser un `std::reverse_iterator`.

Utiliser les itérateurs d'insertion

```
#include <iterator>
class insert_iterator<Conteneur>;
insert_iterator<Conteneur> inserter(Conteneur& c,
➥ Conteneur::iterator i);
```

`insert_iterator<>` est un adaptateur d'itérateur qui fonctionne comme un *OutputIterator :* une opération d'affectation sur un `insert_iterator<>` insère un objet dans une séquence.

L'exemple suivant insère quelques éléments dans une liste :

```
std::list<int> L;
L.push_front(3);
std::insert_iterator< std::list<int> > ii(L, L.begin());
*ii++ = 0;
*ii++ = 1;
*ii++ = 2;
std::copy(L.begin(), L.end(), std::ostream_iterator
➥ <int>(std::cout, " "));
// Affiche : 0 1 2 3
```

L'exemple ci-après fusionne deux listes triées en insérant le résultat dans un `set`. Notez qu'un `set` (ou ensemble) ne contient jamais deux éléments identiques (doublons).

```
const int N = 6;
int A1[N] = { 1, 3, 5, 7, 9, 11 };
int A2[N] = { 1, 2, 3, 4, 5, 6 };
std::set<int> resultat;
std::merge(A1, A1+N, A2, A2+N,
          ➥ std::inserer(resultat, resultat.begin()));
std::copy(L.begin(), L.end(), std::ostream_iterator
➥ <int>(std::cout, " "));
// Affiche : 1 2 3 4 5 6 7 9 11
```

Utiliser les itérateurs d'insertion en début de conteneur

```
#include <iterator>
class front_insert_iterator<FrontInsertionSequence>;
front_insert_iterator<FrontInsertionSequence>
front_inserer(FrontInsertionSequence & c);
```

`front_insert_iterator<>` est un adaptateur d'itérateur qui fonctionne comme un *OutputIterator :* une opération d'affectation sur un `front_insert_iterator<>` insère un objet juste avant le premier élément d'une séquence. Le conteneur doit supporter l'insertion d'un élément au début par la fonction membre `push_front()`. Ce conteneur doit donc supporter le concept de *FrontInsertionSequence*.

```
std::list<int> L;
L.push_front(3);
std::front_insert_iterator< std::list<int> > fii(L);
*fii++ = 0;
*fii++ = 1;
*fii++ = 2;
std::copy(L.begin(), L.end(), std::ostream_iterator
➥ <int>(std::cout, " "));
// Affiche : 2 1 0 3
```

Utiliser les itérateurs d'insertion en fin de conteneur

```
#include <iterator>
class back_insert_iterator<BackInsertionSequence>;
back_insert_iterator<BackInsertionSequence>
➥ front_inserter(BackInsertionSequence & c);
```

`back_insert_iterator<>` est un adaptateur d'itérateur qui fonctionne comme un *OutputIterator :* une opération d'affectation sur un `back_insert_iterator<>` insère un objet juste après le dernier élément d'une séquence. Le conteneur doit supporter l'insertion d'un élément à la fin par la fonction membre `push_back()`. Ce conteneur doit donc supporter le concept de *BackInsertionSequence*.

```
std::list<int> L;
L.push_front(3);
std::back_insert_ierator< std::list<int> > bii(L);
*bii++ = 0;
*bii++ = 1;
*bii++ = 2;
std::copy(L.begin(), L.end(), std::ostream_iterator
➥ <int>(std::cout, " "));
// Affiche : 3 0 1 2
```

9

Conteneurs standard

La quasi-totalité des compilateurs modernes fournit une implémentation de la STL (*Standard Template Library*, bibliothèque standard fournie avec tous les compilateurs C++ modernes). Ce chapitre en récapitule les principales fonctions.

Les conteneurs standard fournis par la STL sont articulés autour de quelques grandes familles, derrière lesquelles se retrouvent les itérateurs vus au chapitre précédent. Pour chacune de ces familles, on retrouve, en bonne partie, les mêmes fonctions membres. Le tableau suivant donne la liste des conteneurs STL regroupés par famille.

Les chaînes de caractères seront traitées à part, au chapitre suivant.

Conteneurs STL par famille

Séquentiel	Associatif	Adaptateur
`basic_string`	`map`	`priority_queue`
`string`	`multimap`	`queue`
`wstring`	`set`	`stack`
`deque`	`multiset`	`bitset`
`list`		
`vector`		

Créer un conteneur

```
#include <header_du_conteneur>
Conteneur();
Conteneur(const allocator_type& allocateur);
Conteneur(const Conteneur&);
Conteneur(size_type n, const value_type& valeur);
Conteneur(size_type n, const value_type& valeur,
➥ const allocator_type& allocateur);
Conteneur(InputIterator deb, InputIterator fin);
Conteneur(InputIterator deb, InputIterator fin,
➥ const allocator_type& allocateur);
// pour les conteneurs associatifs
Conteneur(const _Compare& comparateur_de_cles);
Conteneur(const _Compare& comparateur_de_cles,
➥ const allocator_type& allocateur);
Conteneur(InputIterator deb, InputIterator fin,
➥ const _Compare& comparateur_de_cles);
Conteneur(InputIterator deb, InputIterator fin,
➥ const _Compare& comparateur_de_cles, const
➥ allocator_type& allocateur);
```

Créer un conteneur se fait toujours de la même manière. Vous pouvez créer un conteneur vide (constructeur par défaut) ou pré-initialisé avec `n` `valeur` (`valeur` par défaut ou donnée). Il est également possible de créer un conteneur en copiant le contenu d'un autre, totalement ou partiellement à travers les itérateurs.

Si un `allocateur` est fourni, il sera copié dans le conteneur. Sinon, un allocateur par défaut sera instancié et utilisé. Pour les conteneurs associatifs, des constructeurs supplémentaires permettent de spécifier un foncteur utilisé pour comparer les clés (les conteneurs associatifs sont par essence des conteneurs implicitement triés).

Par exemple, le code suivant crée un vecteur de cinq entiers de valeur 42 :

```
std::vector<int> v(5,42);
```

Choisir son conteneur séquentiel

Dans un cadre général, utiliser `std::vector<>` est préférable à `std::deque<>` et `std::list<>`. Un `std::deque<>` est utile si vous insérez souvent en début ou fin de séquence. Les `std::list<>` et `std::slist<>` sont plus performants lorsque vous insérez le plus souvent en milieu de séquence. Dans la plupart des autres situations, un `std::vector<>` sera le plus efficace.

Ajouter et supprimer dans un conteneur séquentiel

```
conteneur.insert(iter_pos, valeur);
conteneur.insert(iter_pos, n, valeur);
conteneur.insert(iter_pos, iter_debut, iter_fin);
conteneur.push_back(valeur);
conteneur.push_front(valeur);
```

```
conteneur.erase(iter_pos);
conteneur.erase(iter_debut, iter_fin);
conenteur.clear();
```

L'insertion se fait toujours *juste avant* la position indiquée par l'itérateur. Lorsqu'une quantité `n` est fournie en plus d'une valeur, cette valeur est insérée `n` fois par copie. Si une séquence [`iter_debut`, `iter_fin`[est fournie, celle-ci sera copiée et insérée. Là encore, elle sera insérée juste avant la position `iter_pos`.

Info

Un appel à `conteneur.insert(p,n,t)` est garanti d'être au moins aussi rapide que d'appeler `n` fois `conteneur.insert(p,t)`. Dans certains cas, il peut même se révéler être beaucoup plus rapide.

`push_back()` et `push_front()` insèrent, respectivement, la valeur donnée en fin ou en début de conteneur. Leur pendant pour la suppression, `pop_back()` et `pop_front()` sont disponibles suivant les conteneurs.

`erase()` supprime l'élément à la position `iter_pos` donnée, ou toute la sous-séquence [`iter_debut`, `iter_fin`[donnée. Il va de soi que l'itérateur ou la séquence doivent être valides et donc, par voie de conséquence, pointer sur des éléments du conteneur utilisé.

`clear()` efface toutes les valeurs du conteneur et le rend vide. Notez toutefois que les structures internes du conteneur ne sont pas forcément libérées et peuvent toujours occuper de l'espace mémoire. Ce choix provient d'un souci d'efficacité à l'exécution (les opérations d'allocation et désallocation mémoire sont lentes par nature). Si vous êtes dans un contexte ou l'espace mémoire est plus important, prenez bien garde à ce phénomène.

Parcourir un conteneur

```
conteneur.begin();
conteneur.end();
conteneur.rbegin();
conteneur.rend();
```

La STL unifie la manière de coder. Cet avantage se retrouve dans la manière de parcourir tous les éléments

d'un conteneur. Ainsi, que vous ayez à faire à un tableau (que l'on peut aussi parcourir par indice), à une liste ou un arbre (à travers les `std::map<>` ou les `std::set<>`) il est possible de procéder toujours de la même manière, grâce à la notion d'itérateur.

Vous trouverez dans le code ci-après tous les exemples de parcours de conteneur : à l'endroit ou à l'envers. Si vous parcourez des conteneurs passés en arguments constants, il vous sera nécessaire d'utiliser des itérateurs dits constants.

```
std::Conteneur<…>::iterator it;
for (it = conteneur.begin(); it != conteneur.end(); ++it)
{
    // …
}
std::Conteneur<…>::const_iterator it;
for (it = conteneur.begin(); it != conteneur.end(); ++it)
{
    // …
}

std::Conteneur<…>::reverse_iterator it;
for (it = conteneur.rbegin(); it != conteneur.rend();
➥ ++it)
{
    // …
}
std::Conteneur<…>::const_reverse_iterator it;
for (it = conteneur.rbegin(); it != conteneur.rend();
➥ ++it)
{
    // …
}
```

Lorsque vous parcourez un conteneur associatif, vous rencontrez les éléments selon l'ordre croissant (suivant la relation d'ordre fournie au type de conteneur) de leur clé.

Dans l'exemple ci-après, nous remplissons un `std::set<int>` avec quelques valeurs dans un ordre quelconque et constatons que lors du parcours, nous les retrouvons dans l'ordre croissant.

```
std::set<int> S;
S.insert(7);
S.insert(5);
S.insert(9);
S.insert(1);
S.insert(3);
for (std::set<int>::const_iterator it = S.begin();
     it != S.end(); ++it)
{
    std::cout << *it << " " ;
}
std::cout << std::endl;
// Affiche: 1 3 5 7 9
```

Accéder à un élément d'un conteneur

```
conteneur.at(i);
conteneur[i];
conteneur.back();
conteneur.front();
conteneur.count(cle);
conteneur.find(cle);
```

Ces fonctions ne sont pas valables pour tous les conteneurs ; leur existence dépend du type de conteneur. Le tableau suivant précise la disponibilité de chacune d'elle pour chaque conteneur.

Disponibilité des fonctions par conteneur

Fonction	Conteneurs
`at(i)`	`std::basic_string<>`, `std::deque<>`, `std::vector<>`
`conteneur[]`	`std::basic_string<>`, `std::deque<>`, `std::vector<>` `std::map<>`, `std::multimap<>`, `std::set<>`, `std::multiset<>`[1]
`back()`	`std::deque<>`, `std::list<>`, `std::vector<>`
`front()`	`std::deque<>`, `std::list<>`, `std::vector<>`
`count(cle)`	`std::map<>`, `std::multimap<>`, `std::set<>`, `std::multiset<>`
`find(cle)`	`std::map<>`, `std::multimap<>`, `std::set<>`, `std::multiset<>`

1. Attention, en plus d'accéder à l'élément, celui-ci sera créé s'il n'existe pas. Si vous ne voulez pas créer une entrée, utilisez `find(cle)` plutôt que `conteneur[cle]` pour les conteneurs associatifs.

La fonction `count()` est un moyen simple de tester si une clé est présente dans un conteneur associatif. Toutefois, si vous souhaitez tester la présence de la clé puis éventuellement utiliser la valeur associée, utilisez plutôt un `find()`, comme l'illustre le code suivant :

```
std::map<int,int> M;
// …
std::map<int,int>::iterator it = M.find(3);
if (it != M.end())
{
   std::cout << "3 existe et est associé à " << it->second
   << std::endl
   << "La valeur de la clé est bien " << it->first
   << std::endl;
}
else
    std::cout << "La clé 3 n'est pas présente dans M.\n";
```

Créer et utiliser un tableau

```
#include <vector>
std::vector<Type> V;
```

Les vecteurs de la STL sont des vecteurs dynamiques (leur taille peut changer au cours de la vie du programme). Ils font parti des conteneurs dits séquentiels. De fait, l'insertion et la suppression d'éléments à la fin d'un vecteur s'effectuent en temps quasi constant. En début ou en tout autre endroit qu'à la fin, le temps de ces opérations sera proportionnel au nombre d'éléments présents après le lieu d'insertion. Le type d'itérateur associé aux vecteurs est de type *RandomAccesIterator*, autrement dit à accès direct.

Attention

La bibliothèque STL spécialise le type `std::vector<bool>`. Celui-ci permet de stocker l'information au niveau du bit par l'intermédiaire de masques. Si cela à l'avantage d'utiliser très peu de mémoire, il en coûte un très fort impact sur les performances. Si les performances sont primordiales pour vous, préférez-lui un `std::vector<unsigned char>`.

Le dernier constructeur crée un vecteur contenant une copie des éléments entre `debut` et `fin`. Considérez l'exemple suivant :

```
// Crée un vecteur d'entiers aléatoires
std::vector<int> v(10) ;
std::cout << "vecteur original : ";
for (int i=0; i<v.size(); i++)
    std::cout << (v[i] = (int) rand() % 10) << " ";
std::cout << std::endl;
```

```
// trouver le premier élément pair du tableau
std::vector<int>::iterator it1 = v.begin();
while (it1!=v.end() && *it1%2 !=0)
    it1++;
// trouver le dernier élément pair du tableau
std::vector<int>::iterator it2 = v.end();
do { it2--; } while (it1!=v.begin() && *it1%2 !=0) ;
// si au moins un élément trouvé
if (it1 != v.end())
{
    std::cout << "premier nombre pair : " << *it1
              << ", dernier nombre pair : " << *it2
              << std::endl
              << "nouveau vecteur : " ;
    std::vector<int> v2(it1, it2);
    for (int i=0; i<v2.size(); i++)
       std::cout << v2[i] << " ";
    std::cout << std::endl;
}
```

Son exécution produira la sortie suivante :

```
vecteur original : 1 7 8 3 4 2 0 8 6 5
premier nombre pair : 8, dernier nombre pair : 6
nouveau vecteur : 8 3 4 2 0 8 6
```

Attention

Les itérateurs sur les vecteurs ne sont généralement pas stables à l'insertion ou à la suppression d'éléments. En effet, les éléments étant stockés de manière contiguë, ceux-ci peuvent physiquement changer de place en mémoire et les itérateurs devenir invalides. Même à l'insertion en fin de vecteur : cette opération peut nécessiter une augmentation de la taille réservée impliquant le déplacement de tous les éléments.

Info

Pour de petites séquences, les vecteurs sont plus performants que les listes. Si donc vous utilisez de petites séquences et que la stabilité des itérateurs n'est pas importante pour vous, préférez-les aux listes.

Créer et utiliser une liste chaînée

```
#include <list>
std::list<Type> L;
```

Les listes STL sont des listes doublement chaînées. La plupart des implémentations utilisent un chaînon sans donnée comme tête/queue de liste. Un chaînon correspond à un itérateur. Le chaînon de tête se retrouve à travers un appel de la fonction membre `end()`. Les itérateurs des listes sont garantis comme stables à l'insertion et à la suppression (pour autant que ce ne soit pas celui que vous supprimiez…).

Attention

Dans certaines implémentations de la STL, un itérateur de liste est plus qu'un pointeur sur un maillon, et contient aussi d'autres informations. C'est par exemple le cas avec l'implémentation de Microsoft Visual C++ 8, où il peut atteindre la taille de trois pointeurs en mode debug (celui sur le maillon plus un sur la liste et un autre sur le conteneur, mais qu'importe puisque leur implémentation a encore changé avec VC++ 9).

Le parcours d'une liste chaînée s'effectue grâce aux itérateurs :

```
std::list<int> L;
// …
for (std::list<int>::iterator it = L.begin(); it !=
➥ L.end(); ++it)
{
    // … accès à l'élément
    // avec (*it).methode(…) ou it->methode(…)
}
```

Vous pouvez facilement parcourir une liste à l'envers à l'aide des itérateurs inversés :

```
std::list<int> L;
// …
for (std::list<int>::reverse_iterator it = L.rbegin();
it != L.rend(); ++it)
{
    // … accès à l'élément
    // avec (*it).methode(…) ou it->methode(…)
}
```

Si vous êtes dans une fonction où la liste que vous utilisez est fournie en référence constante, il vous faudra utiliser des itérateurs constants (par exemple, `std::list<int>::const_iterator`).

Créer et utiliser une file à double entrée

```
#include <deque>
std::deque<Type> D;
```

Les files à doubles entrées sont un peu comme des piles dans lesquelles il est possible d'ajouter et de retirer un élément

au début (avec `push_front()` et `pop_front()`) ou à la fin (avec `push_back()` et `pop_back()`). L'accès au début et à la fin se fait avec `front()` et `back()`. De plus, l'accès par indice demeure possible et reste en temps constant à travers l'opérateur `[]` et la fonction `at()` qui fournit en plus un contrôle des bornes.

La stabilité des itérateurs que vous auriez stockés n'est pas garantie, sauf en cas de suppression en tête ou en queue de file.

Astuce

Les files à double entrée permettent de coder facilement des conteneurs *First In First Out* (FIFO), ou « premier entré, premier sorti ». Pour ce faire, il suffit d'utiliser `push_front()` et `pop_pack()` conjointement. Le code suivant pourrait être un moyen très simple de la simuler à l'aide d'une `std::deque<>` :

```
template <class T>
struct FIFO
{
   inline T   pop()  { return m_values.pop_back(); }
   inline void push(const T& v) { m_values.push_front(v); }
protected:
   std::deque<T> m_values;
};
```

Mais ne vous donnez pas cette peine ! Pour les conteneurs FIFO, la STL fournie les `std::queue<>`. Et pour les conteneurs LIFO, vous disposez des `std::stack<>` (piles).

Créer et utiliser une pile

```
#include <stack>
std::stack<Type,Conteneur> S;
std::stack<Type> S;
```

Une pile supporte l'insertion et la suppression d'éléments à une seule extrémité à travers `push()` et `pop()`. Du coup, on retrouve la règle du « dernier inséré, premier sorti », typique d'un conteneur LIFO (*Last In First Out*). L'accès au dernier élément empilé se fait avec `top()`.

Vous pouvez définir une pile avec n'importe quel conteneur séquentiel comme `std::deque<>`, `std::list<>` ou `std::vector<>`. Si vous ne spécifiez pas le type de conteneur à utiliser, un `std::deque<>` le sera par défaut.

Attention

Le constructeur d'une pile prenant pour argument un conteneur (dont le type doit correspondre au type de conteneur spécifié dans l'instanciation) *n'effectue pas* un lien vers ce conteneur pour l'utiliser mais copie bel et bien tous ses éléments au sein de la pile ainsi créée.

```
std::deque<int> D;
D.push_back(1);
D.push_back(2);
D.push_back(3);
std::stack<int> S(D);
// S contient 1 2 3
D[1] = 20;
// D contient 1 20 3
// S contient toujours 1 2 3
```

Créer et utiliser une queue

```
#include <queue>
std::queue<Type,Conteneur> Q;
std::queue<Type> Q;
```

Une queue supporte l'insertion d'éléments en début de conteneur (avec `push()`) et la suppression d'éléments en fin de conteneur (avec `pop()`). On retrouve la règle du « premier inséré, premier sorti » typique d'un conteneur FIFO (*First In First Out*). L'accès au dernier élément empilé se fait avec `front()` et l'accès au prochain élément dépilé avec `back()`.

Vous pouvez définir une queue avec n'importe quel conteneur séquentiel comme `std::deque<>`, `std::list<>` ou `std::vector<>`. Si vous ne spécifiez pas le type de conteneur à utiliser, un `std::deque<>` le sera par défaut.

Pour déterminer si une queue est vide, utilisez `empty()`. Pour connaître le nombre d'éléments qu'elle contient, utilisez `size()`.

Info

Les piles, queues et queues de priorités sont des adaptateurs, basés sur des conteneurs de la STL. Si vous voulez créer vos propres conteneurs en les basant sur des conteneurs (ou adaptateurs) de la STL, n'hésitez pas à regarder comment ces classes sont écrites. Vous pourrez glaner ainsi beaucoup de techniques et idées utiles pour maximiser leur compatibilité avec la STL et pérenniser votre code.

Créer et utiliser une queue de priorité

```
#include <priority_queue>
std::priority_queue<Type,Conteneur,RelationDOrdre> PQ;
std::priority_queue<Type,Conteneur> PQ;
std::priority_queue<Type> PQ;
```

Une queue de priorité est une queue un peu particulière : elle classe ses éléments de telle sorte que le prochain élément à dépiler corresponde à l'élément le plus prioritaire. Un élément est défini comme plus prioritaire s'il se trouve en début de liste après ordonnancement suivant la relation d'ordre fournie (la relation d'ordre par défaut est `std::less<Type>`).

Le conteneur utilisé par défaut est un `std::vector<>`. Vous pouvez utiliser n'importe quel conteneur supportant l'accès direct (c'est-à-dire par indice). En effet, pour classer ses éléments, un tel conteneur utilise les algorithmes `std::make_heap()`, `std::push_heap()` et `std::pop_heap()` qui reposent sur ce concept.

L'insertion se fait avec `push()`, et la suppression ou dépilement avec `pop()`. L'interrogation de l'élément le plus prioritaire se fait avec `top()`. Comme d'habitude, `size()` retourne le nombre d'éléments présents et `empty()`indique si le conteneur est vide.

Attention

`pop()` ne renvoie pas l'élément mais se contente de supprimer l'élément le plus prioritaire du conteneur.

Créer et utiliser un ensemble

```
#include <set>
std::set<Type, RelationDOrdre> S;
std::set<Type> S;
std::multiset<Type, RelationDOrdre> MS;
std::multiset<Type> MS;
```

Les ensembles, simples ou multiples, sont des conteneurs associatifs simples : ils n'ont pas de valeur associée à chaque clé, la clé elle-même faisant office de valeur. Ce sont des conteneurs triés. Un ensemble simple `std::set<>` ne peut pas contenir deux éléments identiques, alors qu'un ensemble multiple `std::multiset<>` l'autorise.

Les itérateurs sur les ensembles sont stables à l'insertion et la suppression d'éléments, à l'exception (bien sûr) de ceux pointant sur des éléments retirés de l'ensemble considéré.

Pour insérer des éléments, utilisez `insert(elt)`. Cette fonction renvoie une `std::pair<iterator,bool>` contenant l'emplacement de la valeur insérée si `second` vaut vrai, ou de l'emplacement de la valeur déjà existante si `second` vaut faux. Si vous connaissez déjà l'endroit où insérer la valeur, utilisez plutôt `insert(iter_pos,valeur)` qui garantit dans ce cas une insertion en temps constant.

Créer et utiliser une table associative

```
#include <map>
std::map<Cle, Type, RelationDOrdre> M;
std::map<Cle, Type> M;
std::multimap<Cle, Type, RelationDOrdre> M;
std::multimap<Cle, Type> M;
```

Les tables associatives permettent d'associer à une clé une (pour les `std::map<>`) ou plusieurs valeurs (pour les `std::multimap<>`) d'un `Type` donné. La relation d'ordre utilisée par défaut est le foncteur `std::less<Cle>`, qui utilise l'opérateur `<` par défaut. Pour chercher un élément sans créer d'entrée dans le conteneur, n'utilisez pas l'opérateur `[]` mais plutôt la fonction membre `find()` qui renvoie un itérateur sur l'élément trouvé ou l'itérateur `end()` sinon. L'exemple suivant montre comment obtenir toutes les valeurs associées à une clé dans un `std::multimap<>`.

```
typedef std::multimap<TypeCle,TVal>::const_iterator
➥ ConstIter;
std::pair<ConstIter,ConstIter> bound = mm.equal_range
➥ (ma_cle);
for (ConstIter it = bound.first; it != bound.second; ++it)
    // …
```

L'exemple d'utilisation suivant illustre comment retrouver le nombre de jours dans un mois en fonction du nom du mois :

```
std::map<std::string, int> mois;
std::map<std::string, int>::iterator it_cour, it_avant,
➥ it_apres;
mois[std::string("janvier")]  = 31;
mois[std::string("février")]  = 28;
mois[std::string("mars")]     = 31;
// …
mois[std::string("décembre")] = 31;

std::cout << "mars : " << mois[std::string("mars")]
➥ << std::endl;
it_cour  = mois.find("june");
it_avant = it_apres = it_cour;
++it_apres;
--it_avant;
std::cout << "Avant (en ordre alphabétique) : "
          << it_avant->first << std::endl;
std::cout << "Après (en ordre alphabétique) : "
          << it_apres->first << std::endl;
```

L'insertion dans les `std::multimap<>` est un peu moins intuitive. L'exemple suivant montre comment utiliser ce conteneur :

```
struct ltstr
{
  bool operator()(const char* s1, const char* s2)
const
  {
    return strcmp(s1, s2) < 0;
  }
};
//…
std::multimap<const char*, int, ltstr> m;
std::multimap<const char*, int, ltstr>::iterator it;
```

```
m.insert(std::pair<const char* const, int>("a", 1));
m.insert(std::pair<const char* const, int>("c", 2));
m.insert(std::pair<const char* const, int>("b", 3));
m.insert(std::pair<const char* const, int>("b", 4));
m.insert(std::pair<const char* const, int>("a", 5));
m.insert(std::pair<const char* const, int>("b", 6));

std::cout << "Nombre d'éléments avec 'a' pour cle :
➥ " << m.count("a") << std::endl;
std::cout << "Nombre d'éléments avec 'b' pour cle :
➥ " << m.count("c") << std::endl;
std::cout << "Nombre d'éléments avec 'c' pour cle :
➥ " << m.count("c") << std::endl;

std::cout << "Nombre d'éléments dans m : " << std::
endl;
for (it = m.begin(); it != m.end(); ++it)
{

std::cout << "  <" << it->first << ", " << it->second
➥ << ">" << std::endl;
}
```

Info

Les `std::map<>` et `std::multimap<>` sont généralement basées sur des arbres binaires équilibrés. L'insertion et la recherche d'éléments s'effectue donc en temps *O(log n)*. Si cela est suffisant dans bien des cas, ce peut être pénalisant dans d'autres. Si donc vous avez besoin de meilleures performances, vous pouvez vous tourner, au prix d'une occupation mémoire plus importante, vers d'autres solutions. Avant de développer les vôtres, essayez les tables de hachage fournies par la STL. Leur temps d'accès et d'insertion est en temps quasi constant, comme cela est expliqué à la section suivante.

Créer et utiliser une table de hachage

```
#include <hash_set>
std::hash_set<Cle[,FonctionDeHash[,TestEgaliteCle]]>
➥ M;
std::hash_multiset<Cle[,FonctionDeHash[,
➥ TestEgaliteCle]]> M;
```

```
#include <hash_map>
std::hash_map<Cle,Type[,FonctionDeHash[,
➥ TestEgaliteCle]]> M;
std::hash_multimap<Cle,Type[,FonctionDeHash[,
➥ TestEgaliteCle]]> M;
```

Attention

Avec g++, vous trouverez ces classes dans `<ext/hash_set>` et `<ext/hash_map>`.

Les tables de hachage sont plus performantes que les conteneurs associatifs (*map* et *set)* mais n'ordonnent pas les éléments en fonction de leur clé. Elle repose sur le calcul d'une sous-clé de type entier pour donner un indice dans un tableau contenant (suivant le type de table de hachage utilisé) des *maps* ou des *sets* multiples ou non.

Le foncteur de test d'égalité des clés par défaut est `std::equal_to<Cle>`. La fonction de hachage par défaut est le foncteur `std::hash<Cle>`.

L'utilisation des tables de hachage est comparable à celle des tables associatives et des ensembles suivant le cas.

Les fonctions de hachage par défaut supportent les types suivants : `char*`, `const char*`, `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long` et `unsigned long`, plus `std::crope<>` et `std::wrope<>` pour une gestion efficace au niveau des chaînes de caractères. Si vous voulez écrire vos propres fonctions de hachage pour vos types personnalisés, sachez que ce foncteur doit impérativement retourner un `std::size_t`, comme le montre le modèle suivant :

```
struct MonHash
{
    std::size_t operateor()(const MaCle& cle)
    {
        // calcul
    }
};
```

Connaître la complexité des fonctions membres des conteneurs

Comparaison des complexités

Groupe	Fonction	vector	deque	list	set	multiset
	Constructeur	⋆	⋆	⋆	⋆	⋆
	Destructeur	O(n)	O(n)	O(n)	O(n)	O(n)
	operateur=	O(n)	O(n)	O(n)	O(n)	O(n)
Itérateurs	`begin`	O(1)	O(1)	O(1)	O(1)	O(1)
	`end`	O(1)	O(1)	O(1)	O(1)	O(1)
	`rbegin`	O(1)	O(1)	O(1)	O(1)	O(1)
	`rend`	O(1)	O(1)	O(1)	O(1)	O(1)
Capacité	`size`	O(1)	O(1)	O(1)[1]	O(1)	O(1)
	`max_size`	O(1)	O(1)	O(1)[1]	O(1)	O(1)
	`empty`	O(1)	O(1)	O(1)	O(1)	O(1)
	`resize`	O(n)	O(n)	O(n)	–	–
Accès	`front`	O(1)	O(1)	O(1)	–	–
	`back`	O(1)	O(1)	O(1)	–	–
	`top`	–	–	–	–	–
	`operator[]`	O(1)	O(1)	–	–	–
	`at`	O(1)	O(1)	–	–	–
Modifieurs	`assign`	O(n)	O(n)	O(n)	–	–
	`insert`	O(n+m)	O(m)[2]	O(m)	Log[3]	Log[3]

map	multimap	bitset	stack	queue	priority_queue
⋆	⋆	⋆	⋆	⋆	⋆
O(n)	O(n)	–	–	–	–
O(n)	O(n)	O(1)[4]	–	–	–
O(1)	O(1)	–	–	–	–
O(1)	O(1)	–	–	–	–
O(1)	O(1)	–	–	–	–
O(1)	O(1)	–	–	–	–
O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
O(1)	O(1)	–	–	–	–
O(1)	O(1)	–	O(1)	O(1)	O(1)
–	–	–	–	–	–
–	–	–	–	O(1)	–
–	–	–	–	O(1)	–
–	–	–	O(1)	–	O(1)
Log	–	O(1)[4]	–	–	–
–	–	–	–	–	–
–	–	–	–	–	–
Log[3]	Log[3]	–	–	–	–

Comparaison des complexités *(Suite)*

Groupe	Fonction	vector	deque	list	set	multiset
	`erase`	O(n)[5]	O(m)[6]	O(m)	O(1)+[7]	O(1)+[7]
	`swap`	O(1)	O(1)	O(1)	O(1)	O(1)
	`clear`	O(n)	O(n)	O(n)	O(n)	O(n)
	`push_front`	–	O(1)	O(1)	–	–
	`pop_front`	–	O(1)	O(1)	–	–
	`push_back`	O(1)	O(1)	O(1)	–	–
	`pop_back`	O(1)	O(1)	O(1)	–	–
	`push`	–	–	–	–	–
	`pop`	–	–	–	–	–
Observeurs	`key_comp`	–	–	–	O(1)	O(1)
	`value_comp`	–	–	–	O(1)	O(1)
Opérations	`find`	–	–	–	Log	Log
	`count`	–	–	–	Log	Log
	`lower_bound`	–	–	–	Log	Log
	`upper_bound`	–	–	–	Log	Log
	`equal_range`	–	–	–	Log	Log

1. En *O(n)* dans certaines implémentations.
2. En *O(n+m)* dans certaines implémentations (*m* est le nombre d'éléments à insérer).
3. `insert(x)` est logarithmique (*O(log n)*) ; `insert(position)` est en général logarithmique (*O(log n)*), mais peut être en temps constant amorti (*O(1)+*) si `x` est inséré juste après l'élément pointé par `position` ; `insert(first, last)` est généralement en *m* x *log(n+m)*, où *m* est le nombre d'éléments à insérer et *n* la taille du conteneur avant insertion, mais linéaire (*O(m)*) si les éléments insérés sont triés avec le même critère que le conteneur.

map	multimap	bitset	stack	queue	priority_queue
O(1)+[7]	O(1)+[7]	–	–	–	–
O(1)	O(1)	–	–	–	–
O(n)	O(n)	–	–	–	–
–	–	–	–	–	–
–	–	–	–	–	–
–	–	–	–	–	–
–	–	–	–	–	–
–	–	–	O(1)	O(1)	O(1)
–	–	–	O(1)	O(1)	O(1)
O(1)	O(1)	–	–	–	–
O(1)	O(1)	–	–	–	–
Log	Log	–	–	–	–
Log	Log	–	–	–	–
Log	Log	–	–	–	–
Log	Log	–	–	–	–
Log	Log	–	–	–	–

4. Avec un surcoût non négligeable dû au calcul de décalage et de masque.
5. Linéaire en fonction du nombre d'éléments à effacer (*O(m)*), plus le nombre d'éléments à déplacer après le dernier élément effacé (*O(m+n – pos)*).
6. Linéaire en fonction du nombre d'éléments effacés (*O(m)*). Dans certaines implémentations, ajoute également un temps linéaire au nombre d'éléments restant après les éléments supprimés (*O(m+n-pos)*).
7. `erase(position)` est en temps constant amorti (*O(1)+*) ; `erase(x)` est en *O(log n)* ; `erase(first,last)` est en *O(log n) + O(m)*.

Fonctions spécifiques

Conteneur	Fonctions
`vector`	`capacity`, `reserve`, `splice`
`list`	`splice`, `reserve`, `remove`, `remove_if`, `unique`, `merge`, `sort`
`bitset`	`set`, `reset`, `flip`, `to_ulong`, `to_string`, `test`, `any`, `none`

10

Chaînes de caractères

Les chaînes de caractères sont utilisées dans pratiquement tous les programmes informatiques, du moins la quasi-totalité de ceux communiquant avec l'homme. Manipuler de simples pointeurs sur des `char` (ou `wchar_t`) est non seulement risqué (débordement de mémoire) mais devient totalement caduc lorsque l'on désire rendre son programme polyglotte. Pour ce faire, la STL fournit différent types de chaînes de caractères s'intégrant parfaitement avec les algorithmes de la STL (le contraire aurait été surprenant) mais surtout facilitant la vie du programmeur.

D'autres implémentations existent comme les `QString` de QT ou les `wxString` de wxWidgets. Ils ont parfois des avantages par rapport à ceux de la STL (ne serait-ce que leur intégration au sein de leur bibliothèque respective) mais aussi leurs inconvénients. Il faut donc opter pour l'un ou l'autre en fonction de vos besoins. Il n'est pas rare de les voir cohabiter dans des programmes d'une certaine envergure.

Créer une chaîne

```
#include <string>
std::string une_chaine;
```

```
#include <wstring>
std::wstring une_chaine_unicode;
```

Les chaînes de caractères de la STL ne sont pas forcément des chaînes AZT (à zéro terminal). Elles dépassent cette limite en embarquant simplement le nombre de caractères. On obtient ainsi directement la taille de ces chaînes avec la fonction membre `size()`. Notez qu'il s'agit bien ici du nombre de caractères et non du nombre d'octets. Les `std::string<>` stockent des chaînes dont le jeu de caractères s'étend sur 8 bits maximum, et les `std::wstring<>` sont dédiées aux chaînes utilisant un jeu de caractères étendus comme Unicode.

Info

Si vous manipulez de très grosses chaînes de caractères, examinez les `std::crope<>` et `std::wrope<>`.

```
#include <rope> // <ext/rope> sous G++
std::crope c;
std::wrope w;
```

Elles sont stockées sous une forme évolutive conçue pour rendre efficace les traitements impliquant la chaîne dans son ensemble. Ainsi, des opérations comme l'affectation, la concaténation et l'extraction de sous-chaînes prendront un temps presque indépendant de la taille de la chaîne. Par contre, remplacer un caractère dans une *rope* est coûteux, de même que la parcourir caractère par caractère. C'est un peu le revers de la médaille (sinon quelle serait l'utilité des *string* ?).

Les constructeurs de chaînes de caractères supportent plusieurs formes d'initialisation. L'exemple suivant en dresse la liste et leurs éventuelles limitations :

```
std::string s1; // une chaîne vide
std::string s2("abc\x00def"); // contiendra "abc"
std::string s3 = "abc\x00def"; // contiendra "abc"
std::string s4("abc\x00def",6); // contiendra "abc\x00de"
std::string s5(5,'a'); // contiendra "aaaaa"
std::string s6(s5); // contiendra une copie de s5
std::string s7(s4,2,3); // contiendra "c\x00d"
std::set<char> aSet;
aSet.insert('e'); aSet.insert('a');aSet.insert('z');
std::string s8(aSet.begin(),aSet.end()); // contiendra "aez"
s1 = s3; // contiendra une copie de s3
```

Si vous avez besoin d'obtenir une conversion en chaîne AZT d'une `std::string`, utilisez la fonction membre `c_str()` qui ajoutera un caractère nul si il n'y était pas (sans modifier la taille de votre chaîne), avant de retourner un pointeur sur le premier caractère. Attention toutefois, si votre chaîne contient déjà en son sein un caractère nul, vous retrouverez les mêmes restrictions qu'avec les chaînes C classiques…

Astuce

Si vous avez besoin de réinitialiser une chaîne, utilisez les fonctions membres `assign()`. Elles prennent les mêmes arguments que les différents constructeurs mais offrent l'avantage d'éviter de créer un objet temporaire dans certains cas.

Connaître la longueur d'une chaîne

```
#include <string>
bool string::empty();
bool string::length();
bool string::size();
```

La fonction `empty()` indique si la chaîne est vide. Les fonctions `length()` et `size()` sont identiques et renvoient la longueur de la chaîne.

Comparer des chaînes

```
int string::compare(const string& s2);
int string::compare(const char* s2);
int string::compare(size_type index, size_type len,
➥ const string& s2);
int string::compare(size_type index, size_type len,
➥ const string& s2, size_type index_s2, size_type
➥ len_s2);
int string::compare(size_type index, size_type len,
➥ const char* s2, size_type len_s2);
```

La fonction membre `compare()` permet de comparer la chaîne considérée (`s1`) avec une autre (`s2`). Si le résultat est négatif, alors `s1 < s2`. Si le résultat est nul, alors `s1 == s2`. Enfin si le résultat est positif, alors `s1 > s2`. Notez que ces opérateurs de comparaison (<, > et ==) existent également, mais prenez garde à l'allocation dynamique si vous comparez une chaîne à un `char*`.

```
std::string noms[] = {
    "Alfred", "Robin", "5e élément", "inconnu"
};

for (int i=0; i<4; ++i)
{
    for (int j=0; j<4; ++j)
        std::cout << noms[i].compare( noms[j] ) << " ";
    std::cout << std::endl;
}
```

Échanger le contenu de deux chaînes

```
void string::swap(string& s2);
void swap(string& s1, string& s2);
```

Les fonctions `swap()` sont beaucoup plus performantes que d'utiliser une variable temporaire de type `std::string`. Elles effectuent un échange de la taille et du pointeur interne plutôt que de recopier toutes les chaînes, ce qui est évidemment beaucoup plus rapide.

```
std::string s1 = "Première chaîne";
std::string s2 = "Deuxième contenu";
s1.swap(s2);
std::cout << s1 << std::endl; // "Deuxième contenu"
std::cout << s2 << std::endl; // "Première chaîne"
```

Rechercher une sous-chaîne

```
size_type string::find(const string& str, size_type
➥ index);
size_type string::find(const char* str, size_type
➥ index);
size_type string::find(const char* str, size_type
➥ index, size_type length);
size_type string::find(char ch, size_type index);
```

```
size_type string::rfind(const string& str, size_type
➥ index);
size_type string::rfind(const char* str, size_type
➥ index);
size_type string::rfind(const char* str, size_type
➥ index, size_type num);
size_type string::rfind(char ch, size_type index);
```

```
size_type string::find_first_not_of(const string&
➥ str, size_type index = 0);
size_type string::find_first_not_of(const char* str,
➥ size_type index = 0 );
size_type string::find_first_not_of(const char* str,
➥ size_type index, size_type num);
size_type string::find_first_not_of(char ch,
➥ size_type index = 0);
```

```
size_type string::find_first_of(const string &str,
➥ size_type index = 0);
size_type string::find_first_of(const char* str,
➥ size_type index = 0);
size_type string::find_first_of(const char* str,
➥ size_type index, size_type num);
size_type string::find_first_of(char ch, size_type
➥ index = 0);
```

```
size_type string::find_last_not_of(const string& str,
➥ size_type index = npos);
size_type string::find_last_not_of(const char* str,
➥ size_type index = npos );
size_type string::find_last_not_of(const char* str,
➥ size_type index, size_type num);
size_type string::find_last_not_of(char ch, size_type
➥ index = npos);
```

```
size_type string::find_last_of(const string &str,
➥ size_type index = npos);
size_type string::find_last_of(const char* str,
➥ size_type index = npos);
size_type string::find_last_of(const char* str,
➥ size_type index, size_type num);
size_type string::find_last_of(char ch, size_type
➥ index = npos);
```

Dans tous les cas, ces fonctions renvoient soit l'indice de la position dans la chaîne considérée (celle qui correspond au `this` de la fonction membre), soit `string::npos` si rien n'est trouvé.

Les fonctions `find()` renvoient l'indice où se trouve la première occurrence de la chaîne `str` ou du caractère `ch` recherché. Lorsque le paramètre `length` est donné, la fonction ne recherchera que les `length` premiers caractères de `str`. Le paramètre `index` permet de préciser le point de départ de la recherche (ceci permet de trouver les occurrences suivantes).

Les fonctions `rfind()` effectuent la recherche en remontant vers le début de la chaîne en partant de l'`index` donné.

```
std::string str1( "Alpha Beta Gamma Delta" );
std::string::size_type loc = str1.find( "Omega", 0 );
if( loc != std::string::npos )
    std::cout << "Found Omega at " << loc << std::endl;
else
    std::cout << "Didn't find Omega" << std::endl;
```

Les fonctions `find_first_not_of()` recherchent le premier caractère n'appartenant pas à `str` (ou différent de `ch`). Là encore, il est possible de ne considérer que les `num` premiers caractères de `str`, et ne commencer la recherche qu'à l'`index` voulu.

Les fonctions `find_first_of()` recherchent le premier caractère appartenant à `str`. Les fonctions `find_last_not_of()` et `find_last_of()` sont comme ce que `rfind()` est à `find()`, mais recherchent n'importe quel caractère de la chaîne plutôt que la chaîne elle-même.

L'exemple ci-après recherche le premier caractère qui ne soit pas une minuscule. Il affichera la valeur `33`.

```
std::string minuscule = "abcdefghijklmnopqrstuvwxyz ,-";
std::string chaine = "ceci est la partie en minuscule,
➥ ET CELLE-CI EST LA PARTIE EN MASJUCULE";
std::cout << "la première lettre non minuscule dans la
➥ chaîne est à l'indice : " << chaine.find_first_not_of
➥ (minuscule) << std::endl;
```

Extraire une sous-chaîne

```
string string::substr(size_type index, size_type
➥ length = npos);
```

La fonction `substr()` retourne la sous-chaîne commençant à l'indice `index` et de taille `length`. Si ce dernier paramètre n'est pas fourni, la sous-chaîne contiendra tous les caractères suivant l'`index` donné, jusqu'à la fin.

Attention

La sous-chaîne renvoyée est une copie des caractères et n'est pas un lien vers une sous-partie de la chaîne d'origine.

Rien ne vaut un exemple pour comprendre immédiatement. En voici donc un très simple :

```
std::string s("What we have here is a failure to
➥ communicate");
std::string sub = s.substr(21);
std::cout << "La chaîne originale est : " << s << std::endl;
std::cout << "La sous-chaîne est : " << sub << std::endl;
```

Il produira le résultat suivant :

```
La chaîne originale est : What we have here is a
➥ failure to communicate
La sous-chaîne est : a failure to communicate
```

Remplacer une partie d'une chaîne

```
string& string::replace(size_type index, size_type
➥ num, const string& str);
string& string::replace(size_type index1, size_type
➥ num1, const string& str, size_type index2,
➥ size_type num2);
string& string::replace(size_type index, size_type
➥ num, const char* str);
string& string::replace(size_type index, size_type
➥ num1, const char* str, size_type num2);
string& string::replace(size_type index, size_type
➥ num1, size_type num2, char ch);
string& string::replace(iterator start, iterator
➥ end, const string& str);
string& string::replace(iterator start, iterator
➥ end, const char* str);
string& string::replace(iterator start, iterator
➥ end, const char* str, size_type num);
string& string::replace(iterator start, iterator
➥ end, size_type num, char ch);
```

Les fonctions `replace()` font, au choix, les actions suivantes :

- remplacer les caractères de la chaîne courante avec au plus `num` caractères de `str`, en commençant à `l'index` donné ;
- remplacer jusqu'à `num1` caractères de la chaîne courante (en commençant à l'`index1`) avec au plus `num2` caractères de `str` en partant de l'`index2` ;
- remplacer jusqu'à `num` caractères de la chaîne courante par ceux de `str`, en commençant à l'indice `index` ;
- remplacer jusqu'à `num1` caractères de la chaîne courante (en partant de l'indice `index1`) par les `num2` caractères de `str` à partir de l'indice `index2` ;
- remplacer jusqu'à `num1` caractères de la chaîne courante (en commençant à l'indice `index`) avec `num2` copies du caractère `ch` ;
- remplacer les caractères de la chaîne courante depuis `start` jusqu'à `end` avec la chaîne `str` ;
- remplacer les caractères de la chaîne courante depuis `start` jusqu'à `end` avec les `num` premiers caractères de `str` ;
- remplacer les caractères de la chaîne courante depuis `start` jusqu'à `end` avec `num` copies du caractère `ch`.

Par exemple, le code suivant affiche la chaîne « Ils disent que ça ressemble à…un truc très cool,Vincent. »

```
std::string s = "Ils disent que ça ressemble à... un
➥ très GROS truc!";
std::string s2 = "un truc très cool, Vincent.";
s.replace( 32, s2.length(), s2 );
std::cout << s << std::endl;
```

Insérer dans une chaîne

```
iterator string::insert( iterator it, const char&
➥ ch );
string& string::insert( size_type index, const
➥ string& str );
string& string::insert( size_type index,  const
➥ char* str );
string& string::insert( size_type index1, const
➥ string& str, size_type index2, size_type num );
string& string::insert( size_type index, const
➥ char* str, size_type num );
string& string::insert( size_type index, size_type
➥ num, char ch );
void string::insert( iterator it, size_type num,
➥ const char& ch );
void string::insert( iterator it, iterator debut,
➥ iterator fin );
```

Ces différentes versions de `insert()` permettent d'insérer dans la chaîne courante :

- le caractère `ch` avant la position indiquée par l'itérateur `it` ;
- la chaîne `str` à l'indice `index` ;
- la sous-chaîne `str` (commençant à l'indice `index2` et de longueur `num`), à l'indice `index1` ;
- les `num` premiers caractères de `str`, à l'indice `index` ;
- `num` fois le caractère `ch`, à l'indice `index` ;
- `num` fois le caractère `ch`, avant la position indiquée par l'itérateur `it` ;
- les caractères de la séquence [`debut`, `fin`[, avant la position indiquée par l'itérateur `it`.

Concaténer des chaînes

```
string& string::append( const string& str );
string& string::append( const char* str );
string& string::append( const string& str,
➥ size_type index, size_type len );
string& string::append( const char* str, size_type
➥ num );
string& string::append( size_type num, char ch );
string& string::append( input_iterator start,
➥ input_iterator end );
```

Les différentes fonctions `append()` permettent d'ajouter à la fin de la chaîne courante :

- la chaîne `str` ;
- la sous-chaîne de `str` commençant à l'indice `index` et de longueur `len` ;
- les `num` premiers caractères de `str` ;
- `num` fois le caractère `ch` ;
- les caractères contenus dans la séquence [`debut`, `fin`[.

Par exemple, le code ci-après utilise `append()` pour ajouter dix points d'exclamation à une chaîne. Il affichera la chaîne « Bonjour le monde!!!!!!!!!! ».

```
std::string str = "Bonjour le monde";
str.append(10, '!');
std::cout << str << std::endl;
```

Cet autre exemple ajoute une sous-chaîne à une autre chaîne. Il affichera « str1 vaut : Une première chaîne… rallongée. »

```
std::string str1 = "Une première chaîne...";
std::string str2 = "qui peut être rallongée...";
str1.append(str2, 16, 11);
std::cout << "str1 vaut : " << str1 << std::endl;
```

Effacer une partie d'une chaîne

```
iterator string::erase( iterator it );
iterator string::erase( iterator debut, iterator fin );
string& string::erase( size_type index = 0, size_type
➥ num = npos );
```

Les fonctions `erase()` permettent d'effacer :

- le caractère pointé par l'itérateur `it`, puis de renvoyer l'itérateur sur le caractère suivant ;
- les caractères de la sous-séquence [`debut`, `fin`[, puis de renvoyer l'itérateur pointant sur le caractère suivant le dernier supprimé ;
- les `num` caractères à partir de l'indice `index`, puis de renvoyer une référence sur la chaîne elle-même.

Les paramètres `index` et `num` ont des valeurs par défaut et peuvent être omis. Si vous ne spécifiez pas `num`, tous les caractères après l'indice `index` inclus seront supprimés. Si vous n'indiquez aucun des deux, toute la chaîne sera vidée ; c'est l'équivalent d'un `clear()`.

```
std::string s("Alors, on aime les beignets ? Il faut
➥ beignetiser le monde entier !");
std::cout << "La chaîne originale est '" << s << "'" <<
➥ std::endl;

s.erase( 50, 16 );
std::cout << "Maintenant c'est '" << s << "'" << std::endl;
s.erase( 29 );
std::cout << "Maintenant c'est '" << s << "'" << std::endl;
s.erase();
std::cout << "Maintenant c'est '" << s << "'" << std::endl;
```

Le code ci-avant produira la sortie ci-après.

```
The original string is 'Alors, on aime les beignets ?
➥ Il faut beignetiser le monde entier !'
Maintenant c'est 'Alors, on aime les beignets ? Il faut
➥ beignetiser !'
Maintenant c'est 'Alors, on aime les beignets ?'
Maintenant c'est ''
```

Lire des lignes dans un flux

```
istream& getline(istream& is, string& s, char
➥ delimiteur = '\n' );
```

La fonction `getline()` n'est pas une fonction membre de `std::string<>` mais une fonction globale. Elle lit une ligne dans le flux `is` et stocke les caractères lus dans la chaîne `s`. Lire une ligne consiste à lire tous les caractères qui se présentent jusqu'à en rencontrer un égal au `delimiteur`.

Par exemple, le code suivant lit une ligne sur l'entrée standard et l'affiche sur la sortie standard :

```
std::string s;
std::getline( std::cin, s );
std::cout << "Vous avez ecrit : " << s << std::endl;
```

Après avoir récupéré le contenu d'un flux dans une chaîne, vous trouverez probablement utile d'utiliser un `std::string_stream` pour en extraire certains types d'information. Par exemple, le code ci-après lit des nombres sur l'entrée standard, tout en ignorant les lignes commentées commençant par « // ».

```
string s;
while ( std::getline(std::cin,s) )
{
   if ( s.size() >= 2 && s[0] == '/' && s[1] == '/' )
   {
      std::cout << "* commentaire ignoré : " << s <<
      ➥ std::endl;
   }
   else
   {
       std::istringstream ss(s);
       double d;
       while ( ss >> d )
       {
           std::cout << "* un nombre : " << d << std::
           ➥ endl;
       }
   }
}
```

Avec ce code, vous pouvez, en entrant les mêmes valeurs, obtenir le résultat suivant :

```
// test
* commentaire ignoré : // test
22.3 -1 3.13149
* un nombre : 22.3
* un nombre : -1
* un nombre : 3.14159
// prochaine séquence
* commentaire ignoré : // prochaine séquence
1 2 3 4 5
* un nombre : 1
* un nombre : 2
* un nombre : 3
* un nombre : 4
* un nombre : 5
50
* un nombre : 50
```

11

Fichiers et flux

La bibliothèque standard (STL) définit, à travers l'en-tête `<iostream>`, une série de flux dits standard :

- `std::cout` est un objet de type `std::ostream` permettant d'afficher des données sur la sortie standard ;
- `std::cerr` est un autre objet `std::ostream` permettant la même chose, mais sans mise en tampon ;
- `std::clog` est la version avec mise en tampon de `std::cerr` ;
- `std::cin` est un objet de type `std::istream` permettant de lire des données sur l'entrée standard.

L'en-tête `<fstream>` contient tout le nécessaire pour effectuer des opérations sur fichiers avec les classes `std::ifstream`, pour la lecture, et `std::ofstream`, pour l'écriture.

Certains comportements des flux d'entrée/sortie de la bibliothèque standard C++ (comme la précision, la justification, etc.) peuvent être modifiés grâce aux divers manipulateurs de flux.

Ouvrir un fichier

```
#include <fstream>
std::fstream(const char* filename, openmode mode);
std::ifstream(const char* filename, openmode mode);
std::ofstream(const char* filename, openmode mode);
std::fstream::open(const char* filename, openmode
➥ mode);
```

Ces trois classes permettent de manipuler des fichiers. Le paramètre `mode` est optionnel. Vous trouverez les différentes significations et possibilités dans le tableau ci-après. Elles s'utilisent de manière analogue aux flux prédéfinis `std::cin` et `std::cout`.

Mode d'ouverture des fichiers

Mode	Description
`std::ios::app`	Ajouter à la fin (*append*)
`std::ios::ate`	Se placer à la fin lors de l'ouverture
`std::ios::binary`	Ouvrir le fichier en mode binaire
`std::ios::in`	Ouvrir le fichier en lecture
`std::ios::out`	Ouvrir le fichier en écriture
`std::ios::trunc`	Écraser le fichier existant
`std::ios::nocreate`	Unix seulement, ne crée pas le fichier s'il n'existe pas

Attention

Avec Microsoft, vous trouverez ces constantes dans `ios_base` en lieu et place de `ios`.

L'exemple suivant ajoute le contenu d'un fichier à un autre :

```
char temp;
std::ifstream fin("fichier1.txt");
std::ofstream fout("fichier2.txt", std::ios::app);
while ( fin >> temp )
    fout << temp;
fin.close();
fout.close();
```

Tester l'état d'un flux

```
stream::operator bool();
bool stream::fail();
bool stream::good();
bool stream::bad();
bool stream::eof();
ios::iostate stream::rdstate();
```

La conversion implicite en booléen permet de tester aisément si une erreur est survenue. Par ce biais, il est facile de tester si l'ouverture d'un fichier s'est bien déroulée. Le code ci-après l'illustre simplement. Si vous préférez un appel plus explicite, utilisez la fonction membre `fail()`.

```
std::string nom_de_fichier = "data.txt";
std::ifstream fichier( nom_de_fichier.c_str() );
if ( ! fichier )
   std::cout << "Erreur lors de l'ouverture du fichier.\n";
```

La fonction membre `good()` permet de vérifier qu'aucune erreur n'est apparue. La fonction membre `bad()` permet de tester si une erreur fatale est survenue. La fonction membre `eof()` permet de tester simplement si la fin du fichier est atteinte.

Par exemple, le code ci-après lit des données sur le flux d'entrée `in` et les écrits sur le flux de sortie `out` pour enfin utiliser `eof()` et vérifier qu'aucune erreur n'est survenue.

```
char tampon[TAILLE];
do
{
    in.read( tampon, TAILLE );
    std::streamsize n = in.gcount();
    out.write( tampon, n );
}
while ( in.good() );
if ( in.bad() || !in.eof() )
{
    // une erreur fatale est survenue
}
in.close();
```

Enfin, la fonction membre `rdstate()` retourne l'état complet du flux considéré. Le tableau suivant donne la liste des valeurs possibles.

État d'un flux

Flag	Description
`std::ios::badbit`	Une erreur fatale est survenue
`std::ios::eofbit`	Fin de fichier atteinte
`std::ios::failbit`	Une erreur non critique est survenue
`std::ios::goodbit`	Aucune erreur n'est survenue

Attention

Avec Microsoft, vous trouverez ces constantes dans `ios_base` en lieu et place de `ios`.

Lire dans un fichier

```
std::istream& operator >> (std::istream&, …);
std::istream& std::istream::getline(char* tampon,
➥ stringsize n);
std::istream& std::istream::getline(char* tampon,
➥ stringsize n, char delim);
std::streamsize std::fstream::gcount();
int std::fstream::get();
std::istream& std::istream::get(char& ch);
std::istream& std::istream::get(char* tampon,
➥ streamsize num);
std::istream& std::istream::get(char* tampon,
➥ streamsize num, char delim);
std::istream& std::istream::get(streambuf& tampon);
std::istream& std::istream::get(streambuf& tampon,
➥ char delim);
int std::fstream::peek();
std::istream& std::istream::read(char* buffer,
➥ streamsize num);
```

Il existe plusieurs façons de lire dans un flux. La plus « pratique » est d'utiliser l'opérateur de redirection >>, car la plupart des conversions sont alors faites automatiquement. De plus, comme le montre l'exemple ci-après, utiliser conjointement cet opérateur avec le type chaîne permet de lire un fichier mot par mot. La séparation des mots ne correspond pas exactement au langage naturel. Elle se base uniquement sur les caractères espace, tabulation et retour chariot.

```
std::ifstream fin("donnees.txt");
std::string s;
while ( fin >> s )
    std::cout << "Mot lu : " << s << std::endl;
```

L'exemple suivant montre comment lire un fichier ligne par ligne :

```
std::ifstream fichier("donnees.txt");
const int TAILLE = 100;
char str[TAILLE];
while ( fichier.getline(str, TAILLE) )
    std::cout << "Ligne lue : " << s << std::endl;
```

Si vous souhaitez éviter le tableau de caractères à la C, vous pouvez utiliser la fonction `std::getline()` qui lit des lignes et les stocke dans un `std::string` (voir la section « Lire des lignes dans un flux » au Chapitre 10 pour de plus amples explications).

```
std::ifstream fichier("donnees.txt");
std::string s;
while ( std::getline(fichier,s) )
    std::cout << "Ligne lue : " << s << std::endl;
```

`gcount()` est utile pour connaître le nombre de caractères effectivement lus lors de la dernière opération de lecture.

Les fonctions membres `get()` permettent de :

- lire un caractère et retourner sa valeur ;
- lire un caractère et le stocker dans la variable `ch` ;
- lire jusqu'à `num-1` caractères (s'arrête si la fin du fichier ou un retour chariot ou le caractère `delim` est atteint) ;
- lire tous les caractères jusqu'à la fin, le prochain retour chariot ou le caractère `delim`, et les stocker dans le tampon donné.

```
char ch;
std::ifstream fichier("donnees.txt");
while (fichier.get(ch))
    std::cout << ch;
fichier.close();
```

La fonction membre `peek()` retourne le prochain caractère du flux, sans pour autant le retirer de celui-ci.

Enfin, la fonction membre `read()` permet de lire `num` octets (et pas caractères) dans le flux et de les placer dans le tampon donné. Si la fin de fichier est atteinte avant, la lecture s'arrête et les octets lus sont placés dans le tampon.

```
struct Rectangle { int h,w; };
// …
fichier.read(reinterpret_cast<char*>(&rect),
➥ sizeof(Rectangle));
if (fichier.bad())
{
    std::cerr << "Erreur lors de la lecture des données\n";
    exit(0);
}
```

Astuce : ignorer une partie d'un flux

`std::istream& std::istream::ignore(streamsize num=1,`
`➥ int delim=EOF);`

La fonction membre `ignore()` s'utilise avec les flux d'entrée. Elle lit et ignore jusqu'à `num` caractères ou moins si le caractère `delim` est rencontré avant. Par défaut, `num` vaut 1 et `delim` correspond à la fin du fichier.

Cette fonction peut parfois s'avérer utile lorsque l'on utilise conjointement la fonction `getline()` et l'opérateur >>. Par exemple, si vous lisez une entrée finissant par une fin de ligne avec l'opérateur >>, le retour chariot reste présent dans le flux. Comme `getline()` s'arrête par défaut sur le prochain retour chariot, le prochain appel à cette fonction renverra une chaîne vide. Dans ce cas, la fonction `ignore()` peut être appelée avant `getline()` pour débarrasser le flux du retour chariot gênant.

Écrire dans un fichier

```
std::ostream& std::ostream::put( char ch );
std::ostream& std::ostream::write(const char*
➥ tampon, streamsize num )
std::ostream& std::ostream::flush();
std::ostream& operator << (std::ostream&, …);
```

```
std::istream& std::istream::putback(char ch);
```

La fonction membre `put()` écrit le caractère `ch` dans le flux. La fonction membre `write()` écrit les `num` premiers octets du `tampon` donné dans le flux.

La fonction `flush()` force l'écriture des tampons internes du flux. Ceci est particulièrement utile pour l'écriture d'information de débogage : dans certains cas de plantage, une partie des données écrites dans un flux de fichiers peut ne pas avoir été transférée sur le disque mais être restée en mémoire, et donc perdue. Un appel judicieux à `flush()` assure le transfert du tampon interne vers le périphérique lié au flux.

Pour les flux en mode texte, vous disposez de nombreux opérateurs `>>` sur tous les types de base du C++. Il est même possible, à la `printf()`, de préciser comment formater les données à écrire. Reportez-vous à la section « Manipuler des flux » un peu plus loin dans ce chapitre.

La fonction membre `putback()`, contrairement à ce que l'intitulé de cette section pourrait laisser penser, s'utilise sur les flux de lecture. Elle permet de simuler un retour en arrière en remettant le caractère `ch` dans le flux, comme si on ne l'avait pas encore lu. Pour ceux qui connaissent la bibliothèque C, elle correspond à la fonction C `int ungetc(int ch, FILE*)`.

Se déplacer dans un flux

```
std::istream& std::istream::seekg( std::off_type
➥ offset, std::ios::seekdir origine );
std::istream& std::istream::seekg( std::pos_type
➥ position );
std::pos_type std::istream::tellg();
```

```
std::ostream& std::ostream::seekp( std::off_type
➥ offset, std::ios::seekdir origine );
std::ostream& std::ostream::seekp( std::pos_type
➥ position );
std::pos_type std::ostream::tellp();
```

Les fonctions membres `seekg()` permettent de déplacer le curseur de lecture d'un flux d'entrée, soit à la position `offset` relative à l'`origine` donnée, soit à la `position` absolue fournie. Le prochain appel de `get()` partira de ce nouvel emplacement.

La fonction membre `tellg()` permet de connaître la position actuelle dans le flux de lecture.

Position relative dans un flux

Valeur	Description
`std::ios::beg`	Décalage relatif au début du fichier
`std::ios::cur`	Décalage relatif à la position courante
`std::ios::end`	Décalage relatif à la fin du fichier

Les fonctions membres `seekp()` et `tellp()` sont les équivalentes des précédentes, mais pour les flux d'écriture (ou de sortie).

L'exemple suivant affiche la position courante d'un flux de fichier et l'affiche sur la sortie standard :

```
std::string s("Une chaîne de caractères quelconque...");
std::ofstream fichier("sortie.txt");
for (int i=0; i < s.length(); ++i)
{
    std::cout << "Position : " << fichier.tellp();
    fichier.put( s[i] );
    std::cout << " " << s[i] << std::endl;
}
fichier.close();
```

Manipuler des flux

```
std::fmtflags std::stream::flags();
std::fmtflags std::stream::flags( fmtflags f )
```

```
std::fmtflags std::stream::setf( fmtflags flags );
std::fmtflags std::stream::setf( fmtflags flags,
➥ fmtflags needed );
```

```
void std::stream::unsetf( fmtflags flags );
```

Les fonctions membres `flags()` retournent le masque de formatage des données du flux courant, ou permettent de le changer. Vous retrouverez les différentes valeurs de ce masque dans le tableau ci-après.

Les fonctions membres `setf()` permettent d'activer des options de formatage (`flags`). Le paramètre `needed` permet de ne changer que les options communes. La valeur retournée est l'état avant changement. La fonction membre `unsetf()` permet au contraire de désactiver l'option spécifiée.

```
int nombre = 0x3FF;
std::cout.setf( std::ios::dec );
std::cout << "Décimal : " << nombre << std::endl;
std::cout.unsetf( std::ios::dec );
std::cout.setf( std::ios::hex );
std::cout << "Héxadécimal : " << nombre << std::endl;
```

Grâce aux manipulateurs, le code précédent peut être remplacé par le code suivant, qui est beaucoup plus simple :

```
int nombre = 0x3FF;
std::cout << "Décimal : " << std::dec << nombre
          <<  std::endl
          << "Hexadécimal : " << std::hex << nombre
          << std::endl;
```

Manipulateurs de flux

Manipulateur	Description	Entrée	Sortie
`boolalpha`	Affiche les booléens sous forme textuelle («true» et «false»)	X	X
`dec`	Passe en mode décimal	X	X
`endl`	Écrit un caractère de fin de ligne, vide le tampon	_	X
`ends`	Écrit un caractère nul	_	X
`fixed`	Affiche les nombres réels en mode standard (par opposition à scientifique)	_	X
`flush`	Vide le tampon interne du flux	_	X
`hex`	Passe en mode hexadécimal	X	X
`internal`	Si un nombre est complété pour remplir une taille donnée, des espaces sont insérés entre le signe et le symbole de la base	_	X
`left`	Justifie le texte à gauche	_	X

Manipulateurs de flux *(suite)*

Manipulateur	Description	Entrée	Sortie
`noboolalpha`	N'affiche pas les booléens sous forme textuelle	X	X
`noshowbase`	N'affiche pas le préfixe servant de symbole de la base	_	X
`noshowpoint`	Désactive l'affichage forcé de la virgule et des zéros inutiles des nombres réels	_	X
`noshowpos`	Désactive l'affichage forcé du «+» devant les nombres positifs	_	X
`noskipws`	Pour ne plus ignorer les « blancs »	X	_
`nounitbuf`	Désactive le mode `initbuf`	_	X
`nouppercase`	Affiche le «e» de la notation scientifique et le «x» de la notation hexadécimale en minuscule	_	X
`oct`	Passe en mode octal	X	X
`right`	Passe en alignement à droite	_	X
`scientific`	Affiche les réels en mode scientifique	_	X
`showbase`	Affiche le préfixe, symbole de la base utilisée	_	X
`showpoint`	Affiche toujours le point des nombres réels	_	X
`showpos`	Afficher toujours un « plus » devant les nombres positifs	_	X
`skipws`	Passe en mode ignorer les « blancs »	X	_
`unitbuf`	Force l'écriture (vide le tampon) après chaque insertion	_	X
`uppercase`	Passe en mode majuscules forcées pour le «e» de la notation scientifique et le «x» de la notation hexadécimale	_	X
`ws`	Saute les « blancs » restant	X	_

Manipulateurs définis dans <iomanip>

Manipulateur	Description	Entrée	Sortie
`resetiosflags(int long)`	Met à « off » le flag spécifié	X	X
`setbase(int base)`	Précise la base à utiliser pour l'affichage	–	X
`setfill(int ch)`	Précise le caractère à utiliser pour le remplissage	–	X
`setiosflags(long f)`	Met à « on » le flag spécifié	X	X
`setprecision(int p)`	Fixe le nombre de chiffres après la virgule	–	X
`setw(int w)`	Fixe la largeur pour les fonctions d'alignement	–	X

Manipuler une chaîne de caractères comme un flux

```
#include <sstream>
std::stringstream::stringstream( [ std::string s [,
➥ std::openmode mode ]] );
void std::stringstream::str(std::string s);
std::string std::stringstream::str();
```

Les flux de chaînes de caractères s'utilisent de manière analogue aux flux de fichiers. Le paramètre `mode` est le même que pour ces derniers.

Écrire dans une chaîne de caractère comme dans un flux

```
std::ostringstream::ostringstream( [ std::string s
➥ [, std::openmode mode ]] );
```

Un objet `std::ostringstream` peut être utilisé pour écrire le contenu d'une chaîne. C'est un peu le pendant de la fonction C `sprintf()`.

```
std::ostringstream s_flux;
int i = 3;
s_flux << "Coucou puissance " << i << std::endl;
std::string str = s_flux.str();
std::cout << str;
```

Lire le contenu d'une chaîne comme avec un flux

```
std::istringstream::istringstream( [ std::string s
➥ [, std::openmode mode ]] );
```

Un objet `std::istringstream` permet de lire le contenu d'une chaîne, un peu comme le `sscanf()` du C.

```
std::istringstream flux_chaine;
std::string chaine = "33";
flux_chaine.str(chaine);
int i;
flux_chaine >> i;
std::cout << i << std::endl; // affiche 33
```

Vous pouvez aussi spécifier directement sur quelle chaîne travailler en la fournissant au constructeur :

```
std::string chaine = "33";
std::istringstream flux_chaine(chaine);
int i;
flux_chaine >> i;
std::cout << i << std::endl; // affiche 33
```

Un objet `std::stringstream` permet d'effectuer à la fois des opérations de lecture et d'écriture, comme pour les objets `std::fstream`.

12

Algorithmes standard

La bibliothèque standard (STL) fournit de nombreux algorithmes utilisables sur ses conteneurs ou tous conteneurs compatibles. Ce chapitre vous permettra d'en exploiter tout le potentiel.

Algorithmes standard

Algorithmes standard *(Suite)*

Nom	Description	Page
`count`	Compter le nombre d'éléments correspondant à une valeur donnée	223
`count_if`	Compter le nombre d'éléments conformes à un test donné	224
`equal`	Tester si deux séquences sont identiques	225
`equal_range`	Chercher la sous-séquence d'éléments tous égaux à un certain élément	226
`fill`	Initialiser une séquence avec une valeur	228
`fill_n`	Initialiser les *n* premiers éléments d'une séquence avec une valeur	228
`find`	Chercher le premier élément égal à une valeur dans une séquence	228
`find_end`	Chercher la dernière apparition d'une sous-séquence donnée	266
`find_first_of`	Chercher le premier élément dont la valeur est présente dans un ensemble donné	229
`find_if`	Chercher le premier élément vérifiant un test donné	228
`for_each`	Appliquer une fonction/foncteur sur tous les éléments d'une séquence	230
`generate`	Initialiser une séquence à l'aide d'un générateur de valeurs	231
`generate_n`	Initialiser les *n* premières valeurs d'une séquence avec un générateur de valeurs	231
`includes`	Déterminer si tous les éléments d'une séquence sont dans une autre	232
`inner_product`	Calculer le produit intérieur (produit scalaire généralisé) de deux séquences	234
`inplace_merge`	Fusionner deux séquences triées (dans la première)	243
`iota`	Initialiser les éléments d'une séquence avec une valeur (en l'incrémentant)	235

Algorithmes standard *(Suite)*

Nom	Description	Page
`is_heap`	Tester si la séquence est un tas	236
`is_sorted`	Tester si une séquence est triée	274
`iter_swap`	Échanger le contenu des deux variables pointées	275
`lexicographical_compare`	Tester si une séquence est lexicographiquement plus petite qu'une autre	238
`lexicographical_compare_3way`	Tester si une séquence est lexicographiquement plus petite (–1), égale (0), ou supérieure (1) qu'une autre	238
`lower_bound`	Chercher le premier endroit où insérer une valeur sans briser l'ordre de la séquence	241
`make_heap`	Transformer la séquence en tas	236
`max`	Récupérer le plus grand élément (entre deux)	245
`max_element`	Récupérer le plus grand élément d'une séquence	246
`merge`	Fusionner deux séquences triées (dans une troisième)	243
`min`	Récupérer le plus petit élément (entre deux)	245
`min_element`	Récupérer le plus petit élément d'une séquence	246
`mismatch`	Trouver le premier endroit où deux séquences diffèrent	247
`next_permutation`	Générer la prochaine plus grande permutation lexicographique d'une séquence	248
`nth_element`	Faire en sorte que le nième élément soit le même que si la séquence était triée et assurer qu'aucun élément à sa gauche ne soit plus grand qu'un à sa droite	250
`partial_sort`	Trier les *n* premiers éléments d'une séquence	251

Algorithmes standard *(Suite)*

Nom	Description	Page
`partial_sort_copy`	Copier les *n* plus petits éléments d'une séquence (le résultat est trié)	252
`partial_sum`	Calculer la somme partielle généralisée d'une séquence	253
`partition`	Couper une séquence en deux en fonction d'un prédicat (ne préserve pas forcément l'ordre des éléments identiques)	254
`pop_heap`	Retirer le plus grand élément d'un tas	236
`power`	Calculer *xi* (fonction puissance généralisée)	256
`prev_permutation`	Générer la prochaine plus petite permutation lexicographique d'une séquence	248
`push_heap`	Ajouter un élément à un tas	236
`random_sample`	Copier aléatoirement un échantillon d'une séquence (nombre d'éléments déterminés par la taille de la séquence résultat)	257
`random_sample_n`	Copier aléatoirement un sous-échantillon (de *n* éléments) d'une séquence, en préservant leur ordre d'origine	258
`random_shuffle`	Mélanger les éléments d'une séquence	259
`remove`	Supprimer les éléments égaux à une valeur donnée	260
`remove_copy`	Copier une séquence en omettant les éléments égaux à une valeur donnée	262
`remove_copy_if`	Copier une séquence en omettant les éléments vérifiant un test donné	262
`remove_if`	Supprimer les éléments vérifiant un test donné	260
`replace`	Remplacer tous les éléments égaux à une valeur par une autre	263

Algorithmes standard *(Suite)*

Nom	Description	Page
`replace_copy`	Copier une séquence en remplaçant certaines valeurs par une autre	263
`replace_copy_if`	Copier une séquence en remplaçant certaines valeurs vérifiant un test par une autre	263
`replace_if`	Remplacer tous les éléments respectant un test donné par une nouvelle valeur	263
`reverse`	Inverser l'ordre de la séquence	264
`reverse_copy`	Copier une séquence en inversant son ordre	264
`rotate`	Effectuer une rotation des éléments de la séquence	264
`rotate_copy`	Copier une séquence en effectuant une rotation des éléments de la séquence	264
`search`	Chercher la première apparition d'une sous-séquence donnée	265
`search_n`	Chercher la première apparition de *n* occurrences consécutives d'une valeur donnée	265
`set_difference`	Construire la différence de deux séquences triées	268
`set_intersection`	Construire l'intersection de deux séquences triées	270
`set_symmetric_difference`	Construire la différence symétrique de deux séquences triées	272
`set_union`	Construire l'union de deux séquences triées	273
`sort`	Trier une séquence (ne préserve pas forcément l'ordre des éléments identiques)	274
`sort_heap`	Transformer un tas en séquence triée	236
`stable_partition`	Couper une séquence en deux en fonction d'un prédicat (préserve l'ordre des éléments identiques)	254

Nom	Description	Page
`stable_sort`	Trier une séquence (préserve l'ordre des éléments identiques)	274
`swap`	Échanger le contenu de deux variables	275
`swap_ranges`	Échanger le contenu de deux séquences de même taille	275
`transform`	Transformer une (ou deux) séquences en une autre	276
`unique`	Supprimer les doublons d'une séquence	278
`unique_copy`	Copier une séquence en supprimant les doublons d'une séquence	278
`upper_bound`	Chercher le dernier endroit où insérer une valeur sans briser l'ordre de la séquence	241
`uninitialized_copy`	Copier à l'aide du constructeur par copie	281
`uninitialized_copy_n`	Copier à l'aide du constructeur par copie (*n* éléments)	281
`uninitialized_fill`	Initialiser à l'aide du constructeur par copie	282
`uninitialized_fill_n`	Initialiser à l'aide du constructeur par copie (*n* éléments)	282

Calculer la somme des éléments d'une séquence

```
#include <numeric>
TYPE accumulate(InputIterator debut, InputIterator
➥ fin, TYPE init);
TYPE accumulate(InputIterator debut, InputIterator
➥ fin, TYPE init, BinaryFunction f);
```

La fonction `accumulate()` calcule la somme de `init` plus tout les éléments de l'ensemble [`debut`, `fin`[. Si la fonction binaire `f` est fournie, elle sera utilisée à la place de l'opérateur +.

La complexité est en temps linéaire O(*n*) fois la complexité de l'opérateur utilisé.

```
std::list<double> l;
double moyenne = std::accumulate(l.begin(), l.end(), 0.0)
                 ➥/ l.size();
double produit = std::accumulate(l.begin(), l.end(), 1.0,
                 ➥ std::multiplies<double>());
```

Calculer les différences entre éléments consécutifs d'une séquence

```
#include <numeric>
TYPE adjacent_difference(InputIterator debut,
➥ InputIterator fin, OutputIterator resultat);
TYPE adjacent_difference(InputIterator debut,
➥ InputIterator fin, OutputIterator resultat,
➥ BinaryFunction f);
```

La fonction `adjacent_difference()` calcule la différence des éléments adjacents de l'ensemble [`debut`, `fin`[. Si l'ensemble en entrée contient les éléments (a, b, c, d), le résultat sera (a, b-a, c-b, d-c). Si la fonction binaire `f` est fournie, elle sera utilisée à la place de l'opérateur -.

Info

La sauvegarde dans le résultat du premier élément en entrée n'est pas inutile. Elle permet d'avoir suffisamment d'information pour reconstruire la séquence d'origine. En particulier, avec les opérateurs arithmétiques que sont l'addition et la soustraction, `adjacent_difference` et `partial_sum` sont la réciproque l'une de l'autre.

```
std::vector<int> v1;
// … initialisation de v1 avec des valeurs ...
std::vector<int> v2(v1.size());

std::cout << "V1 : ";
std::copy(v1.begin(), v1.end(), std::ostream_iterator
➥<int>(std::cout," "));
std::cout << std::endl;

std::adjacent_difference(v1.begin(), v1.end(), v2.
➥begin());
std::cout << "Différences : ";
std::copy(v2.begin(), v2.end(), std::ostream_iterator
➥<int>(std::cout," "));
std::cout << std::endl;

std::cout << "Reconstruction : ";
std::partial_sum(v2.begin(), v2.end(),
➥ std::ostream_ iterator<int>(std::cout," "));
std::cout << std::endl;
```

Chercher la première occurrence de deux éléments consécutifs identiques

```
#include <algorithm>
ForwardIterator adjacent_find(ForwardIterator debut,
➥ ForwardIterator fin);
ForwardIterator adjacent_find(ForwardIterator debut,
➥ ForwardIterator fin, BinaryPredicate pred);
```

La fonction `adjacent_find()` renvoie le premier itérateur `i` tel que `*i == *(i+1)` et tel que les itérateurs `i` et `i+1` appartiennent à la séquence [`debut`, `fin`[donnée. Si un tel itérateur n'existe pas, la fonction renvoie `fin`. Si le prédicat binaire `pred` est fourni, il sera utilisé à la place de l'opérateur `==`.

Par exemple, si `v` est un vecteur d'entier contenant les valeurs (1, 2, 3, 3, 4, 5, 6, 7, 8). Le code suivant permet de repérer l'emplacement de la paire de 3 :

```
std::vector<int>::iterator it = adjacent_find(v.begin(),
➥ v.end());
if (it == v.end())
    std::cout << "Pas d'éléments contigus égaux dans
    ➥ v\n";
else
    std::cout << "Deux éléments contigus trouvés, de
    ➥ valeur : "
              << *it << std::endl;
```

Rechercher un élément dans une séquence

```
#include <algorithm>
bool binary_search(ForwardIterator debut,
➥ ForwardIterator fin, const LessThanComarable& val);
bool binary_search(ForwardIterator debut,
➥ ForwardIterator fin, const TYPE& val,
➥ StrickWeakOrdering comp);
```

La fonction `binary_search()` recherche la valeur `val` dans [`debut`, `fin`[. Les éléments dans l'ensemble de recherche doivent impérativement être triés en ordre croissant (relativement à l'opérateur <).

Si `val` est trouvé, la fonction renvoie `true`, sinon elle renvoie `false`.

Si une fonction de comparaison `comp` est fournie, elle sera utilisée pour comparer les éléments. Bien évidemment, les éléments de l'ensemble de recherche doivent être triés d'après ce comparateur.

Cette recherche est logarithmique pour les itérateurs de type *RandomAccessIterator*, quasi linéaire sinon (logarithmique pour les comparaisons, linéaire pour le nombre d'étapes).

Attention

Cette recherche binaire ne fonctionne *que si* l'ensemble dans lequel est effectuée la recherche *est trié*.

L'opérateur de comparaison `comp` est une relation d'ordre strict (au sens mathématique du terme). C'est-à-dire que si a *comp* b est vrai, alors b *comp* a est faux. Également, si a *comp* b et b *comp* c sont vrais, alors a *comp* c est vrai. Enfin si a *comp* b et b *comp* a sont faux tous les deux, alors a et b sont équivalents (égaux).

Le code suivant teste la présence des nombres 0 à 9 dans un tableau :

```
for (int i=0 ; i<10 ; i++)
{
    if (binary_search(tab.begin(), tab.end(), i))
        std::cout << i << " est dans le tableau\n" ;
    else
        std::cout << i << " N'EST PAS dans le tableau\n" ;
}
```

Si le tableau en entrée vaut {–23, –12, 0, 1, 2, 4, 5, 6, 8, 9, 50, 100, 300}, vous obtiendrez :

```
0 est dans le tableau
1 est dans le tableau
2 est dans le tableau
3 N'EST PAS dans le tableau
4 est dans le tableau
5 est dans le tableau
6 est dans le tableau
7 N'EST PAS dans le tableau
8 est dans le tableau
9 est dans le tableau
```

Copier les éléments d'une séquence dans une autre

```
#include <algorithm>
OutputIterator copy(InputIterator debut,
➥ InputIterator fin, OutputIterator resultat);
```

La fonction `copy()` copie un par un les éléments de [`debut`, `fin`[dans `resultat`. Le résultat final se situe dans [`resultat`, `resultat + (fin - debut)` [. Généralement, la copie est

effectuée par l'opération `*(result + n) = *(first + n)` pour `n` allant de 0 à `fin - debut`, dans l'ordre croissant.

```
std::vector<int> source(10), dest(10);
// initialisation de la source
std::iota(source.begin(), source.end(), 1);
// copie les éléments de source dans dest
std::copy(source.begin(), source.end(), dest.begin());
```

Vous pouvez aussi utiliser l'algorithme de copie pour écrire le contenu d'une séquence sur la sortie standard très simplement. L'exemple suivant l'illustre, tout en séparant les éléments écrits par un espace :

```
// #include <iomanip> pour ostream_iterator
std::copy(source.begin(), source.end(),
        ➥ std::ostream_iterator<int>(std::cout, " "));
```

Attention

L'itérateur `resultat` doit pointer sur une séquence mémoire valide.

À cause de l'ordre de la copie, l'itérateur `resultat` ne doit pas être dans la séquence [`debut`, `fin`[. Par contre, la fin de la séquence résultat peut avoir une partie commune avec la séquence source. L'algorithme `copy_backward` a la restriction opposée.

Copier les éléments d'une séquence dans une autre en commençant par la fin

```
#include <algorithm>
BidirectionalIterator2 copy_backward
➥(BidirectionalIterator1 debut,
➥ BidirectionalIterator1 fin, BidirectionalIterator2
➥ resultat);
```

Cette fonction est presque la même que la précédente (`copy()`). La différence réside dans l'ordre de la copie. `copy_backward()` commence par le dernier et finit par le premier élément (sans inverser la séquence).

Ainsi, la fonction `copy_backward()` copie un par un les éléments de [`debut`, `fin`[dans `resultat`. Le résultat final se situe dans [`resultat - (fin - debut)`, `resultat`[. Généralement, la copie est effectuée par l'opération `*(result - n - 1) = *(first - n)` pour `n` allant de 0 à `fin - debut`, dans l'ordre croissant. La copie est donc faite depuis le dernier jusqu'au premier élément de la séquence.

L'exemple suivant copie les dix premiers éléments d'un vecteur à la fin de celui-ci :

```
std::copy_backward( vec.begin(), vec.begin() + 10,
➥ vec.end() );
```

Attention

L'itérateur `resultat` doit pointer sur une séquence mémoire [`resultat - (fin - debut)`, `resultat`[valide.

À cause de l'ordre de la copie, l'itérateur `resultat` ne doit pas être dans la séquence [`debut`, `fin`[. Par contre, le début de la séquence de résultat peut avoir une partie commune avec la séquence source. L'algorithme `copy()` a la restriction opposée.

Copier les *n* premiers éléments d'une séquence dans une autre

```
#include <algorithm>
OutputIterator copy_n(InputIterator debut, Size n,
➥ OutputIterator resultat);
```

La fonction `copy_n()` copie les éléments de la séquence [`debut`,`debut + n`[dans [`resultat`,`resultat +n`[. Généralement, la copie est faite par l'opération `*(resultat + i) = *(debut +i)` pour `i` allant de 0 à `n` non inclus, dans l'ordre croissant.

```
std::vector<int> V(5);
std::iota(V.begin(), V.end(), 1);

std::list<int> L(V.size());
std::copy_n( V.begin(), V.size(), L.begin());
```

Attention

`Size` doit être de type entier et `n` doit être positif. Les séquences [`debut`, `debut + n`[et [`resultat`, `resultat + n`[doivent être valides. L'itérateur `resultat` ne doit pas faire partie de la séquence source. La même restriction que celle de `copy` sur la superposition des séquences source et résultat doit être respectée.

Info

Cette fonction peut paraître redondante avec `copy`. Contrairement à cette dernière, `copy_n` permet d'utiliser des itérateurs de type *input iterator* et pas seulement *forward iterator*.

Compter le nombre d'éléments correspondant à une valeur donnée

```
#include <algorithm>
iterator_traits<InputIterator>::difference_type
➥ count(InputIterator debut, InputIterator fin,
➥ const EqualityComparable& valeur);
void count(InputIterator debut, InputIterator fin,
➥ const EqualityComparable& valeur, Size& n);
➥ // (ancien)
```

La fonction `count()` compte le nombre d'éléments de la séquence [`debut`, `fin`[étant égaux à `valeur`. Plus exactement, la première fonction retourne le nombre d'itérateurs `i` de [`debut`, `fin`[tels que `*i == valeur`. La seconde ajoute ce nombre à `n`.

Info

La deuxième version est celle que l'on trouve dans la STL d'origine, elle est conservée dans certaines implémentations mais est susceptible d'être enlevée à tout moment. Seule la première version fait partie du standard C++.

Voici un exemple simple illustrant l'utilisation de la STL avec des types C :

```
int A[] = { 4, 3, 8, 0, 2, 5, 7, 0, 3, 8, 5, 6 };
const int N = sizeof(A) / sizeof(int);
std::cout << "Il y a " << std::count(A, A+N, 0) <<
➥ "zéro(s).\n" ;
```

Compter le nombre d'éléments conformes à un test donné

```
#include <algorithm>
iterator_traits<InputIterator>::difference_type
➥ count_if(InputIterator debut, InputIterator fin,
➥ const UnaryPredicate& pred);
void count_if(InputIterator debut, InputIterator
➥ fin, const UnaryPredicate& pred, Size& n);
➥ // (ancien)
```

La fonction `count_if()` compte le nombre d'éléments de la séquence [`debut`, `fin`[qui respectent le prédicat unaire `pred`. Plus exactement, la première fonction retourne le nombre d'itérateurs `i` de [`debut`, `fin`[tels que `pred(*i)` est vrai. La seconde (ancienne et pas toujours implémentée) ajoute ce nombre à `n`.

L'exemple suivant compte tous les nombres pairs :

```
int A[] = { 4, 3, 8, 0, 2, 5, 7, 0, 3, 8, 5, 6 };
const int N = sizeof(A) / sizeof(int);
std::cout << "Il y a " << std::count_if(A, A+N,
➥ std::compose1(std::bind2nd(std::equal_to<int>(),0),
➥ std::bind2nd(std::modulus<int>(),2)))
➥ << " nombre(s) pair(s).\n";
```

Info

Les nouvelles interfaces de `count` utilisent les classes `iterator_traits`, qui reposent sur une particularité du C++ connue sous le nom de *spécialisation partielle*. La plupart des compilateurs n'implémentent pas la totalité de la norme ; ce faisant, même des compilateurs relativement récents n'implémentent pas toujours la spécialisation partielle. Si tel est le cas, la nouvelle version de `count` ne sera pas forcément présente (à moins qu'elle retourne un `size_t`) ; de même que les autres parties de la STL utilisant les `iterator_traits`.

Tester si deux séquences sont identiques

```
#include <algorithm>
bool equal(InputIterator1 debut1, InputIterator1
➥ fin1, InputIterator2 debut2);
bool equal(InputIterator1 debut1, InputIterator1
➥ fin1, InputIterator2 debut2, BinaryPredicate pred);
```

La fonction `equal()` retourne vrai si les deux séquences [`debut1`, `fin1`[et [`debut2`, `debut2 + (fin1 - debut1)`[sont égales en les comparant élément par élément. La première version utilise la comparaison `*i == *(debut2 + (i - debut1))`, et la seconde `pred(*i1,* (debut2 + (i - debut1))) == true`, pour tout `i` appartenant à la première séquence.

L'exemple suivant compare deux `vector` de même taille :

```
bool egaux = std::equal(v1.begin(), v1.end(), v2.begin());
```

Chercher la sous-séquence d'éléments tous égaux à un certain élément

```
#include <algorithm>
pair<ForwardIterator,ForwardIterator>
➥ equal_range(ForwardIterator debut,
➥ ForwardIterator fin, const LessThanComparable&
➥ valeur);
pair<ForwardIterator,ForwardIterator> equal_range
➥(ForwardIterator debut, ForwardIterator fin,
➥ const T& valeur, StricWeakOrdering comp);
```

La fonction `equal_range()` est une variante de `binary_search()` : elle renvoie la sous-séquence [`debut`, `fin`[dont les éléments sont tous équivalents (voir la note ci-après) à `valeur`. La première utilise l'opérateur de comparaison `<`, la deuxième la relation d'ordre stricte `comp`.

Info

La relation d'ordre utilisée doit être stricte, mais pas nécessairement totale. Il peut exister des valeurs `x` et `y` telles que les tests `x < y`, `x > y` (plus exactement `y < x`) et `x == y` soient tous faux. Trouver une `valeur` dans la séquence ne revient donc pas à trouver un élément qui soit égal à `valeur` mais *équivalent* : ni inférieur, ni supérieur à `valeur`. Si vous utilisez une relation d'ordre totale, l'équivalence et l'égalité représentent la même chose. C'est le cas par exemple pour la comparaison d'entiers ou de chaînes de caractères avec `strcmp`.

Attention

equal_range() ne doit être utilisé que sur une séquence triée avec la même relation d'ordre, ou selon une relation d'ordre « compatible ».

Cette fonction peut être vue comme une combinaison des fonctions `lower_bound()` et `upper_bound()`. Ce sont en effet ces valeurs que l'on retrouve dans la paire retournée par `equal_range()`. Mais c'est plus rapide que d'invoquer les deux fonctions précitées.

Pour bien comprendre ce que fait cette fonction, considérez son utilisation ci-après, en postulant que le vecteur `v` utilisé contienne les valeurs –78, –34, –6, 2, 5, 5, 5, 5, 6, 12.

```
std::pair<std::vector<int>::iterator, std::vector
➥ <int>::iterator> res;
res = std::equal_range(v.begin(), v.end(), 5);
std::cout << "Le premier emplacement pour insérer 5
➥est avant" << *res.first << " et le dernier (où il
➥ peut être inséré avant) est " << *res.second << ".\n";
```

Vous obtiendrez la phrase « Le premier emplacement pour insérer 5 est avant 5 et le dernier (où il peut être inséré avant) est 6. » Bien entendu, le 5 où l'on peut insérer avant est le premier de la liste.

Info

equal_range() peut retourner une séquence vide (une paire contenant deux fois le même itérateur). C'est le cas dès que la séquence d'entrée ne contient aucun élément équivalent à la `valeur` recherchée. L'itérateur renvoyé est alors la seule position où la `valeur` donnée peut être insérée sans violer la relation d'ordre.

Initialiser une séquence

```
#include <algorithm>
void fill(ForwardIterator debut, ForwardIterator
➥ fin, const T& valeur);
OutptIterator fill_n(OutputIterator debut,
➥ Size n, const T& valeur);
```

`fill()` affecte à tous les éléments de la séquence [`debut`, `fin`[la valeur donnée. `fill_n()` le fait sur la séquence [`debut`, `debut + n`[puis retourne `debut + n`.

```
std::vector<int> v(5);
std::vector<int>::iterator res;
std::fill(v.begin(), v.end(), -2);  // v = -2, -2, -2, -2, -2
res = std::fill_n(v.begin(), 3, 0); // v =  0,  0,  0, -2, -2
*res = 1;                           // v =  0,  0,  0,  1, -2
```

Chercher le premier élément tel que...

```
#include <algorithm>
InputIterator find(InputIterator debut, InputIterator
➥ fin, const EqualityComparable& value);
InputIterator find_if(InputIterator debut,
➥ InputIterator fin, Predicate pred);
```

`find()` retourne le premier itérateur de la séquence [`debut`, `fin`[tel que `*i == value`. L'algorithme `find_if()` utilise `pred(*i)==true`. Si aucun élément ne valide le test, la valeur de retour est `fin`.

L'exemple suivant renvoie un itérateur sur le premier élément positif d'une liste d'entiers :

```
res = std::find_if(L.begin(), L.end(),std::bind2nd
➥ (greater<int>, 0));
```

Celui-ci cherche la première occurrence de 7 :

```
res = std::find(L.begin(), L.end(), 7);
```

Chercher le premier élément parmi...

```
#include <algorithm>
InputIterator find_first_of(InputIterator debut1,
➥ InputIterator fin1, ForwardIterator debut2,
➥ ForwardIterator fin2);
InputIterator find_first_of(InputIterator debut1,
➥ InputIterator fin1, ForwardIterator debut2,
➥ ForwardIterator fin2, BinaryPredicate comp);
```

`find_first_of()` est un peu comme `find()` en ce sens qu'il recherche linéairement à travers la séquence d'entrée [`debut1`, `fin1`[. La différence tient au fait que `find()` cherche une valeur donnée dans la séquence, alors que `find_first_of()` recherche n'importe quelle valeur apparaissant dans la deuxième séquence [`debut2`, `fin2`[. Ainsi, `find_forst_of()` retourne l'itérateur pointant sur la première valeur de [`debut1`, `fin1`[appartenant à [`debut2`, `fin2`[, ou `fin1` sinon.

La première version de `find_first_of()` utilise l'opérateur `==` pour comparer les éléments. La deuxième utilise le prédicat `comp` fourni.

Par exemple, essayez la fonction ci-après sur ces chaînes de caractères : «Une phrase avec plusieurs mots.» ou «UnSeulMot».

```
char* premier_separateur(char* chaine, const int taille)
{
    const char* blanc = "\t\n ";
    return std::find_first_of(chaine, chaine+taille,
    ➥ blanc, blanc+4);
}
```

Appliquer une fonction/foncteur sur tous les éléments d'une séquence

```
#include <algorithm>
UnaryFunction for_each(InputIterator debut,
➥ InputIterator fin, UnaryFunction f);
```

`for_each()` applique la fonction `f` sur tous les éléments de la séquence [`debut`, `fin`[. Si `f` retourne une valeur, elle est ignorée. Les opérations sont faites dans l'ordre d'apparition des éléments de la séquence, de `debut` (inclus) à `fin` (exclus). À la fin, `for_each()` retourne l'objet fonction passé en paramètre.

Voici une façon un peu plus complexe que la combinaison de `copy()` et `ostream_iterator()` pour écrire le contenu d'une séquence sur la sortie standard :

```
template <class T>
struct Ecrire : public std::unary_function<T, void>
{
    Ecrire(std::ostream& un_flux) : flux(un_flux), compteur(0)
    {
    }
    void operator() (T x)
```

```
    {
        flux << x << ' ';
        compteur++;
    }

    std::ostream& flux;
    int compteur;
};

int main(int, char**)
{
    int tableau[] = { 1, 2, 3, 4, 5, 6, 7 };
    const int taille = sizeof(tableau) / sizeof(int);

    Ecrire<int> E = std::for_each(tableau,
    ➥ tableau+taille, Ecrire<int>(std::cout));
    std::cout << std::endl << E.compteur
            ➥<< " objets écrits.\n";
}
```

Initialiser une séquence à l'aide d'un générateur de valeurs

```
#include <algorithm>
ForwardIterator generate(ForwardIterator debut,
➥ ForwardIterator fin, Generator g);
OutputIterator generate_n(OutputIterator debut,
➥ Size n, Generator g);
```

`generate()` et `generate_n()` affectent à chaque élément de la séquence [`debut`, `fin`[(respectivement [`debut`, `debut + n`[) le résultat de la fonction génératrice `g()`, puis retourne l'itérateur `fin` (respectivement `debut + n`).

Info

La fonction génératrice est bien appelée *n* fois. Son résultat n'est pas stocké puis affecté à tous les éléments. Cela peut paraître lourd si cette fonction retourne le résultat d'un algorithme complexe, mais cela offre la souplesse d'utiliser comme fonction génératrice une fonction qui ne retourne pas toujours le même résultat, telle une fonction aléatoire ou une fonction incrémentale.

Info

Pour affecter la même valeur à tous les éléments de la séquence, utilisez plutôt `fill()` ou `fill_n()`.

Cet exemple écrit une liste de valeurs aléatoires sur la sortie standard :

```
std::generate_n(std::ostream_iterator<int>(std::
➥ cout,"\n"), 10, rand);
```

Tester si tous les éléments d'une séquence sont dans une autre

```
#include <algorithm>
bool includes(InputIterator debut1, InputIterator
➥ fin1, InputIterator debut2, InputIterator fin2);
bool includes(InputIterator debut1, InputIterator
➥ fin1, InputIterator debut2, InputIterator fin2,
➥ StrictWeakOrdering comp);
```

`includes()` teste si tous les éléments de la deuxième séquence [`debut2`, `fin2`[sont dans la première séquence [`debut1`, `fin1`[.

Cette fonction, qui s'exécute en temps linéaire, requiert que les deux séquences soient triées selon la relation d'ordre mentionnée (< par défaut).

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

#define TAILLE_TABLEAU(tab) (sizeof(tab) / sizeof(int))

#define TEST_INCLUDES(tab1,tab2) \
    (includes(tab1, tab1 + TAILLE_TABLEAU(tab1), \
                    tab2, tab2 + TAILLE_TABLEAU(tab2)) \
    ? "oui" : "non")

#define PRINT_TABLEAU(tab) \
    cout << "{ " << tab[0]; \
    for(int i=1; i<TAILLE_TABLEAU(tab); i++) \
        cout << ", " << tab[i]; \
    cout << "}";

#define PRINT_TEST(tab1,tab2) \
    PRINT_TABLEAU(tab1) \
    cout << " contient "; \
    PRINT_TABLEAU(tab2)      \
    cout << " ? " << TEST_INCLUDES(tab1,tab2) << endl ;

int A1[] = { 1, 2, 3, 4, 5, 6, 7 };
int A2[] = { 1, 4, 7 };
int A3[] = { 2, 7, 9 };
int A4[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
int A5[] = { 1, 2, 13, 13 };
int A6[] = { 1, 1, 3, 21 };

int main(int,char**)
{
    PRINT_TEST(A1,A2)
    PRINT_TEST(A1,A3)
    PRINT_TEST(A4,A5)
    PRINT_TEST(A4,A6)
    return 0;
}
```

Le programme précédent produit le résultat suivant :

```
{ 1, 2, 3, 4, 5, 6, 7} contient { 1, 4, 7} ? oui
{ 1, 2, 3, 4, 5, 6, 7} contient { 2, 7, 9} ? non
{ 1, 1, 2, 3, 5, 8, 13, 21} contient { 1, 2, 13, 13} ? non
{ 1, 1, 2, 3, 5, 8, 13, 21} contient { 1, 1, 3, 21} ? oui
```

Calculer le produit intérieur (produit scalaire généralisé) de deux séquences

```
#include <algorithm>
T inner_product(InputIterator1 debut1,
➥ InputIterator1 fin1, InputIterator2 debut2, T init);
T inner_product(InputIterator1 debut1,
➥ InputIterator1 fin1, InputIterator2 debut2, T
➥ init, BinaryFunction1 op1, BinaryFunction2 op2);
```

La fonction `inner_product()` calcule le produit intérieur (un cas particulier bien connu est le produit scalaire) de deux séquences de même taille. De manière pratique, si la deuxième séquence doit contenir au moins autant d'éléments que la première, les autres étant ignorés, il vaut mieux utiliser cette fonction sur des séquences de taille strictement identique.

Le résultat de la première version est comparable au pseudo-code suivant :

```
T resultat = init;
for (int i=0 ; i<taille(sequence1) ; i++)
    resultat = resultat + sequence1[i] * sequence2[i];
```

La deuxième version utilise l'opération `resultat = op1(result, op2(*it1,*it2))`.

Voici un exemple simple illustrant le produit scalaire de deux vecteurs :

```
double V1[3] = { 0.5, 1.2, 5.4 };
double V2[3] = { 10.0, -2.7, 3.76 };
double p = std::inner_product(V1, V1+3, V2, 0.0);
// p == 22.064
```

Initialiser les éléments d'une séquence avec une valeur (en l'incrémentant)

```
#include <numeric>
void iota(ForwardIterator debut,
➥ ForwardIterator fin, T valeur);
```

`iota()` affecte un à un les éléments de la séquence [`debut`, `fin`[avec la `valeur` donnée, en l'incrémentant entre chaque affectation. Ainsi, le code suivant remplira le tableau d'entiers `V` avec les valeurs 7, 8, 9, …, 16.

```
std::vector<int> V(10);
std::iota( V.begin(), V.end(), 7);
```

Attention

Cette fonction est une extension SGI et ne fait pas partie du standard. Elle est décrite ici au cas où vous la rencontreriez dans du code existant ou si vous travailliez dans cet environnement.

Vous pouvez la trouver dans `<ext/numeric>` dans l'espace de noms `__gnu_cxx` avec g++.

Transformer une séquence en tas et l'utiliser

```
#include <algorithm>
bool is_heap(RandomAccessIterator debut,
➥ RandomAccessIterator fin);
bool is_heap(RandomAccessIterator debut,
➥ RandomAccessIterator fin, StrictWeakOrdering comp);
```

```
void make_heap(RandomAccessIterator debut,
➥ RandomAccessIterator fin);
bool make_heap(RandomAccessIterator debut,
➥ RandomAccessIterator fin, StrictWeakOrdering comp);
```

```
void sort_heap(RandomAccessIterator debut,
➥ RandomAccessIterator fin);
bool sort_heap(RandomAccessIterator debut, Random
➥ AccessIterator fin, StrictWeakOrdering comp);
```

```
void push_heap(RandomAccessIterator debut,
➥ RandomAccessIterator fin);
bool push_heap(RandomAccessIterator debut, Random
➥ AccessIterator fin, StrictWeakOrdering comp);
```

```
void pop_heap(RandomAccessIterator debut,
➥ RandomAccessIterator fin);
bool pop_heap(RandomAccessIterator debut, Random
➥ AccessIterator fin, StrictWeakOrdering comp);
```

`is_heap()` retourne vrai si la séquence [`debut`, `fin`[est un tas, et faux sinon. La première version utilise l'opérateur `<`, la seconde la relation d'ordre `comp`.

`make_heap()` transforme l'ordre des éléments de la séquence de manière à ce qu'elle forme un tas.

`sort_heap()` transforme une séquence sous forme de tas en une séquence triée standard. Ce tri n'est pas un tri stable : il ne garantit pas la préservation de l'ordre relatif d'éléments égaux.

`push_heap()` ajoute un élément au tas, en postulant que [`debut`, `fin-1`[est déjà un tas et que l'élément à ajouter est `*(fin-1)`.

`pop_heap()` supprime les plus grands éléments du tas (en l'occurrence `*debut`). Après exécution, on retrouve l'élément supprimé à la position `fin-1`, et les autre éléments sont sous forme de tas dans la séquence [`debut`, `fin-1`[. Ainsi un `push_heap()` suivi d'un `pop_heap()` revient à… ne rien faire en y passant du temps.

Info

Un tas est une manière particulière d'ordonner les éléments d'une séquence [`debut`, `fin`[. Les tas sont utiles, particulièrement pour les tris et les queues de priorité, car ils respectent deux propriétés importantes. Premièrement, `*debut` est le plus grand élément du tas. Deuxièmement, il est possible d'ajouter un élément à un tas (en utilisant `push_heap()`), ou de supprimer `*debut`, en temps logarithmique. En interne, un tas est un arbre stocké sous forme d'une séquence. L'arbre est construit de tel façon que chaque nœud soit plus petit ou égal à son nœud parent.

```
std::vector<int> tas(9);
for (int i=0; i<=8; i++) tas[i]=i;

assert(is_heap(tas.begin(),tas.end())==false);
std::copy(tas.begin(),tas.end(),std::ostream_iterator
➥<int>(std::cout," "));

std::make_heap(tas.begin(), tas.end());
assert(is_heap(tas.begin(),tas.end())==true);
```

```
std::copy(tas.begin(),tas.end(),std::ostream_iterator
➥ <int>(std::cout," "));

tas.push_back(9);
std::push_heap(tas.begin(),tas.end());
std::copy(tas.begin(),tas.end(),std::ostream_iterator
➥ <int>(std::cout," "));

std::pop_heap(tas.begin(),tas.end());
std::copy(tas.begin(),tas.end(),std::ostream_iterator
➥ <int>(std::cout," "));

std::sort_heap(tas.begin(), tas.end());
assert(is_heap(tas.begin(),tas.end())==false);
std::copy(tas.begin(),tas.end(),std::ostream_iterator
➥ <int>(std::cout," "));
```

L'exemple précédent produira la sortie suivante :

```
0 1 2 3 4 5 6 7 8
8 7 6 3 4 5 2 1 0
9 8 6 3 7 5 2 1 0 4
8 7 6 3 4 5 2 1 0 9
0 1 2 3 4 5 6 7 8 9
```

Comparer lexicographiquement deux séquences

```
#include <algorithm>
bool lexicographical_compare(InputIterator1
➥ debut1, InputIterator1 fin1, InputIterator2
➥ debut2, InputIterator2 fin2);
bool lexicographical_compare(InputIterator1
➥ debut1, InputIterator1 fin1, InputIterator2
➥ debut2, InputIterator2 fin2, BinaryPredicate comp);
```

```
int lexicographical_compare_3way(InputIterator1
➥ debut1, InputIterator1 fin1, InputIterator2
➥ debut2, InputIterator2 fin2);
```

`lexicographical_compare()` retourne vrai si la séquence [`debut1`, `fin1`[est lexicographiquement plus petite que la séquence [`debut2`, `fin2`[et faux sinon.

`lexicographical_compare_3way()` est une généralisation de la fonction C `strcmp()`. Elle retourne un nombre négatif si la première séquence est (lexicographiquement) plus petite que la deuxième, un nombre positif si la deuxième est plus petite que la première, et zéro sinon (c'est-à-dire si elles sont lexicographiquement équivalentes).

Info

`lexicographical_compare_3way()` peut paraître identique au code suivant :

```
lexicographical_compare(d1,f1, d2,f2)
    ? -1
    : (lexicographical_compare(d2,f2, d1,f1) ? -1 : 0)
```

C'est vrai pour le résultat obtenu, mais faux vis-à-vis des performances. Un appel à `lexicographical_compare_3way()` est plus performant que deux appels à `lexicographical_compare()`.

Attention

Avec g++, `lexicographical_compare_3way()` se trouve dans le fichier en-tête `<ext/algorithm>` et dans l'espace de noms `__gnu_cxx`.

L'exemple ci-après montre que l'on peut comparer des tableaux d'entiers comme si l'on comparait des chaînes de

caractères. Il donne un aperçu des possibilités qu'offrent ces fonctions.

```
int A1[] = { 5, 3, 7, 1, 9, 5, 8 };
int A2[] = { 5, 3, 7, 0, 7, 7, 5 };
int A3[] = { 2, 4, 6, 8 };
int A4[] = { 2, 4, 6, 8, 10};

const int N1 = sizeof(A1) / sizeof(int);
const int N2 = sizeof(A2) / sizeof(int);
const int N3 = sizeof(A3) / sizeof(int);
const int N4 = sizeof(A4) / sizeof(int);

int C12 = std::lexicographical_compare(A1, A1+N1, A2,
➥ A2+N2);
int C34 = std::lexicographical_compare(A3, A3+N3, A4,
➥ A4+N4);

int C12w = std::lexicographical_compare(A1, A1+N1, A2,
➥ A2+N2);
int C34w = std::lexicographical_compare(A3, A3+N3, A4,
➥ A4+N4);
int C34b = std::lexicographical_compare(A3, A3+N3, A4,
➥ A4+N4-1);

std::cout << "A1 < A2 == " << (C12 ? "vrai" : "faux")
➥ << std::endl;
std::cout << "A3 < A4 == " << (C34 ? "vrai" : "faux")
➥ << std::endl;

std::cout << "A1 ? A2 == " << C12w << std::endl;
std::cout << "A3 ? A4 == " << C34w << std::endl;
std::cout << "A3 ? A4' == " << C34b << std::endl;
```

Cet exemple produira la sortie suivante :

```
A1 < A2 == faux
A3 < A4 == vrai
A1 ? A2 == 1
A3 ? A4 == -1
A3 ? A4' == 0
```

Chercher le premier/dernier endroit où insérer une valeur sans briser l'ordre d'une séquence

```
#include <algorithm>
ForwardIterator lower_bound(ForwardIterator debut,
➥ ForwardIterator fin, const LessThanComparable&
➥ valeur);
ForwardIterator lower_bound(ForwardIterator debut,
➥ ForwardIterator fin, const LessThanComparable&
➥ valeur, StrictWeakOrdering comp);
```

```
ForwardIterator upper_bound(ForwardIterator debut,
➥ ForwardIterator fin, const LessThanComparable&
➥ valeur);
ForwardIterator upper_bound(ForwardIterator debut,
➥ ForwardIterator fin, const LessThanComparable&
➥ valeur, StrictWeakOrdering comp);
```

`lower_bound()` est une variante de `binary_search()`. Elle cherche et renvoie la première position où la `valeur` donnée peut être insérée dans la séquence triée [`debut`, `fin`[sans rompre le tri.

`upper_bound()` cherche et renvoie la dernière position possible.

Dans les deux cas, la séquence d'entrée est supposée triée selon l'opérateur `<` ou la relation d'ordre `comp` donnée.

Par exemple, le code suivant utilise ces fonctions pour insérer des valeurs triées selon les dizaines :

```
#include <iostream>
#include <iterator>
#include <vector>
```

```
#include <ext/algorithm>

struct CompareDizaines
{
    bool operator()(int a, int b) const
    {
        return a/10 < b/10;
    }
};

void affiche(const std::vector<int>& V)
{
    std::cout << "V = ";
    std::copy(V.begin(), V.end(),
          ➥ std::ostream_iterator<int>(std::cout," "));
    std::cout << "(";
    if (__gnu_cxx::is_sorted(V.begin(), V.end(),
                ➥ CompareDizaines()) == false)
        std::cout << "NON ";
    std::cout << "trié)" << std::endl;
}

int main(int,char**)
{
    std::vector<int> V;
    std::vector<int>::iterator it;
    int valeur;

    V.push_back(-123);
    V.push_back(- 45);
    V.push_back(   0);
    V.push_back(  50);
    V.push_back(  87);
    V.push_back(  83);
    V.push_back( 120);
    affiche(V);

    valeur = 82;
    it = std::lower_bound(V.begin(), V.end(), valeur,
                      ➥ CompareDizaines());
    V.insert(it, valeur);
```

```
    std::cout << "\nAprès insertion 'lower_bound' de "
              << valeur << ":\n";
    affiche(V);

    valeur = 81;
    it = std::upper_bound(V.begin(), V.end(), valeur,
                          ➥ CompareDizaines());
    V.insert(it, valeur);
    std::cout << "\nAprès insertion 'upper_bound' de "
              << valeur << ":\n";
    affiche(V);

    return 0;
}
```

Ce code produit la sortie suivante :

```
V = -123 -45 0 50 87 83 120 (trié)

Après insertion 'lower_bound' de 82:
V = -123 -45 0 50 82 87 83 120 (trié)

Après insertion 'upper_bound' de 81:
V = -123 -45 0 50 82 87 83 81 120 (trié)
```

Fusionner deux séquences triées

```
#include <algorithm>
OutputIterator merge(InputIterator1 debut1,
➥ InputIterator1 fin1, InputIterator2 debut2,
➥ InputIterator2 fin2, OutputIterator resultat);
OutputIterator merge(InputIterator1 debut1,
➥ InputIterator1 fin1, InputIterator2 debut2,
➥ InputIterator2 fin2, OutputIterator resultat,
➥ StrictWeakOrdering comp);
```

```
void inplace_merge(BidirectionalIterator debut,
➥ BidirectionalIterator milieu,
➥ BidirectionalIterator fin);
void inplace_merge(BidirectionalIterator debut,
➥ BidirectionalIterator milieu,
➥ BidirectionalIterator fin, StrictWeakOrdering comp);
```

`merge()` combine deux séquences triées [`debut1`, `fin1`[et [`debut2`, `fin2`[en une seule séquence triée. Les éléments des deux séquences d'entrée sont copiés pour former la séquence [`resultat`, `resultat + (fin1 - debut1) + (fin2 - debut2)`[. Les séquences d'entrée ne doivent pas avoir de partie commune avec la séquence résultat. Notez également que la fonction `merge()` est *stable* : si `i` précède `j` dans une des séquence d'entrée, alors la copie de `i` précédera la copie de `j` dans la séquence résultat.

`inplace_merge()` effectue le même genre d'opération. Si [`debut`, `milieu`[et [`milieu`, `fin`[respectent l'ordre de tri < (ou `comp`), alors à la fin de `inplace_merge()`, [`debut`, `fin`[respecte l'ordre de tri < (ou `comp`). Par respecter l'ordre de tri, on entend que pour tous itérateurs `i` et `j` d'une séquence, si `i` précède `j`, alors `*j < *i` est faux (respectivement `comp(*j,*i)` est faux).

Attention

La complexité de ces deux algorithmes est différente.

`merge()` est linéaire sur le nombre total d'éléments, soit O(N) où N vaut `(fin1 - debut1) + (fin2 - debut2)`.

`inplace_merge()` est un algorithme adaptatif : sa complexité dépend de la mémoire disponible. Si la première séquence est vide, aucune comparaison n'est faite. Dans le pire des cas, lorsqu'il n'y a pas de mémoire tampon possible, sa complexité est en O(N log N), où N vaut `debut - fin`. Dans le meilleur des cas, lorsque l'on peut allouer une mémoire tampon suffisante, il y a au plus `(fin - debut) - 1` comparaisons.

Voici deux exemples illustrant ces fonctions :

```
int A[] = { 1, 3, 5, 7, 9, 2, 4, 6, 8 };
std::inplace_merge(A, A + 5, A + 9);
std::copy(A, A + 9, std::ostream_iterator<int>
          ➥(std::cout, " "));
// La sortie est "1 2 3 4 5 6 7 8"
```

```
int B[] = { 1, 3, 5, 7 };
int C[] = { 2, 4, 6, 8, 9 };
std:: merge(B, B + 4, C, C+5, std::ostream_iterator
           ➥<int>(std::cout, " "));
// La sortie est "1 2 3 4 5 6 7 8 9"
```

Récupérer le plus petit/grand élément

```
#include <algorithm>
const T& min(const T& x, const T& y);
const T& min(const T& x, const T& y,
➥ BinaryPredicate comp);
```

```
const T& max(const T& x, const T& y);
const T& max(const T& x, const T& y,
➥ BinaryPredicate comp);
```

Comme l'indique le nom très explicite de ces fonctions, `min()` et `max()` retournent le minimum ou le maximum entre deux valeurs `x` et `y` données en utilisant l'opérateur `<`, ou le prédicat de comparaison `comp` fourni. Pour illustrer ces fonctions, l'exemple suivant parle de lui-même :

```
std::cout << "Le maximum entre 1 et 99 est "
        ➥<< std::max(1, 99) << std::endl;
std::cout << "Le minimum entre 'a' et 'c' "
        ➥<< std::max('a', 'c') << std::endl;
```

Info

Cet algorithme est plus puissant qu'une macro. Il évite en effet dans certains cas d'effectuer des calculs inutiles. C'est le cas par exemple lorsque vous souhaitez obtenir le maximum entre le résultat de deux appels de fonction, comme `max(sqrt(3), log(27))`.

Récupérer le plus petit/grand élément d'une séquence

```
#include <algorithm>
ForwardIterator min_element(ForwardIterator debut,
➥ ForwardIterator fin);
ForwardIterator min_element (ForwardIterator debut,
➥ ForwardIterator fin, BinaryPredicate comp);
```

```
ForwardIterator max_element(ForwardIterator debut,
➥ ForwardIterator fin);
ForwardIterator max_element (ForwardIterator debut,
➥ ForwardIterator fin, BinaryPredicate comp);
```

`min_element()` retourne le plus petit élément de la séquence [`debut`, `fin`[. Plus exactement, il retourne le premier itérateur `i` appartenant à la séquence donnée tel qu'aucun autre itérateur de cette séquence pointe vers un élément plus grand que `*i`. Cette fonction retourne `fin` uniquement si la séquence donnée est une séquence vide.

La deuxième version de `min_element()` diffère de la première en ce qu'elle n'utilise pas l'opérateur `<` mais le prédicat binaire `comp` pour comparer les éléments. Ainsi l'itérateur `i` retourné vérifie la condition `comp(*j, *i) == false` pour tout `j` (autre que `i` lui-même) de la séquence.

`max_element()` fait la même chose mais renvoie le plus grand élément. L'itérateur `i` retourné vérifie le test `(*i < *j) == false` ou `comp(*i, *j) == false`.

```
int tableau[] = { 5, 6, 2, 8, 5, 8, 0 };
int N = sizeof(tableau) / sizeof(int);
int* i = std::max_element(tableau, tableau+N);
std::cout << "Le plus grand est " << *i << " à la "
➥ << i-tableau+1 << "-ème position.\n";
```

Cet exemple produit la sortie suivante :

```
Le plus grand est 8 à la 4-ème position.
```

Trouver le premier endroit où deux séquences diffèrent

```
#include <algoritmh>
std::pair<InputIterator1, InputIterator2>
➥ mismatch(InputIterator1 debut1, InputIterator1
➥ fin1, InputIterator2 debut2);
std::pair<InputIterator1, InputIterator2>
➥ mismatch(InputIterator1 debut1, InputIterator1
➥ fin1, InputIterator2 debut2, BinaryPredicate comp);
```

`mismatch()` compare les éléments de la première séquence avec ceux de la deuxième, que l'on suppose de même taille (sinon les éléments supplémentaires sont ignorés), et renvoie le premier endroit où les éléments diffèrent. Les éléments sont comparés, soit avec l'opérateur ==, soit avec le prédicat de comparaison `comp` fourni.

```
int A1[] = { 3, 1, 3, 5, 3, 7, 8 };
int A2[] = { 3, 1, 3, 5, 4, 9, 2 };
int N = sizeof(A1) / sizeof(int);
std::pair<int*, int*> res = std::mismatch(A1, A1+N, A2);
std::cout << "La première différence se situe à
➥ l'indice " << res.first - A1 << std::endl
➥ << "Les valeurs sont : " << *(res.first) << " et "
➥ << *(res.second) << std::endl;
```

Générer la prochaine plus petite/grande permutation lexicographique d'une séquence

```
#include <algorithm>
bool next_permutation(BidirectionalIterator debut,
➥ BinirectionalIterator fin);
bool next_permutation(BidirectionalIterator debut,
➥ BinirectionalIterator fin, StrictWeakOrdering comp);
```

```
bool prev_permutation(BidirectionalIterator debut,
➥ BinirectionalIterator fin);
bool prev_permutation(BidirectionalIterator debut,
➥ BinirectionalIterator fin, StrictWeakOrdering comp);
```

`next_permutation()` transforme la séquence [`debut`, `fin`[donnée en la prochaine plus grande permutation, lexicographiquement parlant. Il y a un nombre fini de permutations distinctes de N éléments : au plus N! (factoriel N). Ainsi, si ces permutations sont ordonnées en ordre lexicographique, il y a une définition non ambiguë de ce que

signifie la prochaine plus grande permutation. Donc, si une telle permutation existe, `next_permutation()` transforme [`debut`, `fin`[en celle-ci et renvoie vrai, sinon il la transforme en la plus petite et renvoie faux.

Si une relation d'ordre stricte `comp` est fournie, elle est utilisée comme comparaison d'élément pour définition de l'ordre lexicographique à la place de l'opérateur `<`.

`prev_permutation()` est l'inverse de `next_permutation()`. Elle transforme la séquence en la précédente permutation, selon le même critère que précédemment.

```
template<class BidiIter>
void le_pire_des_tris(BidiIter deb, BidiIter fin)
{
    while (std::next_permutation(deb, fin)) { }
}

int main(int, char**)
{
    int A[] = { 8, 3, 6, 1, 2, 5, 7, 4 };
    int N = sizeof(A) / sizeof(int);
    le_pire_des_tris(A, A+N);
    std::copy(A, A+N, std::ostream_iterator<int>(std::
    ➥ cout," "));
    return 0;
}
```

Cet exemple utilise `next_permutation()` pour implémenter la pire des manières de faire un tri. La plupart des algorithmes de tri sont en $O(n \log n)$, et même le tri à bulles est seulement en $O(n^2)$. Celui-ci est en $O(n!)$.

Faire en sorte que le nième élément soit le même que si la séquence était triée

```
#include <algorithm>
void nth_element(RandomAccesIterator debut,
➥ RandomAccesIterator n_ieme, RandomAccesIterator
➥ fin);
void nth_element(RandomAccesIterator debut,
➥ RandomAccesIterator n_ieme, RandomAccesIterator
➥ fin, StrickWeakOrdering comp);
```

`nth_element()` est comparable à `partial_sort()` en ce sens qu'elle ordonne une partie des éléments de la séquence. Elle arrange la séquence donnée de telle sorte que le `n_ieme` itérateur pointe sur le même élément que si la séquence avait été triée complètement. De plus, l'arrangement est tel qu'aucun élément de la séquence [`n_ieme`, `fin`[soit plus petit qu'un des éléments de la séquence [`debut`, `n_ieme`[.

Encore une fois, cette fonction utilise soit l'opérateur `<`, soit la relation d'ordre stricte `comp` fournie pour ordonner les éléments de la séquence [`debut`, `fin`[.

Info

`nth_element()` diffère de `partial_sort()` en ce sens qu'aucune des sous-séquences gauche et droite ne sont triées. Elle garantit seulement que tous les éléments de la partie gauche sont plus petits que ceux de la partie droite. En ce sens, `nth_element()` est plus comparable à une partition qu'à un tri. Elle fait moins que `partial_sort()` et est donc également plus rapide. Pour cette raison, il vaut mieux utiliser `nth_element()` plutôt que `partial_sort()` lorsqu'elle est suffisante pour vos besoins.

```
int A[] = { 7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5 };
const int N = sizeof(A) / sizeof(int);
std::nth_element(A, A+6, A+N);
std::copy(A, A+6, std::ostream_iterator<int>( std::
➥ cout," "));
std::cout << "/ ";
std::copy(A+6, A+N, std::ostream_iterator<int>(
➥ std::cout," "));
// Affichera : "5 2 6 1 4 3 / 7 8 9 10 11 12"
```

Trier les *n* premiers éléments d'une séquence

```
#include <algorithm>
void partial_sort(RandomAccessIterator debut,
➥ RandomAccessIterator milieu, RandomAccesIterator
➥ fin);
void partial_sort(RandomAccessIterator debut,
➥ RandomAccessIterator milieu, RandomAccesIterator
➥ fin, StrictWeakOrdering comp);
```

`partial_sort()` réordonne les éléments de la séquence [`debut`, `fin`[de telle sorte qu'ils soient partiellement dans l'ordre croissant. Il place les `milieu-debut` plus petits éléments, dans l'ordre croissant, dans la séquence [`debut`, `milieu`[. Les `fin-milieu` éléments restants se retrouvent, sans ordre particulier, dans la séquence [`milieu`, `fin`[.

Cette fonction utilise la relation d'ordre partielle `comp` fournie, sinon c'est l'opérateur < qui est utilisé.

Cet algorithme effectue environ `(fin - debut) * log( middle - first)` comparaisons.

Info

`partial_sort(debut, fin, fin)` produit le même résultat que `sort(debut, fin)`. Notez toutefois que l'algorithme utilisé n'est pas le même : `sort()` utilise *introsort*, une variante de *quicksort*, alors que `partial_sort()` utilise un *heapsort* (tri par tas). Même si ces deux types de tris ont une complexité analogue, en `O(N log N)`, celui de `sort()` est 2 à 5 fois plus rapide que celui de `partial_sort()`. Ainsi, pour trier la totalité d'une séquence, préférez `sort()` à `partial_sort()`.

Voici un petit exemple pour visualiser l'effet de cet algorithme :

```
int A[] = { 7, 2, 6, 13, 9, 3, 14, 11, 8, 4, 1, 5 };
const int N = sizeof(A) / sizeof(int);
std::partial_sort(A, A + 5, A+ N);
std::copy(A, A+N, std::ostream_iterator<int>(std::cout,
➥ " "));
// Donne : "1 2 3 4 5 13 14 11, 9, 8, 7, 6".
```

Copier les *n* plus petits éléments d'une séquence

```
#include <algorithm>
RandomAccessIterator partial_sort_copy
➥(RandomAccessIterator debut, RandomAccesIterator fin,
➥ RandomAccessIterator debut_resultat,
➥ RandomAccesIterator fin_resultat);
RandomAccessIterator partial_sort_copy
➥(RandomAccessIterator debut, RandomAccesIterator
➥ fin, RandomAccessIterator debut_resultat,
➥ RandomAccesIterator fin_resultat,
➥ StrictWeakOrdering comp);
```

`partial_sort_copy()` copie les `N` plus petits éléments de la séquence [`debut`, `fin`[dans la séquence [`debut_resultat`, `fin_resultat`[, où `N` est la taille de la plus petite des deux séquences données. Les éléments de la séquence résultat sont bien sûr ordonnés, suivant la relation d'ordre `comp` fournie ou `<` le cas échéant. Cette fonction retourne ensuite l'itérateur `debut_resultat + N`.

```
int A[] = { 7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5 };
const int N = sizeof(A) / sizeof(int);
std::vector<int> V(4);
std::partial_sort_copy(A, A+N, V.begin(), V.end());
std::copy(V.begin(), V.end(), std::ostream_iterator
➥ <int>(std::cout," "));
```

L'exemple précédent produit le résultat suivant :

```
1 2 3 4
```

Calculer une somme partielle généralisée d'une séquence

```
#include <numeric>
OutputIterator partial_sum(InputIterator debut,
➥ InputIterator fin, OutputIterator res);
OutputIterator partial_sum(InputIterator debut,
➥ InputIterator fin, OutputIterator res,
➥ BinaryOperation op);
```

`partial_sum()` calcule une somme partielle généralisée. Pour faire simple, c'est un peu l'équivalent du code suivant :

```
res[0] = debut[0];
for (int i=1 ; i < fin-debut ; i++)
{
    res[i] = res[i-1] + debut[i];
    // ou res[i] = op(res[i-1], debut[i]);
}
```

Cette fonction retourne l'itérateur `res+(fin-debut)`.

Info

`res` peut avoir la même valeur que `debut`. Cela peut être utile lorsque vous voulez stocker le résultat directement dans la séquence d'origine.

De plus, étant donné que l'ordre des opérations est totalement défini, il n'est pas nécessaire que l'opérateur `op` que vous fournissez soit associatif ou commutatif.

Couper la séquence en deux en fonction d'un prédicat

```
#include <algorithm>
ForwardIterator partition(ForwardIterator debut,
➥ ForwardIterator fin, Predicate pred);
ForwardIterator stable_partition(ForwardIterator
➥ debut, ForwardIterator fin, Predicate pred);
```

`partition()` réordonne les éléments de la séquence [`debut`,`fin`[de telle sorte qu'il existe un itérateur `milieu` appartenant

à la séquence tel que : les éléments de la sous-séquence [debut, milieu[vérifient pred(*i)==true, et ceux de la sous-séquence [milieu, fin[vérifient pred(*i)==false. La fonction retourne cet itérateur milieu.

stable_partition() diffère de partition() en ce sens qu'elle préserve l'ordre relatif des éléments. Ainsi, si x et y sont des éléments tels que pred(x) == pred(y) et que x précède y, alors x précédera toujours y après l'exécution de stable_sort().

Astuce

Les fonctions stables sont toujours plus lentes que les fonctions dont la stabilité (au sens de la préservation de l'ordre des éléments et non de la présence de bogues, bien sûr) n'est pas garantie. Ne les utilisez donc *que* si cette propriété de stabilité vous est nécessaire.

L'exemple ci-après montre comment séparer les nombres pairs et les nombres impairs d'une séquence. Les nombres pairs sont ici placés au début de la séquence.

```
std::vector<int> A(10);
for (int i=0 ; i<10 ; i++) A[i] = i+1;
std::partition(A.begin(), A.end(),
  ➥std::compose1(std::bind2nd( std::equal_to<int>(), 0),
                  ➥std::bind2nd( std::modulus<int>(), 2)));
// La séquence contient alors les valeurs :
// 10 2 8 4 6 5 7 3 9 1.
```

Calculer x^i (fonction puissance généralisée)

```
#include <numeric>
T power(T x, Integer n);
T power(T x, Integer n, MonoidOperation op);
```

`power()` est une généralisation de la fonction puissance x^n, où n est un entier positif.

La première version retourne `x *x *x … * x`, où `x` est répété `n` fois. Si `n == 0`, alors elle retourne `std::identity( std::multiplies<T>() )`.

La deuxième version est identique à l'exception qu'elle utilise la fonction `op` au lieu de `multiplies<T>` et retourne `std::identity(op)` si `n == 0`.

Attention

`power()` ne repose pas sur le fait que la multiplication est commutative, mais qu'elle est associative. Si vous définissez un opérateur `*` ou une opération `op` qui n'est pas associatif, alors `power()` donnera *une mauvaise réponse*.

Info

Une *monoid operation* est une fonction binaire particulière. Une fonction binaire doit satisfaire trois conditions pour être une opération monoïde. Premièrement, le type de ses deux arguments et de sa valeur de retour doivent être les mêmes. Deuxièmement, il doit exister un élément identité. Troisièmement, l'opération doit être associative. Par exemple, l'addition et la multiplication sont des opérations monoïdes.

L'associativité implique que `f(x, f(y,z)) == f(f(x,y), z)`.

L'élément identité `id` respecte les conditions `f(x, id) == x` et `f(id, x) == x`.

Info

power() fait partie des extensions SGI et peut être codée dans un autre espace de noms que std, si elle est présente dans l'implémentation de votre compilateur.

Copier aléatoirement un échantillon d'une séquence

```
#include <algorithm>
RandomAccessIterator random_sample(InputIterator
➥ debut,InputIterator fin, RandomAccessIterator
➥ rDebut, RandomAccesIterator rFin);
RandomAccessIterator random_sample(InputIterator
➥ debut, InputIterator fin, RandomAccessIterator
➥ rDebut, RandomAccesIterator rFin,
➥ RandomNumberGenerator& rand);
```

random_sample() copie aléatoirement un échantillon de la séquence [debut, fin[dans la séquence [rDebut, rFin[. Chaque élément de la séquence d'entrée apparaîtra au plus une fois dans celle de sortie. Les éléments sont tirés selon une loi de probabilité uniforme. L'ordre relatif des éléments de la séquence d'entrée n'est pas préservé (si vous en avez besoin, jetez un œil sur random_sample_n()).

Le nombre n d'éléments copiés est le minimum entre fin - debut et rFin - rDebut. Du coup, la fonction renvoie l'itérateur rDebut + n.

Attention

Si vous spécifiez un générateur de nombres aléatoires en paramètre, celui-ci doit produire une distribution uniforme. C'est-à-dire que la fréquence d'apparition de chaque valeur doit être la même.

```
const int N = 10;
const int n = 4;
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int B[n];

std::random_sample(A, A+N, B, B+n);
std::copy(B, B+n, std::ostream_iterator<int>(std::cout,
➥" "));
```

Le code précédent affiche une des 5 039 possibilités existantes, comme celle-ci :

```
7 1 5 2
```

Copier aléatoirement un sous-échantillon (de *n* éléments), en préservant leur ordre d'origine

```
#include <algorithm>
OutpuIterator random_sample_n(ForwardIterator debut,
➥ ForwardIterator fin, OutpuIterator res, Distance n);
OutpuIterator random_sample_n(ForwardIterator debut,
➥ ForwardIterator fin, OutpuIterator res, Distance n,
➥ RandomNumberGenerator& rand);
```

`random_sample_n()` copie aléatoirement un sous-échantillon (de `n` éléments s'il y en a assez) de la séquence [`debut`, `fin`[dans la séquence [`res`, `res+n`[. Chaque élément de la séquence d'entrée apparaîtra au plus une fois dans celle de sortie. Les éléments sont tirés selon une loi de probabilité uniforme. L'ordre relatif des éléments de la séquence d'entrée est préservé (si vous ne le voulez pas, jetez un œil sur `random_sample()`).

En réalité, le nombre `m` d'éléments copiés est le minimum entre `fin - debut` et `n`. Du coup, la fonction renvoie l'itérateur `res + m`.

Comme avec `random_sample()`, vous pouvez fournir votre propre générateur de nombres aléatoires.

```
const int N = 10;
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::random_sample_n(A, A+N, std::ostream_iterator
➥ <int>(std::cout, " "), 4);
```

Le code précédent affiche une des 209 possibilités existantes, comme celle-ci :

```
1 2 5 7
```

Mélanger les éléments d'une séquence

```
#include <algorithm>
void random_shuffle(RandomAccesIterator debut,
➥ RandomAccesIterator fin);
void random_shuffle(RandomAccesIterator debut,
➥ RandomAccesIterator fin, RandomNumberGenerator&
➥ rand);
```

`random_shuffle()` mélange aléatoirement les éléments de la séquence [`debut`, `fin`[. Cette séquence se retrouve alors dans un des $N! - 1$ réagencements possibles, où N est la taille de la séquence. Une fois encore, vous avez la possibilité d'utiliser votre propre générateur de nombres aléatoires.

```
const int N = 10;
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::random_shuffle(A, A+N);
std::copy(A, A+N, std::ostream_iterator<int>
➥(std::cout, " "));
```

Le code précédent affiche une des 3 628 899 (soit *10! – 1*) autres possibilités existantes, comme celle-ci :

```
9 8 3 1 6 5 2 4 10 7
```

Supprimer certains éléments d'une séquence

```
#include <algorithm>
ForwardIterator remove(ForwardIterator deb,
➥ ForwardIterator fin, const T& valeur);
ForwardIterator remove_if(ForwardIterator deb,
➥ ForwardIterator fin, Predicate test);
```

`remove()` supprime de la séquence [`debut`, `fin`[tous les éléments égaux à la `valeur` donnée, puis retourne le nouvel itérateur de fin. Cette fonction est stable : elle préserve l'ordre relatif des éléments conservés.

`remove_if()` fait de même en supprimant les éléments vérifiant le `test` donné.

Attention

Le sens du mot « supprimer » est quelque peu déformé dans le contexte de ces fonctions. `remove()` **et** `remove_if()` **ne détruisent aucun itérateur. Ainsi la distance entre** `deb` **et** `fin` **restera inchangée. Par exemple, si la séquence** `V` **donnée est un vecteur (**`std::vector<>`**), alors** `V.size()` **retournera la même taille avant et après la suppression.**

L'itérateur renvoyé indique donc que les éléments qui le suivent sont sans intérêt, et les valeurs des éléments sur lesquels ils pointent ne sont pas garanties.

Si vous voulez réellement supprimer les éléments d'une séquence `S`**, vous pouvez utiliser une formule de ce type :** `S.erase (std::remove(S.begin, S.end(), x), S.end())`**.**

```
std::vector<int>::iterator fin2;
std::vector<int> V;
V.push_back(3);
V.push_back(5);
V.push_back(2);
V.push_back(3);
V.push_back(8);
V.push_back(7);
fin2 = std::remove(V.begin(), V.end(), 3);
std::copy(V.begin(), fin2, std::ostream_iterator <int>
➥ (std::cout," "));
// Affiche : 5 2 8 7
```

Copier une séquence en omettant certains éléments

```
#include <algorithm>
OutputIterator remove_copy(ForwardIterator deb,
➥ ForwardIterator fin, OutputIterator res, const T&
➥ valeur);
OutputIterator remove_copy_if(ForwardIterator deb,
➥ ForwardIterator fin, OutputIterator res, Predicate
➥ test);
```

`remove_copy()` copie tous les éléments non égaux à la `valeur` donnée de la séquence [`debut`, `fin`[dans la séquence commençant à l'itérateur `res`. Cette fonction est stable : elle préserve l'ordre relatif des éléments conservés.

`remove_copy_if()` fait de même en copiant les éléments ne vérifiant pas le `test` donné.

Attention

La séquence résultat doit être assez grande pour contenir l'ensemble des éléments qui y seront copiés. N'hésitez pas à faire usage des `std::back_insert_iterator`, `std::front_insert_iterator` ou autre « utilitaire » dans ce contexte.

```
std::vector<int> V;
V.push_back(3);
V.push_back(5);
V.push_back(2);
V.push_back(3);
V.push_back(8);
V.push_back(7);
std::remove_copy_if(V.begin(), V.end(),
    ➥std::ostream_iterator<int>(std::cout," "),
    ➥std::bind2nd(std::less_equal<int>(),3) );
// Affiche : 5 8 7
```

Remplacer certains éléments d'une séquence

```
#include <algorithm>
void replace(ForwardIterator deb, ForwardIterator fin,
➥ const T& ancienne_valeur, const T& nouvelle_valeur);
void replace_if(ForwardIterator deb, ForwardIterator
➥ fin, Predicate test, const T& nouvelle_valeur);
```

```
OutputIterator replace_copy(InputIterator deb,
➥ InputIterator fin, OutputIterator res, const T&
➥ ancienne_valeur, const T& nouvelle_valeur);
OutputIterator replace_copy_if(InputIterator deb,
➥ InputIterator fin, OutputIterator res, Predicate
➥ test, const T& nouvelle_valeur);
```

`replace()` remplace tous les éléments de [`deb`, `fin`[égaux à l'`ancienne_valeur` par la `nouvelle_valeur`. Ainsi tous les itérateurs `i` tels que `*i == ancienne_valeur` alors `*i = nouvelle_valeur`.

`replace_if()` remplace les éléments vérifiant le prédicat `test(*i) == true`.

`replace_copy()` et `replace_copy_if()` procède de même mais effectue les remplacements sur (et pendant) la copie.

```
std::vector<int> V;
V.push_back(3);
V.push_back(5);
V.push_back(2);
V.push_back(3);
V.push_back(8);
V.push_back(7);
std::list<int> V2;
std::replace_copy_if(V.begin(), V.end(),
 ➥ std::front_insert_iterator< std::list<int> >(V2),
 ➥ std::bind2nd(std::less_equal<int>(),3), 0 );
```

```
std::copy(V2.begin(), V2.end(),
   ➥ std::ostream_iterator<int>(std::cout," "));
// Affiche : 7 8 0 2 5 0
```

Inverser l'ordre de la séquence

```
#include <algorithm>
void reverse(BidirectionalIterator deb,
➥ BidirectionalIterator fin);
OutputIterator reverse_copy(BidirectionalIterator
➥ deb, BidirectionalIterator fin, OutputIterator res);
```

`reverse()` inverse l'ordre des éléments de la séquence. `reverse_copy()` copie la séquence en inversant l'ordre lors de la copie.

Si `A` est un vecteur contenant les éléments 1, 2, 3, 4, 5, alors le code suivant inverse l'ordre de ses éléments de sorte qu'il contienne 5, 4, 3, 2, 1.

```
std::reverse(A.begin(), A.end());
```

Effectuer une rotation des éléments de la séquence

```
#include <algorithm>
ForwardIterator rotate(ForwardIterator debut,
➥ ForwardIterator milieu, ForwardIterator fin);
OutputIterator rotate_copy(ForwardIterator debut,
➥ ForwardIterator milieu, ForwardIterator fin,
➥ OutputIterator res);
```

`rotate()` effectue une rotation des éléments de la séquence. Ainsi, l'élément à la position `milieu` est déplacé à la position `debut`, l'élément à la position `milieu + 1` est déplacé à la position `debut + 1`, et ainsi de suite. Une autre manière de voir les choses est de dire que cette fonction échange les deux sous-séquences [`debut`, `milieu`[et [`milieu`, `fin`[. Enfin, cette fonction retourne l'équivalent de `debut + (fin - milieu)`, la nouvelle position du premier élément de la séquence d'entrée.

Info

Certaines implémentations de `rotate()` retournent `void`.

`rotate_copy()` copie le résultat dans la séquence commençant à la position `res` plutôt que d'écraser la séquence d'origine, puis retourne `result + (last - first)`.

```
char alphabet[] = "abcdefghijklmnopqrstuvwxyz";
std::rotate(alphabet, alphabet + 5, alphabet + 26);
std::cout << alphabet << std::endl;
// Affichera : fghijklmnopqrstuvwxyzabcde
```

Chercher une sous-séquence

```
#include <algorithm>
ForwardIterator1 search(ForwardIterator1 debut1,
➥ ForwardIterator1 fin1, ForwardIterator2 debut2,
➥ ForwardIterator2 fin2);
ForwardIterator1 search(ForwardIterator1 debut1,
➥ ForwardIterator1 fin1, ForwardIterator2 debut2,
➥ ForwardIterator2 fin2, BinaryPredicate comp);
```

```
ForwardIterator search_n(ForwardIterator debut,
➥ ForwardIterator fin, Size n, const T& valeur);
ForwardIterator search_n(ForwardIterator debut,
➥ ForwardIterator fin, Size n, const T& valeur,
➥ BinaryPredicate comp);
```

```
ForwardIterator1 find_end(ForwardIterator1 debut1,
➥ ForwardIterator1 fin1, ForwardIterator2 debut2,
➥ ForwardIterator2 fin2);
ForwardIterator1 find_end(ForwardIterator1 debut1,
➥ ForwardIterator1 fin1, ForwardIterator2 debut2,
➥ ForwardIterator2 fin2, BinaryPredicate comp);
```

La fonction `search()` cherche la première sous-séquence identique à [`debut2`, `fin2`[apparaissant dans [`debut1`, `fin1`[.

La fonction `search_n()` cherche la première sous-séquence de `n` occurrences consécutives de `valeur` dans [`debut1`, `fin1`[.

La fonction `find_end()` porte mal son nom et aurait dû s'appeler `search_end()` car son comportement est plus proche de `search()` que de `find()`. Comme `search()`, elle cherche une sous-séquence de [`debut1`, `fin1`[qui soit identique à [`debut2`, `fin2`[. La différence tient au fait que `search()` cherche la première occurrence, alors que `find_end()` cherche la dernière. Si elle existe, `find_end()` retourne un itérateur sur le début de la sous-séquence trouvée ; sinon, elle retourne `fin1`.

La première version de `find_end()` utilise l'opérateur `==`. La deuxième le prédicat `comp` donné en testant si `comp( *(i + (j-debut2)), *j)` est vrai pour un `i` donné de la première séquence et pour tout `j` de la séquence cherchée. Le même principe est valable pour `search()` et `search_n()`.

Le programme suivant illustre l'utilisation de ces différentes fonctions :

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

bool ci_equal(char ch1, char ch2)
{
    return toupper((unsigned char)ch1)
     ➥ == toupper((unsigned char)ch2);
}

int main(int, char**)
{
    string s = "Exeexecutable.exe";
    string t = "exe";

    string::iterator i;
    i = find_end(s.begin(), s.end(), t.begin(), t.end());

    if (i != s.end())
    {
        cout << "find_end    => Trouvé à position ";
        cout << (i-s.begin()) << " : '";
        copy(i,i+t.size(),ostream_iterator<char>(cout));
        cout << "'" << endl;
    }

    i = search(s.begin(), s.end(), t.begin(), t.end());
    if (i != s.end())
    {
        cout << "search      => Trouvé à position ";
        cout << (i-s.begin()) << " : '";
        copy(i,i+t.size(),ostream_iterator<char>(cout));
        cout << "'" << endl;
    }

    i = search(s.begin(), s.end(), t.begin(), t.end(),
              ➥ci_equal);
    if (i != s.end())
```

```
    {
        cout << "search+pred => Trouvé à position ";
        cout  << (i-s.begin()) << " : '";
        copy(i,i+t.size(),ostream_iterator<char>(cout));
        cout << "'" << endl;
    }

    i = search_n(s.begin(), s.end(), 2, 'e', ci_equal);
    if (i != s.end())
    {
        cout << "search_n    => Trouvé à position  ";
        cout << (i-s.begin()) << " : '";
        copy(i,i+2,ostream_iterator<char>(cout));
        cout << "'" << endl;
    }

    return 0;
}
```

Il produit la sortie suivante :

```
find_end    => Trouvé à position 14 : 'exe'
search      => Trouvé à position 3 :  'exe'
search+pred => Trouvé à position 0 :  'Exe'
search_n    => Trouvé à position 2 :  'ee'
```

Construire la différence de deux séquences triées

```
#include <algorithm>
OutputIterator set_difference(InputIterator1 debut1,
➥ InputIterator1 fint1, InputIterator2 debut2,
➥ InputIterator2 fin2, OutputIterator res);
OutputIterator set_difference(InputIterator1 debut1,
➥ InputIterator1 fin1, InputIterator2 debut2,
➥ InputIterator2 fin2, OutputIterator res,
➥ StrictWeakOrdering comp);
```

`set_difference()` construit la différence des deux séquences triées [`debut1`, `fin1`[et [`debut2`, `fin2`[. Le résultat est une séquence triée commençant à l'itérateur `res` donné et finissant à l'itérateur retourné par la fonction.

D'une manière simple, cette différence correspond à celle de la théorie des ensembles. Le résultat est l'ensemble des éléments de [`debut1`, `fin1`[qui ne sont pas dans [`debut2`, `fin2`[. La réalité est un peu plus complexe : des éléments peuvent apparaître plusieurs fois dans chacune des séquences. Dans ce cas, si un élément apparaît `m` fois dans la première séquence et `n` fois dans la deuxième, alors il sera `max(m-n, 0)` fois dans la séquence résultat.

Info

Cette fonction est stable car elle préserve aussi l'ordre d'apparition des éléments considérés comme identiques.

La deuxième version de `set_difference()` utilise la relation d'ordre donnée au lieu de l'opérateur <.

```
inline bool lt_nocase(char c1, char c2)
{
    return std::tolower(c1) < std::tolower(c2);
}

int main()
{
    int A1[] = {1, 3, 5, 7, 9, 11};
    int A2[] = {1, 1, 2, 3, 5, 8, 13};
    char A3[] = {'a', 'b', 'b', 'B', 'B', 'f', 'g', 'h',
    ➥ 'H'};
    char A4[] = {'A', 'B', 'B', 'C', 'D', 'F', 'F', 'H' };

    const int N1 = sizeof(A1) / sizeof(int);
```

```
    const int N2 = sizeof(A2) / sizeof(int);
    const int N3 = sizeof(A3);
    const int N4 = sizeof(A4);

    std::cout << "Difference A1 - A2 : ";
    std::set_difference(A1, A1 + N1, A2, A2 + N2,
        ➥std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl << "Difference A3 - A4 : ";
    std::set_difference(A3, A3 + N3, A4, A4 + N4,
        ➥std::ostream_iterator<char>(std::cout, " "),
        ➥lt_nocase);
    std::cout << std::endl;
}
```

Cet exemple produira la sortie ci-après. Notez bien la manière dont elle conserve les éléments : *ce sont les premières occurrences identiques qui sont supprimées*.

```
Difference A1 - A2 : 7 9 11
Difference A3 - A4 : B B g H
```

Construire l'intersection de deux séquences triées

```
#include <algorithm>
OutputIterator set_intersection(InputIterator1 debut1,
➥ InputIterator1 fint1, InputIterator2 debut2,
➥ InputIterator2 fin2, OutputIterator res);
OutputIterator set_intersection(InputIterator1 debut1,
➥ InputIterator1 fin1, InputIterator2 debut2,
➥ InputIterator2 fin2, OutputIterator res,
➥ StrictWeakOrdering comp);
```

`set_intersection()` construit l'intersection des deux séquence triées [`debut1`, `fin1`[et [`debut2`, `fin2`[. Le résultat est une séquence triée commençant à l'itérateur `res` donné et finissant à l'itérateur retourné par la fonction.

Cette intersection correspond à celle de la théorie des ensembles. Le résultat est l'ensemble des éléments qui sont à la fois des éléments de [`debut1`, `fin1`[et de [`debut2`, `fin2`[. La réalité est un peu plus complexe : des éléments peuvent apparaître plusieurs fois dans chacune des séquences. Dans ce cas, si un élément apparaît `m` fois dans la première séquence et `n` fois dans la deuxième, alors il sera `min(m,-n)` fois dans la séquence résultat.

Info

Cette fonction est stable car elle préserve aussi l'ordre d'apparition des éléments considérés comme identiques.

La deuxième version de `set_intersection()` utilise la relation d'ordre donnée au lieu de l'opérateur `<`.

En reprenant les valeurs de l'exemple de `set_difference()` mais en appliquant `set_intersection()`, on obtient le résultat suivant :

```
Intersection de A1 et A2 : 1 3 5
Intersection de A3 et A4 : a b b f h
```

Vous remarquerez que ce sont *les premières apparitions de la première séquence qui sont conservées.*

Construire la différence symétrique des deux séquences triées

```
#include <algorithm>
OutputIterator set_symmetric_difference(InputIterator1
➥ debut1, InputIterator1 fin1, InputIterator2 debut2,
➥ InputIterator2 fin2, OutputIterator res);
OutputIterator set_symmetric_difference(InputIterator1
➥ debut1, InputIterator1 fin1, InputIterator2 debut2,
➥ InputIterator2 fin2, OutputIterator res,
➥ StrictWeakOrdering comp);
```

`set_symmetric_difference()` construit la différence symétrique des deux séquences triées [`debut1`, `fin1`[et [`debut2`, `fin2`[. Cette nouvelle séquence est aussi triée selon l'opérateur `<` ou `comp` fourni. L'itérateur retourné est la fin de cette séquence résultat.

Là encore, dans le cas simple, `set_symmetric_difference()` effectue un calcul de la théorie des ensembles : elle construit l'union des deux ensembles `A - B` et `B - A`, où `A` et `B` sont les deux séquences d'entrée. La séquence résultat `R` contient une copie des éléments de `A` qui ne sont pas dans `B`, et une copie des éléments de `B` qui ne sont pas dans `A`.

Lorsque `A` et/ou `B` contiennent des éléments semblables, la règle suivante est appliquée : si un élément apparaît `m` fois dans `A` et `n` fois dans `B`, alors il apparaît `|m - n|` fois dans `R`.

Cette fonction est stable et préserve l'ordre d'apparition des éléments semblables qui se suivent.

En reprenant encore les valeurs de l'exemple de `set_difference()` mais en appliquant cette fois-ci `set_symmetric_difference()`, on obtient le résultat suivant :

```
Intersection symétrique de A1 et A2 : 1 2 7 8 9 11 13
Intersection symétrique de A3 et A4 : B B C D F g H
```

Si `m > n`, alors les `m - n` derniers éléments de `A` sont conservés. Si `n < m`, alors les `n - m` derniers éléments de `B` sont conservés.

Construire l'union de deux séquences triées

```
#include <algorithm>
OutputIterator set_union(InputIterator1
➥ debut1, InputIterator1 fin1, InputIterator2 debut2,
➥ InputIterator2 fin2, OutputIterator res);
OutputIterator set_union(InputIterator1
➥ debut1, InputIterator1 fin1, InputIterator2 debut2,
➥ InputIterator2 fin2, OutputIterator res,
➥ StrictWeakOrdering comp);
```

`set_union()` construit la séquence triée résultant de l'union des deux séquences triées fournies. Le tri correspond soit à l'opérateur `<`, soit à la relation d'ordre partielle stricte `comp` fournie.

Dans le cas simple, cette fonction correspond à l'union de la théorie des ensembles. Le résultat contient une copie de chaque élément apparaissant dans l'un, l'autre ou les deux séquences données.

Le cas général repose sur cette règle : si un élément apparaît `m` fois dans [`debut1`, `fin1`[et `n` fois dans [`debut2`, `fin2`[, alors il apparaîtra `max(m, n)` fois dans le résultat.

Une fois encore, cette fonction est stable et préserve l'ordre relatif des éléments semblables.

Un `set_union()` sur les valeurs des exemples précédents (que l'on retrouve dans l'exemple de `set_difference()`) produira le résultat suivant :

```
Union de A1 et A2 : 1 1 2 3 5 7 8 9 11 13
Union de A3 et A4 : a b B B C D f F H h
```

Si un élément apparaît `m` fois dans [`debut1`, `fin1`[et `n` fois dans [`debut2`, `fin2`[, alors les `m` éléments de [`debut1`, `fin1`[puis les `max(n-m, 0)` premiers de [`debut2`, `fin2`[seront copiés dans le résultat. Notez que cette information n'est utile que si vous n'utilisez pas une relation d'ordre totale mais partielle ; cas dans lesquels des éléments peuvent être équivalents (ou semblables) pas mais égaux, afin de bien comprendre quels éléments sont conservés.

Trier une séquence

```
#include <algorithm>
bool is_sorted(ForwardIterator debut, ForwardIterator
➥ fin);
bool is_sorted(ForwardIterator debut, ForwardIterator
➥ fin, StreakWeakOrdering comp);
```

```
bool sort(ForwardIterator debut, ForwardIterator fin);
bool sort(ForwardIterator debut, ForwardIterator fin,
➥ StreakWeakOrdering comp);
```

`is_sorted()` teste si la séquence est triée ou non selon la relation d'ordre `comp` (ou `<`). Pour cela, elle teste si deux éléments consécutifs `a` et `b` sont tels que `comp(b,a)` (ou `b<a`) est faux.

`sort()` trie la séquence selon la relation d'ordre donnée.

`stable_sort()` trie également la séquence selon la relation d'ordre donnée, mais préserve l'ordre d'origine des éléments de valeurs équivalentes : c'est un tri stable. Utilisez-le uniquement si cette caractéristique vous est nécessaire (le tri stable est plus lent que le tri simple).

```
int A[] = { 1, 4, 7, 2, 5, 7, 9 };
const int N = sizeof(A) / sizeof(int);
assert( is_sorted(A,A+N) == false );
std::sort(A, A+N);
assert( is_sorted(A,A+N) == true );
```

Attention

Avec g++, `is_sorted()` se trouve dans `<ext/algorithm>` dans l'espace de noms `__gnu_cxx`.

Échanger le contenu de deux variables, itérateurs ou séquences

```
#include <algorithm>
void swap(Assignable& a, Assignable& b);
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
void swap_ranges(ForwardIterator1 debut1,
➥ ForwardIterator1 fin1, ForwardIterator2 debut2,
➥ ForwardIterator2 fin2);
```

`swap()` échange le contenu de deux variables.

`iter_swap()` est équivalent à `swap(*a, *b)`. Strictement parlant, cette fonction est redondante et serait donc inutile.

Sa présence est due à une raison technique : certains compilateurs ont parfois du mal à déduire le type requis pour appeler la bonne instanciation de `swap(*a, *b)`.

`swap_range()` échange le contenu de deux séquences de même taille. Les deux séquences ne doivent pas avoir de partie commune.

```
std::vector<int> A,B;
A.push_back(1);
A.push_back(2); // A = { 1, 2 }
B.push_back(3);
B.push_back(4); // B = { 3, 4 }

std::swap(A[0],A[1]);
// A = { 2, 1 } et B = { 3, 4 }
std::iter_swap(A.begin(), B.begin());
// A = { 3, 1 } et B = { 2, 4 }
std::swap_ranges(A.begin(), A.end(), B.begin(), B.end());
// A = { 2, 4 } et B = { 3, 1 }
```

Transformer une (ou deux) séquences en une autre

```
#include <algorithm>
OutputIterator transform(InputIterator debut,
➥ InputIterator fin, OutputIterator res,
➥ UnaryFunction op);
OutputIterator transform(InputIterator1 debut1,
➥ InputIterator1 fin1, InputIterator2 debut2,
➥ OutputIterator res, BinaryFunction op_binaire);
```

Par définition, `transform()` effectue une opération sur des objets. La première version utilise une séquence, la deuxième en utilise deux.

Dans le premier cas, `transform()` applique le foncteur à chaque objet et l'affecte à l'itérateur de sortie : `*o = op(*i)` puis incrémente celui-ci. Enfin elle retourne l'itérateur correspondant à la fin de la séquence de sortie.

Dans le deuxième cas, `transform()` affecte à l'itérateur de sortie le résultat de `op_binaire(*i1,*i2)`, en faisant progresser simultanément (parallèlement) `i1` et `i2` sur chacune des deux séquences fournies.

Attention

On est vite tenté d'utiliser `for_each` à la place de `transform` pour transformer une séquence. Cela s'avère tout à fait possible dans la pratique mais doit *absolument être évité*, car cela romprait la sémantique de `for_each()` qui est qu'elle ne modifie jamais une séquence.

Dans ce cas, utiliser `transform()` en utilisant le même itérateur pour `debut` et `res`. Il serait d'ailleurs agréable que soit ajoutée à la STL à cette fin une fonction `transform()` ne prenant pas d'itérateur `res`. Sinon, il est facile de l'ajouter vous-même. Voici une possibilité :

```
namespace std
{
    template <class Conteneur1, class Conteneur2,
    ➥ class UnaryFunction>

inline void transform_all(Conteneur1 c, Conteneur2 r,
➥ UnaryFunction op)
    {
        transform(c.begin(), c.end(), r.begin(), op);
    }
}
```

L'exemple suivant calcule la somme de deux vecteurs `V1` et `V2` de même taille (bien sûr).

```
std::transform(V1.begin(), V1.end(), V2.begin(),
➥ V1plusV2.begin(), std::plus<int>());
```

Et celui-ci transforme un vecteur de réels en son opposé :

```
std::transform(V1.begin(), V1.end(),V1.begin(),
➥ std::negate<double>());
```

Attention

L'itérateur de sortie `res` ne doit jamais faire partie de la séquence d'entrée, à l'exception de `debut` lui-même, sous peine d'obtenir des résultats imprévisibles.

Supprimer les doublons d'une séquence (dans ou lors d'une copie)

```
#include <algorithm>
ForwardIterator unique(ForwardIterator debut,
➥ ForwardIterator fin);
ForwardIterator unique(ForwardIterator debut,
➥ ForwardIterator fin, BinaryPredicate predicat);
```

```
OutputIterator unique_copy(InputIterator debut,
➥ InputIterator fin, OutputIterator res);
OutputIterator unique_copy(InputIterator debut,
➥ InputIterator fin, OutputIterator res,
➥ BinaryPredicate predicat);
```

Dès qu'un groupe d'éléments dupliqués apparaît dans la séquence [debut, fin[, la fonction unique() les supprime tous sauf un. Ainsi, unique() renvoie l'itérateur nouvelle_fin tel que la séquence [debut, nouvelle_fin[ne contienne aucun élément dupliqué, autrement dit aucun doublon. Les itérateurs de la séquence [nouvelle_fin, fin[sont déjà déréférencés, et les éléments sur lesquels ils pointent sont indéterminés. unique() est stable et préserve l'ordre relatif des éléments conservés.

Une séquence [a, b[est un groupe de doublons si pour tout itérateur i de cette séquence i == a ou *i == *(i-1), pour la première version de unique(). Pour la deuxième version ce sera si i == a ou predicat(*i,*(i-1)) est vrai.

Les deux versions de unique_copy() agissent de même mais ne modifient pas la séquence d'entrée et produisent une nouvelle séquence ne contenant aucun doublon. Elles renvoient l'itérateur de fin de cette nouvelle séquence.

Attention

Ces fonctions ne suppriment que les doublons *consécutifs*. Ainsi, si un vector<int> V contient les valeurs 1, 3, 3, 3, 2, 2, 1, alors un appel à unique() produira un vecteur contenant les valeurs 1, 3, 2, 1.

L'exemple ci-après supprime tous les caractères identiques en ignorant la casse. Cela est fait en deux étapes : d'abord en triant le contenu alphabétiquement, puis en supprimant les doublons consécutifs.

```
inline bool eq_insensitif(char c1, char c2)
{
    return std::tolower(c1) == std::tolower(c2);
}
inline bool inf_insensitif(char c1, char c2)
{
    return std::tolower(c1) < std::tolower(c2);
}
void lettres_utilisees(const char *chaine)
{
    std::cout << chaine << std::endl;

    std::vector<char> V(chaine, chaine + strlen(chaine));
    std::sort(V.begin(), V.end(), inf_insensitif);
    std::copy(V.begin(), V.end(), std::ostream_iterator
    ➥<char>(std::cout));
    std::cout << std::endl;

    std::vector<char>::iterator fin2 =
    ➥std::unique(V.begin(), V.end(), eq_insensitif);
    std::copy(V.begin(), fin2, std::ostream_iterator
    ➥<char>(std::cout));
    std::cout << std::endl;
}
```

L'appel de `lettres_utilisees("La Bibliotheque Cpp Standard");` produira la sortie ci-après.

```
La Bibliotheque Cpp Standard
   aaaBbCddeehiilLnoppqrSttu
 aBCdehilnopqrStu
```

Copier à l'aide du constructeur par copie

```
#include <memory>
ForwardIterator2 uninitialized_copy(ForwardIterator1
➥ debut, ForwardIterator1 fin, ForwardIterator2 res);
ForwardIterator uninitialized_copy_n(InputIterator
➥ debut, Size N, ForwardIterator res);
```

En C++, l'opérateur `new` alloue de la mémoire pour un objet et appelle ensuite le constructeur sur ce nouvel emplacement mémoire. Occasionnellement, il est utile de pouvoir séparer ces deux opérations (ce principe est utilisé dans l'implémentation de certains conteneurs). Si chaque itérateur de la séquence [`res`, `res + (fin - debut)`[pointe vers une portion mémoire non initialisée, alors `uninitialized_copy()` crée une copie de [`debut`, `fin`[dans cette séquence. Ainsi, pour chaque itérateur `i` de la séquence d'entrée, cette fonction crée un copie de `*i` à l'emplacement mémoire indiqué par l'itérateur correspondant dans la séquence de sortie en appelant `construct(&*(res + i - first), *i)`.

`uninitialized_copy_n()` procède de même avec [`debut`, `debut + N`[comme séquence à copier.

```
class Entier
{
public:
    Entier(int e) : m_entier(e) {}
    int get() const { return m_entier; }
private:
    int m_entier;
};
```

```
void test()
{
    int valeurs[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    const int N = sizeof(valeurs) / sizeof(int);

    Entier* entiers = (Entier*) malloc(N *
    ➥ sizeof(Entier));
    std::uninitialized_copy(valeurs, valeurs + N,
    ➥ entiers);
}
```

Initialiser à l'aide du constructeur par copie

```
#include <memory>
void uninitialized_fill(ForwardIterator debut,
➥ ForwardIterator fin, const T& x);
void uninitialized_fill_n(InputIterator debut,
➥ Size N, const T& x);
```

En C++, l'opérateur `new` alloue de la mémoire pour un objet et appelle ensuite le constructeur sur ce nouvel emplacement mémoire. Occasionnellement, il est utile de pouvoir séparer ces deux opérations (ce principe est utilisé dans l'implémentation de certains conteneurs). Si chaque itérateur de la séquence [`debut`, `fin`[pointe vers une portion mémoire non initialisée, alors `uninitialized_fill()` crée autant de copies de `x` que nécessaire dans cette séquence. Ainsi, pour chaque itérateur `i` de la séquence d'entrée, cette fonction crée une copie de `x` à l'emplacement mémoire indiqué par cet itérateur en appelant `construct(&*i, x)`.

`uninitialized_fill_n()` procède de même avec [`debut`, `debut + N`[comme séquence à copier.

L'exemple suivant réutilise la classe `Entier` de l'exemple des fonctions précédentes :

```
Entier* test_initialisation(int val, int quantite)
{
    Entier valeur(val);
    Entier* entiers = (Entier*) malloc(quantite *
    ➥ sizeof(Entier));
    std::uninitialized_fill(entiers, entiers +
    ➥ quantite, valeur);
    return entiers;
}
```

13

BOOST

BOOST s'inscrit dans le prolongement de la bibliothèque standard STL. BOOST est une sorte de super-bibliothèque. Il s'agit en effet d'un ensemble de bibliothèques réunies en une seule. BOOST, conçu pour être utilisé dans un vaste spectre d'applications, est largement répandu. Sa licence (la *BOOST licence*) encourage aussi bien un usage commercial que non commercial.

Dix bibliothèques de BOOST sont déjà incluses dans le TR1 (rapport technique du comité de standardisation du langage C++) et dans les dernières versions de la STL. D'autres bibliothèques de BOOST ont été proposées pour le prochain TR2. Tout programmeur C++ à donc réellement intérêt à connaître BOOST.

BOOST fonctionne sur tout système d'exploitation moderne, qu'il s'agisse des systèmes Unix, GNU/Linux ou Windows. Les distributions GNU/Linux et Unix les plus populaires, comme Fedora, Debian ou NetBSD incluent des packages pré-compilés de BOOST.

Cerise sur le gâteau, certaines bibliothèques de BOOST n'ont même pas besoin d'être compilées. En effet, BOOST repose massivement sur les templates…

Vous pouvez télécharger BOOST sur **http://www.boost.org**.

Mettre en forme des arguments selon une chaîne de formatage

```
boost::format(format-string) % arg1 % arg2 % … % argN;
```

La bibliothèque `boost::format` fournit une classe pour la mise en forme d'arguments selon une chaîne de formatage, comme le fait `printf`. Il existe cependant deux différences majeures :

- `format` envoie les arguments dans un flux interne et est donc complètement sécurisée et supporte les types utilisateurs de façon transparente ;
- les points de suspension (…) ne peuvent pas être utilisés dans un contexte fortement typé ; ainsi l'appel de la fonction avec un nombre arbitraire d'arguments est remplacé par des appels successifs à un *fournisseur d'argument* : l'opérateur `%`.

La chaîne de formatage contient les directives spéciales qui seront remplacées par les chaînes résultant de la mise en forme des arguments donnés.

Notez qu'en tant que fonction sécurisée, `boost::format` contrôle le nombre d'arguments en fonction de la chaîne de formatage. Ainsi, avec la chaîne de formatage `"%1% %10% %3%"`, il est impératif de fournir dix arguments. En mettre plus ou moins provoque la levée d'une exception.

Une spécification de format *spec* est de la forme : [N$] [attributs] [largeur] [. précision] caractère-de-type

- **N$ (optionnel).** Indique que la spécification de format s'applique au *N*-ième argument. On ne peut pas mixer des spécifications avec ou sans indicateur de position d'argument dans une même chaîne de formatage ;

- **Attributs (optionnel).** Un formatage du texte peut être indiqué grâce à une suite d'attributs choisis parmi les possibilités décrites dans le tableau suivant.

Attributs

Attribut	Signification	Effet sur le flux interne
'-'	Aligner à gauche	N/A (appliqué plus tard sur la chaîne)
'='	Centrer	N/A (appliqué plus tard sur la chaîne)[1]
'_'	Alignement interne	Utiliser l'alignement interne[1]
'+'	Afficher le signe	Utiliser *showpos* des nombres positifs
'#'	Afficher la base	Utiliser *showbase* et *showpoint* et la décimale
'0'	Remplir avec des 0 (après le signe ou la base)	Si pas d'alignement à gauche, appelle *setfill('0')* et utilise *internal* Des actions supplémentaires sont faites après la conversion de flux pour les sorties personnalisées.
' '	Si la chaîne ne commence pas par + ou –, insérer un *espace* avant la chaîne convertie	N/A (appliqué plus tard sur la chaîne) Comportement différent de `printf` : il n'est pas affecté par l'alignement interne

1. Nouvelle fonctionnalité par rapport à `printf`.

- **Largeur (optionnel).** Indique une largeur minimale pour la chaîne résultant de la conversion. Si nécessaire, la chaîne sera complétée avec le caractère choisi soit par le manipulateur de flux, soit par celui mentionné dans la chaîne de formatage (par exemple attributs '00', '-', …) ;

- **Précision (optionnel).** Spécifie la *précision* du flux (elle est *toujours* précédée d'un point).
  - lorsque l'on écrit un nombre de type réel, il indique le nombre maximum de chiffres : après le point décimal en mode fixe ou scientifique ; au total en mode par défaut (`%g`),
  - lorsqu'on l'utilise avec un type chaîne `s` ou `S` il sert à limiter la chaîne convertie aux *précision* premiers caractères (avec un éventuel remplissage jusqu'à obtenir *largeur* caractères après cette troncature),
- **Caractère-de-type.** N'impose pas l'argument concerné à être de ce type mais définit les options associées au type mentionné (voir le tableau ci-après).

Caractères de type

Caractère	Signification	Effet sur le flux interne
p ou x	Sortie hexadécimale	Utiliser *hex*
o	Sortie octale	Utiliser *oct*
e	Format scientifique	Mettre les bits des réels à *scientific*
f	Format à virgule fixe	Mettre les bits des réels à *fixed*
g	Format des réels général (par défaut)	Invalide tous les bits de champ pour les réels
X, E ou G	Comme pour leurs équivalents en minuscules mais utilise des capitales (exposant, valeurs hexadécimales, …)	Comme x, e, ou g et utiliser *uppercase* en plus
d, l ou u	Sortie entière	Mettre les bits de base à *dec*

Caractères de type *(Suite)*

Caractère	Signification	Effet sur le flux interne
s ou S	Sortie de chaîne	La spécification de *précision* est invalide, et la valeur est stockée pour une troncature éventuelle (voir l'explication sur *précision* ci-dessus)
c ou C	Sortie d'un caractère	Seul le premier caractère de la conversion en chaîne est utilisé
%	Afficher le caractère %	N/A

Convertir une donnée en chaîne de caractères

```
#include <boost/lexical_cast.hpp>
class bad_lexical_cast;
template<typename TDestination, typename TSource>
  ➥TDestination lexical_cast(const TSource& arg);
```

`lexical_cast` convertit `arg` en chaîne de caractères de type `std::string` ou `std::wstring`. Si la conversion échoue, une exception `boost::lexical_cast` est déclenchée.

Pour fonctionner, ce patron requiert des types source et destination les capacités suivantes :

- `TSource` est redirigeable sur un flux (*OutputStreamable*). Un `operator<<` prenant à gauche un `std::ostream` ou `std::wostream` et à droite un `TSource` doit être défini ;
- `TDestination` est redirigeable depuis un flux (*InputStreamable*). Un `operator>>` prenant à gauche un `std::istream` ou `std::wistream` et à droite un `TDestination` doit être défini ;

- `TSource` et `TDestination` doivent avoir un constructeur par copie ;
- `TDestination` doit avoir un constructeur par défaut.

Le type caractère du flux sous-jacent est supposé être de type `char` à moins que `TSource` ou `TDestination` requiert un type caractère étendu. Les types `TSource` nécessitant un flux caractère étendu sont `wchar_t`, `wchar_t*`, et `std::wstring`. Les types `TDestination` nécessitant un flux caractère étendu sont `wchar_t` et `std::wstring`.

Info

Si vos conversions requièrent un haut niveau de contrôle sur la manière d'opérer la conversion, `std::stringstream` et `std::wstringstream` offrent une solution plus appropriée.

Si vous ne disposez pas de conversion basée sur les flux pour vos types, `lexical_cast` n'est pas le bon outil. D'ailleurs, il n'est pas conçu pour ce type de situation.

L'exemple suivant montre comment traiter la ligne d'argument d'un programme en tant que données numériques :

```
int main(int argc, char* argv[])
{
    std::vector<short> args;
    for (int i=1 ; i<argc && argv[i] !=0 ; ++i)
    {
        try
        {
            args.push_back( boost::lexical_cast<short>
            ➥ (argv[i]) );
        }
        catch(boost::bad_lexical_cast&)
        {
            args.push_back(0);
        }
        // …
    }
}
```

Cet autre exemple illustre l'utilisation d'une donnée numérique dans une opération sur des chaînes :

```
void afficher(const std::string&);
void afficher_erreur(int numero)
{
    using namespace boost;
    afficher("Erreur n°" + lexical_cast<std::
    ➥ string>(numero) + "…");
}
```

Voici un dernier exemple montrant la conversion d'une chaîne de caractères vers divers types numériques :

```
using namespace boost;
// …
std::string txt = "28.654";
// …
float a = lexical_cast<float>(txt);
double b = lexical_cast<double>(txt);
unsigned int c = lexical_cast<unsigned int>(txt);
sdt::string d = lexical_cast<std::string>(16978.3201);
// …
```

Construire et utiliser une expression régulière

```
#include <boost/regex.hpp>
boost::regex er(…);
```

```
bool boost::regex_match(BidirectionalIterator debut,
➥ BidirectionalIterator fin, [boost::match_results
➥ <…>& m,] const boost::basic_regex<…>& er, boost::
➥ match_flag_type = match_default);
```

```
bool boost::regex_match(const charT* str, [boost::
➥ match_results<…>& m,] const boost::basic_regex<…>
➥& er, boost::match_flag_type = match_default);
bool boost::regex_match(const std::basic_string<…>,
➥ [boost::match_results<…>& m,] const boost::basic
➥_regex<…>& er, boost::match_flag_type =
➥ match_default);
```

```
bool boost::regex_search(BidirectionalIterator
➥debut, BidirectionalIterator fin, [boost::match
➥_results<…>& m,] const boost::basic_regex<…>& er,
➥ boost::match_flag_type = match_default);
bool boost::regex_search(const charT* str,
➥[boost::match_results<…>& m,] const boost::
➥basic_regex<…>& er, boost::match_flag_type =
➥match_default);
bool boost::regex_search(const std::basic_string<…>,
➥[boost::match_results<…>& m,] const boost::
➥basic_regex<…>& er, boost::match_flag_type =
➥match_default);
```

La classe `boost::regex` permet de construire une expression régulière que l'on utilise ensuite avec l'une des deux autres fonctions fournies.

L'algorithme `boost::regex_match()` détermine si une expression régulière correspond à l'ensemble des caractères d'une séquence fournie (par une paire d'itérateurs bidirectionnels). Le plus souvent, cet algorithme est utilisé pour valider des données utilisateurs. À la place d'une séquence, vous pouvez aussi fournir une chaîne de caractères simple ou Unicode, style C ou C++. Fournir une variable pour récupérer l'information de correspondance est optionnel.

L'exemple suivant utilise les expressions régulières pour décoder le message de réponse d'un serveur FTP :

```
boost::regex expr("([0-9]+)(\\-| |$)(.*)");
int decoder_msg_FTP(const char* reponse, std::string* msg)
{
    boost::cmatch correspondance;
    if (boost::regex_match(reponse, correspondance, expr))
    {
        // correspondance[0] contient la chaîne complète
        // correspondance[1] contient le code de réponse
        // correspondance[2] contient le séparateur
        // correspondance[3] contient le message
        if (msg)
            msg->assign( correspondance[3].first,
                    ➥correspondance[3].second );
        return std::atoi( correspondance[1].first );
    }
    // pas de correspondance trouvée
    if (msg)
        msg->erase();
    return -1;
}
```

L'algorithme `boost::regex_search()` recherche une sous-chaîne correspondant à l'expression régulière donnée. Cet algorithme utilise diverses heuristiques pour réduire le temps de recherche, notamment si la possibilité de trouver une correspondance à la position courante est forte.

L'exemple ci-après recherche toutes les chaînes de caractères C dans un fichier source préalablement chargé comme chaîne de caractères :

```
boost::regex chaine_c("(\")([^\"]*)(\")");
void chaines_c(const std::string& contenu_fichier)
{
    std::string::const_iterator debut =
    ➥contenu_fichier.begin(), fin   = contenu_fichier.end();
   boost::match_results<std::string::const_iterator> quoi;
```

```
    boost::match_flag_type flags = boost::match_default;
while (boost::regex_search(debut,fin,quoi,chaine_c,flags))
    {
        std::string str( quoi[1].first, quoi[1].second );
        std::cout << "Trouvé : " << str << std::endl;
        // mise à jour de la position
        debut = quoi[0].second;
        // mise à jour des options
        flags |= boost::match_prev_avail;
        flags |= boost::match_not_bob;
    }
}
```

Caractères spéciaux des expressions régulières

Opérateur[1]	Description
.	N'importe quel caractère
x*	0, 1, 2, … fois le caractère x
x+	1, 2, 3, … fois le caractère x
x?	0 ou 1 fois le caractère x
x|y	x ou y
(x)	x
[xy]	x ou y (ensemble de caractères)
[x-y]	x, y, ou z (intervalle de caractères)
[^x]	n'importe quel caractère différent de x
\x	x, même si x est un caractère opérateur
x{n}	*n* fois x
x{n,m}	*n*, *n*+1, …, *m* fois x

1. Sauf indication contraire, « x », « y » ou « z » sont des expressions régulières quelconques. Par exemple, l'expression `t*` dénote 0 à *n* occurrences du caractère `t` et l'expression `'t|m)*` dénote 0 à *n* occurrences des caractères `t` ou `m`. L'expression correspondant à « x » est d'une complexité quelconque.

Caractère spécial	Description
^	En début, signifie début de chaîne. Par exemple, « ^debut » signifie toute chaîne commençant par « début ».
$	En fin, signifie fin de chaîne. Par exemple, « fin$ » signifie toute chaîne finissant par « fin ».

Éviter les pertes mémoire grâce aux pointeurs intelligents

Les pointeurs intelligents sont des objets qui stockent un pointeur sur des objets alloués dynamiquement. Ils fonctionnent de manière analogue aux pointeurs classiques, excepté qu'ils suppriment automatiquement les objets alloués au moment opportun. Plus exactement, lorsque plus aucun pointeur fort ne référence un objet donné, celui-ci est détruit et tous les pointeurs faibles qui étaient encore liés à cet objet deviennent des pointeurs nuls.

Attention

Même avec les pointeurs forts, vous pouvez rencontrer des pertes mémoire. Considérez l'exemple suivant :

```
void f(boost::shared_ptr<int>, int);
int g();
void ok()
{
    boost::shared_ptr<int> p(new int(1));
    f(p, g());
}
void perte()
{
    f(boost::shared_ptr<int>(new int(1)), g());
}
```

Les fonctions `ok()` et `perte()` semblent identiques, et pourtant, comme leur nom l'indique, l'une est correcte et l'autre peut provoquer une perte mémoire. En effet, la norme du C++ n'impose pas l'ordre d'évaluation des arguments d'une fonction. Ainsi, il est possible que l'instruction `new int(1)` soit évaluée en premier, puis que vienne en second `g()`, pour ne jamais arriver à la construction du `shared_ptr` si la fonction `g()` lance une exception. L'entier ainsi alloué ne sera jamais libéré !

Utiliser les pointeurs forts

```
#include <boost/shared_ptr.hpp>
boost::shared_ptr<Type> ptr(new Type);
```

```
#include <boost/shared_array.hpp>
boost::shared_array<Type> ptr(new Type[…]);
```

Les pointeurs forts `boost::shared_ptr<>` offrent le même niveau de sécurité que les type pointeurs bas niveau vis-à-vis des accès concurrents : aucune (ou presque). Une instance de `shared_ptr` peut être « lue » simultanément par plusieurs threads. Différentes instances de `shared_ptr` peuvent être modifiées (par l'opérateur = ou la fonction membre `reset()`) simultanément par différents threads, et (d'où le presque) même si ces différentes instances partagent le même compteur de référence.

```
// déclaration commune à tous les exemples
boost::shared_ptr<int> p(new int(42));
```

```
// thread A
shared_ptr<int> p2(p); // lit p
// thread B
shared_ptr<int> p3(p); // OK, lecture simultanée correcte
```

```
// thread A
p.reset(new int(1912)); // modifie p
// thread B
p2.reset(); // OK, modifie p2
```

```
// thread A
p = p3; // lit p3, modifie p
// thread B
p3.reset(); // écrit p3; INDEFINI, lecture/modif.
➥ simultanée
```

```
// thread A
p3 = p2; // lit p2, modifie p3
// thread B
// p2 devient hors d'atteinte: INDEFINI, le
➥ destructeur est considéré comme une modification
```

```
// thread A
p3.reset(new int(1));
// thread B
p3.reset(new int(2)); // INDEFINI, modification simultanée
```

Info

La classe `boost::shared_ptr<>` ne peut être utilisée que sur des objets simples. Comment faire si vous avez alloué un tableau d'objets ? Dans ce cas, vous avez à votre disposition la classe suivante : `boost::shared_array<>`, disponible dans `<boost/shared_array.hpp>`.

Utiliser les pointeurs faibles

```
#include <boost/weak_ptr.hpp>
boost::weak_ptr<Type> ptr(boost::shared_ptr<Type>);
```

Les pointeurs faibles (`boost::weak_ptr<>`) stockent une référence sur un pointeur fort. Mais du coup, leur utilisation n'est pas sûre, car l'objet sous-jacent peut être libéré à tout moment, comme le montre l'exemple suivant :

```
boost::shared_ptr<int> p(new int(10));
boost::weak_ptr<int> q(p);
//…
if (!q.expired())
{
    // ici q est non nul, mais peut pointer sur n'importe
       ➥ quoi
    // si l'objet a été libéré entre temps à cause d'un
       ➥ autre
    // thread exécutant par exemple p.reset().
}
```

Pour cela, il suffit de convertir momentanément le pointeur faible en fort :

```
boost::shared_ptr<int> p(new int(10));
boost::weak_ptr<int> q(p);
//…
if (boost::shared_ptr<int> r = q.lock())
{
    // ici l'utilisation r (par *r ou r->…) est sûre
}
```

Utiliser les pointeurs locaux

```
#include <boost/scoped_ptr.hpp>
boost::scoped_ptr<Type> ptr(new Type);

#include <boost/scoped_array.hpp>
boost::scoped_array<Type> ptr(new Type[…]);
```

Les pointeurs locaux `boost::scoped_ptr<>` et `boost::scoped_array<>` sont comme des pointeurs forts à existence unique. Ils ne peuvent être copiés, et donc n'ont pas besoin de gérer de compteur de référence. À quoi sont-ils donc utiles ? Non à gérer des singletons mais à associer simplement un objet à un autre, comme le montre l'exemple ci-après. Ou bien encore, servir à assurer la destruction d'objets alloués localement. N'hésitez pas à utiliser ce type de pointeur intelligent, car de par sa simplicité intrinsèque, il ne conduit à aucune perte de performance.

```
class MaClasse
{
    boost::scoped_ptr<int> ptr;
public:
    MaClasse() : ptr(new int) { *ptr=0; }
    // …
};
```

Mettre en œuvre des pointeurs (forts) instrusifs

```
#include <boost/intrusive_ptr.hpp>
boost::intrusive_ptr<Type> ptr(Type*);
boost::intrusive_ptr<Type> ptr(Type*,false);
```

Les `boost::intrusive_ptr<>` permettent de stocker des pointeurs vers des objets possédant déjà un compteur de référence. Cette gestion se fait par l'intermédiaire d'appels aux fonctions `intrusive_ptr_add_ref(`*`Type`*`*)` et `intrusive_ptr_release(`*`Type`*`*)`, qu'il vous appartient de surcharger.

Si vous ne savez pas quoi utiliser, entre pointeurs fort et intrusif, commencez par tester si les pointeurs forts conviennent à votre besoin.

Créer des unions de types sécurisées

```
#include <boost/variant.hpp>
boost::variant<Type1,Type2,…> variable;
U * boost::get(variant<T1, T2, …, TN> * operand);
const U* boost::get(const variant<T1, T2, …, TN>*
➥ operand);
U& boost::get(variant<T1, T2, …, TN>& operand);
const U& boost::get(const variant<T1, T2, …, TN>&
➥ operand);
```

La classe `boost::variant<>` permet de réunir plusieurs types de données en un seul, de manière sûre, simple et efficace. Réunir ne signifie pas que l'on stocke un objet de chacun des types, mais un seul objet de l'un des types spécifiés. Cela permet de créer une sorte de type générique pouvant stocker un objet d'un type ou d'un autre. L'exemple suivant montre comment utiliser cette solution :

```
#include <boost/variant.hpp>
#include <iostream>

struct visiteur_entier : public boost::static_visitor<int>
```

```
{
    // si le variant est un entier
    int operator()(int i) const
    {
       return i; // rien à faire
    }

    // si c'est une chaîne
    int operator()(const std::string& s) const
    {
        return s.length(); // le convertir en sa longueur
    }
};
int main()
{
    boost::variant< int, std::string > u("bonjour");
    std::cout << u; // u est une chaîne, affiche : bonjour

    int r = boost::apply_visitor( visiteur_entier(), u );
    std::cout << r; // affichera la longueur de la chaîne
                    // contenu dans le variant, soit : 7
}
```

Un autre moyen, peut-être plus simple mais moins puissant (car n'effectuant aucune conversion) est d'utiliser la fonction `boost::get<>()`.

L'exemple suivant montre comment la mettre en œuvre dans vos programmes :

```
void fois_trois( boost::variant< int, std::string >& v)
{
    if (int* i_ptr = boost::get<int>(&v))
        *i_ptr *= 3;
    else if (std::string* s_ptr = boost::get
    ➥ <std::string>(&v))
        *s_ptr = (*s_ptr) + (*s_ptr) + (*s_ptr);
}
```

Mais là encore, si l'on ajoute un type supplémentaire, il est facile d'oublier d'ajouter le code nécessaire. La robustesse à ce niveau réside plutôt dans la manière d'écrire un visiteur : en utilisant les templates et en se basant sur un opérateur ou une fonction commune aux différents types présent dans le variant. L'exemple suivant l'illustre à travers un visiteur utilisant l'opérateur += :

```
struct fois_deux_generic : boost:: static_visitor<>
{
    template <typename T>
    void operator()(T& v) const
    {
        v += v;
    }
};
// …
std::vector< boost::variant<int, std:: string> > V;
V.push_back( 21 );
V.push_back( "bonjour ");
fois_deux_generic visiteur;
std::for_each(V.begin(), V.end(),
              ➥ boost::apply_visitor(visiteur));
// V == { 42, "bonjour bonjour " }
```

Info

En programmation, il est bien souvent utile de manipuler des types distincts comme s'il s'agissait d'un seul. En C++, on retrouve cet aspect avec les unions de types (`union`, voir la section « Types élaborés » du Chapitre 1). L'exemple suivant en montre un bref rappel :

```
union { int i; double d; } nombre;
nombre.d = 3.141592654;
nombre.i = 12; // écrase nombre.d
```

Malheureusement, cette technique devient inopérante dès que l'on se replace dans un contexte orienté objet. Les unions de type (`union`) arrivent en effet directement de l'héritage du langage C.

Par voie de conséquence, ils ne supportent que des types primaires ou primaires composés, et ne supportent donc pas la construction ou la destruction d'objet non triviaux. L'exemple suivant illustre clairement cette limitation des unions de type (`union`) :

```
union
{
    int i;
    std::string s; // ne fonctionne pas, n'est pas "trivial"
} u;
```

Une nouvelle approche est donc requise. Vous trouverez, si vous ne les avez pas déjà rencontrées, des solutions basées sur :

- l'allocation dynamique ;
- un type de base commun et l'utilisation de fonctions virtuelles ;
- des pointeurs non typés `void*` (le pire des cas).

Les objets réels sont souvent, dans ces solutions, retrouvés à l'aide de transtypage comme `dynamic_cast`, `boost::any_cast`, etc. Toutefois, ces solutions sont fortement sujettes à erreurs, telles :

- les erreurs de *downcast* ne sont pas détectées à la compilation. Une erreur de ce genre ne peut donc se détecter que durant l'exécution ;
- l'ajout de nouveaux types concrets est ignoré. Si un nouveau type concret est ajouté à la hiérarchie, les *downcasts* existants continueront de fonctionner tel quels, tout en ignorant royalement l'existence de ce nouveau type. Le programmeur doit donc trouver de lui-même tous les emplacements de codes à modifier. Beaucoup de temps à passer et beaucoup d'erreurs en perspective...

De plus, même bien implémentées, ces solutions resteront pénalisées par le coût de l'abstraction à laquelle elles font appel. Ce coût se retrouve dans l'appel des fonctions virtuelles et des transtypages dynamiques.

Parcourir un conteneur avec BOOST_FOREACH

```
BOOST_FOREACH(type element, conteneur)
{
    // …
}
```

En C++, écrire une boucle peut parfois s'avérer fastidieux. Utiliser les itérateurs, avec leurs déclarations templatisées à rallonge, peut facilement alourdir l'écriture et donc la lisibilité. L'algorithme `std::foreach()` quant à lui oblige à déplacer tout le bloc de code dans un prédicat. Si cela peut être très pratique pour une utilisation récurrente d'un même prédicat, cela l'est beaucoup moins pour du ponctuel ou du spécifique.

`BOOST_FOREACH` est conçu pour la facilité d'utilisation et l'efficacité : pas d'allocation dynamique, pas d'appel de fonction virtuelle ou de pointeur de fonction, ni d'appel qui ne puisse être optimisé par le compilateur. Le code produit est proche de l'optimal que l'on aurait obtenu en codant la boucle soi-même. Bien qu'étant une macro, celle-ci se comporte sans surprise : ses arguments ne sont évalués qu'une seule fois.

```
#include <string>
#include <iostream>
#include <boost/foreach.hpp>

//…
std::string chaine( "Bonjour" );
BOOST_FOREACH( char ch, chaine )
{
    std::cout << ch;
}
```

`BOOST_FOREACH` parcourt les séquences, en se basant sur `boost::range`, comme par exemple :

- les conteneurs STL ;
- les tableaux de type C ;
- les chaînes à zéro terminal (`char` et `wchar_t`) ;
- une `std::pair` d'itérateurs.

Vous trouvez qu'écrire `BOOST_FOREACH` est trop long ? Que l'écriture en capitale d'imprimerie n'est pas à votre goût ? Possible, mais cela n'en reste pas moins le standard adopté à travers les conventions de nommage de la bibliothèque BOOST. Il n'appartient qu'à vous de le renommer ainsi :

```
#define foreach BOOST_FOREACH
```

Attention toutefois de ne pas causer un conflit de nom dans votre code.

Attention

N'utilisez pas `#define foreach(x,y) BOOST_FOREACH(x,y)` : cela posera problème lors d'une utilisation avec des macros comme paramètres, provoquant une création de code supplémentaire lors de l'interprétation. Utilisez plutôt la technique précitée.

Générer des messages d'erreur pendant le processus de compilation

```
#include <boost/static_assert.hpp>
BOOST_STATIC_ASSERT( condition )
```

La macro `BOOST_STATIC_ASSERT(x)` permet de générer des messages d'erreur pendant le processus de compilation si l'expression constante `x` est fausse. Elle est l'équivalent des macros d'assertion classiques, mais contrairement à ces dernières qui se déclenchent pendant l'exécution du programme, celles-ci se déclenchent pendant la compilation. Les concepteurs de cette partie de BOOST ont choisi de nommer ce type d'assertion des « assertions statiques » (à l'opposé de dynamique). Notez que si la condition est `true`, alors aucun code n'est généré par cette macro. Si elle est utilisée au sein d'un template, elle sera évaluée lors de l'instanciation de ce dernier. Ceci est particulièrement utile pour vérifier certaines conditions sur les paramètres templates.

L'un des atouts principaux de `BOOST_STATIC_ASSERT()` est de générer des messages d'erreur humainement lisibles. Ils indiquent ainsi immédiatement et clairement si le code d'une bibliothèque (ou votre code) est mal utilisé. L'affichage peut varier d'un compilateur à un autre, mais il devrait ressembler à ceci :

```
Illegal use of STATIC_ASSERTION_FAILURE<false>
```

Vous pouvez utiliser `BOOST_STATIC_ASSERT()` au même endroit que n'importe quelle déclaration : dans une classe, une fonction ou un espace de noms.

L'exemple ci-après permet de contrôler que le type `int` est codé sur au moins 32 bits, et que le type `wchar_t` est bien non signé :

```
#include <climits>
#include <cwchar>
#include <limits>
#include <boost/static_assert.hpp>

namespace mes_conditions
{
  BOOST_STATIC_ASSERT(std::numeric_limits<int>::
  ➥ digits >= 32);
  BOOST_STATIC_ASSERT(WCHAR_MIN >= 0);
}
```

Lors de l'écriture de template, il est intéressant de pouvoir tester si les paramètres fournis vérifient bien certaines conditions. Cela permet, en partie, de garantir le bon fonctionnement de la fonction générique écriture. Sinon, cela apporte au moins l'avantage d'obtenir des message d'erreur plus explicites qu'un traditionnel message relatif à un problème de syntaxe… L'exemple ci-après montre comment utiliser `BOOST_STATIC_ASSERT()` pour être certain qu'un message d'erreur explicite apparaisse lorsqu'on utilise un mauvais type d'itérateur avec un algorithme donné.

```
#include <iterator>
#include <boost/static_assert.hpp>
#include <boost/type_traits.hpp>

template <class RandomAccessIterator >
RandomAccessIterator mon_algorithme(
    RandomAccessIterator from,
    RandomAccessIterator to)
{
    // cet algorithme ne doit être utilisé qu'avec des
```

```
    // itérateurs à accès direct (ou random access
       ➥ iterator)
    typedef typename std::iterator_traits<
    ➥RandomAccessIterator >::iterator_category cat;
    BOOST_STATIC_ASSERT((boost::is_convertible<cat,
    ➥ const std::random_access_iterator_tag&>::value));
    // …
    // implémentation
    // …
    return from;
}
```

Astuce

Les doubles parenthèses de l'exemple précédent ont pour but d'être certain que la virgule ne soit pas prise comme séparateur d'argument de macro.

L'exemple ci-après montre l'utilisation de `BOOST_STATIC_ASSERT()` avec les classes templates. Il expose un moyen de vérifier que le paramètre soit au moins un type entier, signé, et sur au moins 16 bits.

```
#include <climits>
#include <boost/static_assert.hpp>

template <class UnsignedInt>
class MaClasse
{
private:
    BOOST_STATIC_ASSERT(
     ➥(std::numeric_limits<UnsignedInt>::digits >= 16)
     ➥&& std::numeric_limits<UnsignedInt>::is_specialized
     ➥&& std::numeric_limits<UnsignedInt>::is_integer
     ➥&& !std::numeric_limits<UnsignedInt>::is_signed);
public:
   // …
};
```

Attention

Certains compilateurs ont parfois tendance à en faire trop. Théoriquement, les assertions statiques présentes dans une classe ou une fonction template ne sont pas instanciées tant que le template dans lequel elles apparaissent n'est pas instancié. Pourtant, il existe un cas un peu à part : lorsque l'assertion statique ne dépend pas d'au moins un paramètre template, comme ci-après :

```
template <class T>
struct ne_peut_pas_etre_instancie
{
    BOOST_STATIC_ASSERTION(false); // ne dépend
    ➥ d'aucun paramètre template
};
```

Dans ce cas, l'assertion statique peut être évaluée même si la classe n'est jamais instanciée. C'est le cas avec, au moins, les compilateurs Intel 8.1 et gcc 3.4. Un moyen de palier cet inconvénient est de rendre l'assertion dépendante d'un paramètre template. L'exemple suivant montre comme le faire, à l'aide de l'instruction `sizeof()` :

```
template <class T>
struct ne_peut_pas_etre_instancie
{
    BOOST_STATIC_ASSERTION(sizeof(T)==0); // dépendant
    ➥d'un paramètre template
};
```

14

Programmation multithread avec QT

Les threads sont maintenant très répandus. Il existe diverses bibliothèques permettant de les appréhender facilement. BOOST fournit une bibliothèque haut niveau pour les créer et les utiliser, wxWidgets une autre. Dans cet ouvrage, j'ai choisi d'utiliser ceux de QT.

Créer un thread

```
#include <QThread>
class MonThread : public Qthread
{
    Q_OBJECT
public:
   void run() { … }
};
```

Pour créer un thread avec QT, il suffit de dériver la classe `QThread` et de réimplémenter la fonction membre `run()` de cette dernière. Ensuite, il suffit de créer une instance de votre nouvelle classe et d'appeler la fonction membre `start()` pour exécuter le contenu de la fonction `run()` dans un nouveau thread.

Il existe une contrainte avec les threads QT : il est impératif de créer un objet `QApplication` ou `QCoreApplication` avant de créer un `QThread`.

Partager des ressources

```
#include <QSharedMemory>
QSharedMemory sm(const QString& clef, QObject*=0);
QSharedMemory sm(QObject*=0);
bool QSharedMemory::create(int taille, AccessMode
➥ mode = ReadWrite);
int QSharedMemory::size() const;
bool QSharedMemory::attach(AccessMode mode =
➥ ReadWrite );
bool QsharedMemory::detach();
bool QSharedMemory::isAttached() const;
const void* QSharedMemory::constData() const;
void* QSharedMemory::data();
bool QSharedMemory::lock();
void* QSharedMemory::unlock();
```

`QSharedMemory` fournit un accès à un segment de mémoire partagé par plusieurs threads ou processus. Cette classe procure aussi un moyen simple de bloquer cette mémoire en accès exclusif. Lorsque vous utilisez cette classe, il faut être conscient qu'il existe des différences d'implémentation en fonction de la plate-forme sur laquelle vous utilisez la bibliothèque QT.

Sous Windows, la classe ne possède pas le segment de mémoire. Quand tous les threads et processus ayant un lien avec ce segment de mémoire réservé ont détruit leur instance de `QSharedMemory` ou se sont terminés, alors le noyau de Windows libère le segment de mémoire automatiquement.

Sous Unix, `QSharedMemory` possède le segment de mémoire. Le comportement est le même que sous Windows en ce sens que c'est toujours le noyau du système qui libère le segment. Mais, si l'un des threads ou des processus concernés plante, il se peut que le segment de mémoire ne soit jamais libéré.

Sous HP-UX, un processus ne peut être lié qu'une fois sur un segment mémoire. Du coup, vous devrez vous-même gérer les accès concurrentiels à ce segment entre les différents threads d'un même processus. Dans ce contexte, `QSharedMemory` ne pourra pas être utilisé.

Utilisez `lock()` et `unlock()` lorsque vous lisez ou écrivez des données du segment de mémoire partagée.

Se protéger contre l'accès simultané à une ressource avec les mutex

```
#include <QMutex>
QMutex m(mode); // == NonRecursive par défaut
m.lock();
bool r = m.tryLock();
bool r = m.tryLock(timeout);
m.unLock();
```

Les mutex permettent de se prémunir contre l'accès et/ou la modification d'une ressource (mémoire, objet, driver, …) par plusieurs threads en même temps. Cela est souvent essentiel au bon déroulement d'un programme. Pour comprendre l'intérêt d'un tel mécanisme, imaginons que deux fonctions `func1()` et `func2()` soient exécutées dans des threads différents, en même temps, et effectuent chacune des calculs sur la même variable. Le résultat serait imprévisible.

L'exemple ci-après montre comment protéger l'accès à cette variable pour éviter qu'elle soit modifiée et utilisée par plusieurs threads en même temps.

```
QMutex m;
double variable = 20.0;

void func1()
{
    m.lock();
    variable *= 10.0;
    variable += 3.0;
    m.unLock();
}

void func2()
{
    m.lock();
    variable -= 100.0;
    variable /= 34.0;
    m.unLock();
}
```

Nous l'avons dit, un mutex sert à protéger un objet (ici une variable) contre des accès concurrentiels. Pour quoi cela ? Simplement pour éviter des comportements imprévisibles, dans le meilleur des cas, ou des plantages aléatoires et très difficiles à cerner dans le pire des cas. Pour comprendre ce risque, imaginez que l'exemple précédent ne contienne aucune protection par mutex. Dans ce cas, l'exécution du code pourrait se passer comme suit :

```
// thread 1 exécutant func1
variable *= 10.0;        // 200
// thread 2 exécutant func2
variable -= 100.0;       // 100
variable /= 34.0;        // 100/34 = 2 + 16/17
// thread 1 exécutant func1
variable += 3.0;         // 5 + 16/17
```

L'utilisation du mutex empêchera `func2()` de commencer ses calculs avant que `func1()` n'ait fini les siens.

```
// thread 1 exécutant func1
variable *= 10.0;        // 200
variable += 3.0;         // 203
// thread 2 exécutant func2
variable -= 100.0;       // 103
variable /= 34.0;        // 103/34 = 3 + 1/34
```

Cet exemple est trivial mais permet de bien comprendre le mécanisme mis en jeu.

Astuce

Souvent, les fonctions présentent un code bien plus complexe, avec plusieurs instructions `return` disséminées ça et là, sans oublier les lancements d'exceptions ou les appels à d'autres fonctions pouvant elles aussi déclencher d'autres exceptions. Du coup, il peut devenir délicat de n'oublier aucune libération d'un mutex. Pour cela, nous pourrions écrire nous-même une classe liée au mutex qui nous intéresse et laisser le soin à son destructeur de libérer ledit mutex. Or la bibliothèque de QT fournit déjà cette petite merveille : `QMutexLocker`. L'exemple ci-après vous montre comment l'utiliser avec la première fonction. Même s'il reste trivial, il expose l'essentiel de ce qu'il faut savoir :

```
#include <QMutexLocker>
void func1()
{
    QMutexLocker(&m);
    variable *= 10.0;
    variable += 3.0;
    // ici le destructeur de QMutextLocker appelle
       ➥ m.unLock()
}
```

Ainsi, adieu les risques désastreux d'oubli de libération de mutex !

Contrôler le nombre d'accès simultanés à une ressource avec les sémaphores

```
#include <QSemaphore>
QSemaphore s(n); // n = 0 si omis
void QSemaphore::acquire(n); // n = 1 si omis
int QSemaphore::available() const;
void QSemaphore::release(n) // n = 1 si omis
bool QSemaphore::tryAcquire(n); // n = 1 si omis
bool QSemaphore::tryAcquire(n, tempsLimite);
```

Les sémaphores sont un peu comme des mutex mais peuvent permettre à plusieurs threads d'utiliser en même temps les ressources qu'elles protègent. On pourrait voir une sémaphore comme un agent de location qui dispose de `n` (valeur donnée au constructeur de la classe) ressources, comme des vélos par exemple. Un client peut louer un ou plusieurs vélos d'un coup (par l'appel à `acquire(v)`) et les rendre d'un coup ou par lot (par un ou plusieurs appels à `release(w)`).

Un appel à `acquire()` oblige à se placer dans une file d'attente jusqu'à ce que le nombre de ressources demandées soient disponibles. Si le sémaphore n'en dispose pas d'autant, je vous laisse deviner le problème… il risque fort de bloquer tout le monde. Du coup et pour éviter ce problème, vous disposez également de deux autres méthodes :

- `tryAcquire(n)` pour tenter d'acquérir `n` ressources ou partir s'il n'y en a pas. Le client devra patienter de lui-même en dehors de la file d'attente de la boutique et retenter sa chance plus tard ;

- `tryAcquire(n,tempsLimite)` pour demander `n` ressources et se placer dans la file d'attente. Mais si le client n'est pas servi avant le `tempsLimite` (en millisecondes) donné, il décide de sortir de la file d'attente (et de perdre sa place) pour poursuivre son chemin.

Info

Avec les sémaphores, vous pouvez commencer avec un nombre de ressources nul (égal à zéro). Libérer une ou plusieurs ressources revient alors à lui en allouer autant. Ce comportement est valable à tout moment. Cette implémentation ne gère donc pas de test sur un maximum de ressource que l'on définirait à la création d'un sémaphore. Le code suivant illustre ce fait :

```
QSemaphore s(3); // s.available() == 3
s.acquire(1);    // s.available() == 2
s.acquire(2);    // s.available() == 0
s.release(4);    // s.available() == 4, on a dépassé 3
s.release(5);
// s.available() == 9, on en alloue donc bien en plus
```

Attention

Si vous oubliez de libérer des ressources ou que vous en demandez trop par rapport à leur disponibilité, vous provoquerez un *deadlock*. Prenez-y garde dès la conception de vos algorithmes. N'attendez pas de les voir apparaître dans vos tests.

Le *deadlock* peut aussi parfois survenir lorsque l'on oublie de protéger ses ressources avec un mutex ou un sémaphore. Du coup, des variables peuvent avoir été modifiées sans qu'on le veuille et notre programme se mettre en attente ou en boucle infinie...

Prévenir le compilateur de l'utilisation d'une variable dans plusieurs threads

```
volatile Type variable; // global, local ou membre
➥ de classe.
```

Le mot-clé `volatile` permet de prévenir le compilateur que cette variable va être utilisée dans plusieurs threads différents. Cela lui permet de la traiter de manière spéciale, notamment lorsque l'on utilise les options de compilation relatives à l'optimisation.

Ce qualificateur indique que la variable ou l'objet peut être modifié par une interruption matérielle, par un autre programme, un autre processus ou un autre thread. Cela implique de devoir recharger systématiquement la variable à chaque fois qu'elle est utilisée, même si elle se trouve déjà dans un des registres du processeur, ce qui explique le traitement spécial de ce type de variables vis-à-vis des optimisations de programmes réalisées par le compilateur.

15

Base de données

Les bases de données sont couramment utilisées en programmation. Ce chapitre vous présente quelques outils de base selon une approche multisystème.

Nous avons mis l'accent sur SQLite en raison de sa simplicité d'emploi et de sa popularité grandissante. Vous pouvez en télécharger les sources sur **http://www.sqlite.org**. La première section vous aidera à déterminer si SQLite répond à vos besoins, les suivantes à le mettre en œuvre. Les dernières sections vous permettront d'utiliser les bases de données avec un pilote ODBC et la bibliothèque graphique wxWidgets.

Astuce

Afin de rester le plus multiplate-forme possible, préférez un système gérant directement les requêtes SQL.

Voici quelques sites que je vous invite à prendre en considération :

- **http://sql.1keydata.com/fr**. Pour retrouver la syntaxe d'une requête SQL.
- **http://dev.mysql.com/doc/refman/5.1/en/tutorial.html**. Pour apprendre à mettre en œuvre MySQL.

- **http://www.sqlapi.com**. SQLAPI++ est une API C++ permettant d'accéder à diverses bases de données SQL (Oracle, SQL Serveur, DB2, Sybase, Informix, Interbase, SQLBase, MySQL, PostgreSQL, ODBC et SQLite). SQLAPI++ utilise les API natives des moteurs de bases de données concernées et s'exécute donc rapidement et efficacement. Cette bibliothèque fournit également une interface bas niveau permettant d'accéder à des caractéristiques particulières des bases de données. En encapsulant les API des divers moteurs, SQLAPI++ agit comme un intergiciel (*middleware*) et favorise la portabilité. Elle est disponible pour les systèmes Win32, Linux et Solaris, et au moins les compilateurs Microsoft Visual C++, Borland C++ Builder, GNU g++. Cette bibliothèque est un shareware.
- **http://wxcode.sourceforge.net/components/wxsqlite3**. Pour utiliser SQLite avec wxWidgets.
- **http://sqlitepp.berlios.de**. Pour les puristes du C++, voici une bibliothèque C++ encapsulant celle de SQLite.
- **http://debea.net**. Debea (*DataBasE Access library*) est une collection d'interfaces permettant de connecter des objets C++ à diverses bases de données. Cette bibliothèque ne cherche pas à supprimer les requêtes SQL, mais génère automatiquement certaines d'entre elles, accélérant d'autant votre vitesse de développement. Disponible pour Linux (g++ 3.6 ou plus), Windows 98/2000/XP (VC++ 2003 et 2005). Cette bibliothèque possède également une interface pour wxWidgets : wxDba.
- **http://www.oracle.com/technology/software/products/database/index.html**. Oracle est à présent gratuit pour coder des prototypes d'application (non commercialisés, non utilisés intensivement en tant

qu'outil interne). Disponible pour Windows (32 et 64 bits), Linux (32 et 64 bits), Solaris (SPARC) 64 bits, AIX (PPC64), HP-UX (Itanium et PA-RISC).

Savoir quand utiliser SQLite

SQLite est différent de la plupart des autres systèmes de bases de données SQL. Il a d'abord été conçu pour être simple :

- à administrer ;
- à utiliser ;
- à embarquer dans un programme ;
- à maintenir et personnaliser.

SQLite est apprécié parce qu'il est petit, rapide et stable. Sa stabilité découle de sa simplicité (moins de fonctionnalités, moins de risques d'erreur). En revanche, pour atteindre ce niveau de simplicité, il a fallu sacrifier des fonctionnalités qui, dans certains contextes, sont appréciables, comme la possibilité de faire face à une forte demande d'accès simultanés (*high concurrency*), une gestion fine des contrôles d'accès, la mise à disposition d'un large ensemble de fonctions intégrées, les procédures stockées, l'évolution vers le téra-péta-octet, etc. Si vous avez besoin de telles fonctionnalités et que la complexité qu'elles induisent ne vous fait pas peur, alors SQLite n'est probablement pas pour vous. SQLite ne prétend pas concurrencer Oracle ou PostgreSQL. Il n'a pas été conçu (*a priori*) pour être le moteur de base de données de toute une entreprise.

La règle de base pour déterminer s'il convient à vos besoins est est la suivante : préférez SQLite lorsque la simplicité d'administration, de mise en œuvre et de maintenance est plus importante que les innombrables (et complexes)

fonctionnalités que les systèmes de bases de données d'entreprise fournissent. Il se trouve que les situations où la simplicité est primordiale sont bien plus courantes que ce que l'on peut croire.

Info

SQLite s'avère aussi être un moyen de gérer des fichiers, en remplacement de la fonction C `fopen()`.

Cas d'emploi de SQLite

Format de fichier d'une application

SQLite est utilisé avec beaucoup de succès comme un format de fichier disque pour des applications bureautiques telles que outils d'analyse financière, progiciels CAD, progiciels de suivi de dossiers, etc. Les opérations traditionnelles d'ouverture de fichiers effectuent en réalité un `sqlite3_open()` suivi de l'exécution d'un `BEGIN TRANSACTION` pour verrouiller l'accès au contenu. La sauvegarde fait un `COMMIT` suivi d'un autre `BEGIN TRANSACTION`. L'usage des transactions garantit que les mises à jour des fichiers de l'application sont atomiques, durables, isolées et cohérentes.

Des déclencheurs (*triggers*) temporaires peuvent être ajoutés à la base de données pour enregistrer toutes les modifications dans une table temporaire de « annuler/refaire ». Ces changements peuvent ensuite être lus lorsque l'utilisateur appuie sur les boutons « Annuler » et « Refaire ». Grâce à cette technique, et sans limitation de la profondeur de l'historique des « annuler/refaire », la mise en œuvre de cette fonctionnalité peut être faite à l'aide de très peu de code.

Dispositifs et applications embarquées

Une base SQLite requérant peu ou pas d'administration, SQLite est un bon choix pour les dispositifs ou services devant fonctionner sans surveillance et sans intervention humaine. SQLite est adapté pour une utilisation dans les téléphones cellulaires, les assistants numériques (PDA), les boîtes noires et autres appareils. Il fonctionne aussi en tant que base de données embarquée dans des applications téléchargeables.

Sites web

SQLite fonctionne habituellement bien comme moteur de base de données de sites web à faible ou moyen trafic (c'est-à-dire plus de 99 % des sites). La quantité de trafic que SQLite peut traiter dépend bien sûr de la façon dont le site fait usage de ses bases de données. De manière générale, tout site recevant moins de 100 000 visites par jour devrait fonctionner correctement avec SQLite. Cette limite est une estimation prudente, et non une limite supérieure infranchissable. Certains sites atteignant un nombre de visites dix fois supérieur fonctionnent parfaitement avec SQLite.

Remplacement de fichiers propriétaires

Beaucoup de programmes utilisent `fopen()`, `fread()`, et `fwrite()` pour créer et manipuler des fichiers de données dans des formats propriétaires. SQLite fonctionne particulièrement bien pour remplacer ces fichiers propriétaires.

Bases de données temporaires

Pour les programmes qui doivent filtrer ou trier un grand nombre de données, il est souvent plus facile et plus rapide de charger un modèle mémoire de base de

données SQLite et d'utiliser des requêtes avec jointures et des clauses `ORDER BY` pour extraire des données dans la forme et l'ordre désiré plutôt que de devoir coder soi-même ces fonctionnalités à la main. Utiliser une base de données SQL de cette manière permet également une plus grande souplesse, car de nouvelles colonnes et index peuvent être ajoutés sans avoir à recoder chaque requête.

Outils d'analyse de données en ligne de commande

Ceux qui sont expérimentés en SQL peuvent utiliser les programmes en ligne de commande fournis par SQLite pour analyser divers jeux de données. Les données brutes peuvent être importées de fichiers CSV, puis découpées et groupées pour générer une multitude de rapports. Ce type d'utilisation inclut l'analyse des fichiers logs des sites web, l'analyse de statistiques sportives, la compilation de métrique de code source ou l'analyse de résultats expérimentaux.

Vous pouvez bien entendu faire de même avec des bases de données client/serveur professionnelles. SQLite restera toutefois dans ce cas bien plus facile à mettre en place dans le cadre de fichiers simples, surtout si vous souhaitez les partager avec d'autres personnes (*via* une clé USB, une pièce jointe à un courrier électronique, etc.).

Base locale pour tests ou démos

Si vous écrivez une application cliente pour un moteur de base de données d'entreprise, il est logique d'utiliser une interface dorsale (*backend*) pour vous permettre de vous connecter à n'importe quel type de bases de données SQL. Dans ce cadre, il est encore plus logique et bénéfique d'inclure (en édition de lien statique) le support de SQLite dans la partie cliente.

De cette façon, le programme client pourra être utilisé comme un programme autonome avec des fichiers de données SQLite pour des phases de test ou même comme programme de démonstration.

Enseignement et/ou apprentissage

De par sa simplicité d'utilisation, SQLite est un très bon moteur de bases de données pour apprendre ou enseigner le langage SQL. En effet, SQLite est trivial à installer : il suffit de copier l'exécutable sqlite ou sqlite.exe sur la machine souhaitée et de l'utiliser.

Les étudiants (ou vous-même) peuvent facilement créer autant de bases de données qu'ils le souhaitent et peuvent envoyer leurs bases de données à leur professeur (ou à un collègue) par e-mail. Il est ainsi facile de les commenter ou de les archiver.

Pour ceux qui sont curieux de comprendre la façon dont un SGBDR est mis en œuvre, la modularité, les nombreux commentaires et la documentation du code de SQLite sont appréciables. Cela ne signifie pas que SQLite soit un modèle de la façon dont les autres moteurs de base de données sont mis en œuvre. Comprendre comment fonctionne SQLite permet simplement de comprendre plus rapidement les principes de fonctionnement des autres systèmes.

Expérimentations et prototypes

La simplicité et la modularité de conception de SQLite en font une plate-forme de choix pour prototyper et expérimenter de nouvelles fonctionnalités ou idées (d'utilisation ou d'extension du langage SQL lui-même).

Cas où un autre moteur est plus approprié

Applications client/serveur

Si vous prévoyez que plusieurs applications clientes accéderont à travers le réseau (local ou non) à une même base de données, alors orientez-vous plutôt vers un moteur de base de données client/serveur plutôt que vers SQLite. Ce dernier fonctionne sur un système de fichiers réseau, mais en raison de la latence associée à la plupart de ces systèmes, la performance ne sera pas au rendez-vous. De plus, la logique du mécanisme de verrouillage de fichier (*file locking*) de nombreux systèmes de fichiers contient des bugs (que ce soit sous Unix ou Windows). Si le verrouillage de fichiers ne fonctionne pas comme prévu, deux ou plusieurs clients pourraient alors modifier la même partie de la même base de données en même temps, entraînant une corruption de la base. Comme ce problème a pour origine la présence de bugs dans le système de fichiers sous-jacent, SQLite ne peut rien faire pour l'empêcher.

Une bonne règle générale est d'éviter d'utiliser SQLite dans des situations où la même base de données sera accessible simultanément à partir de plusieurs ordinateurs sur un réseau de fichiers.

Très gros sites web

SQLite fonctionne bien comme système de base de données pour site web. Mais si votre site devenait lent et que vous projetiez de déporter la partie base de données sur une deuxième machine, alors vous devriez sérieusement penser à utiliser un moteur client/serveur de type entreprise.

Très grosse base de données

Lorsque vous commencez une transaction dans SQLite, ce qui arrive automatiquement avant chaque opération qui n'est pas explicitement dans un `BEGIN…COMMIT`, le moteur doit créer une image de certains morceaux de la base en cours de modification afin de gérer son système d'annulation. SQLite a besoin de 256 octets de RAM pour chaque mégaoctet de base. Pour de petites bases, ce coût mémoire n'est pas un problème ; mais lorsque la base de données atteint le gigaoctet, la taille de cette image devient importante. Si vous avez besoin de stocker et modifier plus de quelques dizaines de mégaoctets, vous devriez envisager d'utiliser un autre moteur de base de données.

Forte demande d'accès simultanés

SQLite utilise des verrous de lecture/écriture sur l'ensemble de la base. Cela signifie que dès qu'un processus lit une partie de la base, tous les autres sont empêchés d'écrire dans n'importe quelle autre partie de celle-ci (et inversement). Dans bien des cas, ce n'est pas forcément un problème (une opération dure rarement plus de quelques dizaines de millisecondes). Mais pour les applications devant gérer beaucoup d'accès simultanés, il faudra se tourner vers une autre solution.

Créer et utiliser une base avec l'interpréteur SQLite

```
sqlite3 nom_de_fichier      // Unix/Linux/Mac OS X…
sqlite3.exe nom_de_fichier  // Windows
```

La bibliothèque SQLite inclut un utilitaire en ligne de commande appelé *sqlite3* (ou sqlite3.exe sous Windows) pour saisir et exécuter des requêtes SQL sur une base de donnée SQLite. Pour lancer ce programme, il suffit de taper `sqlite3` (ou `sqlite3.exe`) suivi du nom de fichier contenant la base de données. Si le fichier n'existe pas, ce programme le créera automatiquement. Une invite de commande apparaît ensuite dans laquelle vous pourrez saisir et exécuter des requêtes SQL.

L'exemple ci-après vous montre comment créer une base nommée `exemple1.db` avec une seule table `table1` contenant quelques champs.

```
C:\Projet1\> sqlite3.exe exmple1.db
SQLite version 3.6.6
Enter ".help" for instructions
sqlite> create table table1(un varchar(10), deux smallint);
sqlite> insert into table1 values('bonjour!',10);
sqlite> insert into table1 values('au revoir',20);
sqlite> select * from table1;
bonjour!|10
au revoir|20
sqlite>
```

Pour quitter l'interpréteur sqlite3, appuyez sur les touches Ctrl+D ou Ctrl+C. Vous pouvez aussi utiliser la commande spéciale `.exit` (attention au point au début de celle-ci).

Attention

Prenez garde à bien terminer chacune de vos requêtes par un point-virgule (;). C'est sa présence qui permet à l'interpréteur sqlite3 de déterminer la fin d'une requête. Si vous l'omettez, sqlite3 affichera une invite dite de continuation et attendra le reste de la requête.

L'exemple suivant montre comment se passe une requête multiligne :

```
sqlite> create table table2 (
   …>   f1 varchar(30) primary key,
   …>   f2 text,
   …>   f3 real
   …> );
sqlite>
```

L'interpréteur sqlite3 possède également quelques commandes internes, qui commencent toutes par le signe point (.) (les *dot commands*). Elles servent principalement à modifier le format d'affichage du résultat des requêtes ou à exécuter certaines instructions de requêtes préconditionnées. Le tableau ci-dessous fournit un récapitulatif de ces commandes.

Commandes internes (interpréteur sqlite3)

Commande	Description
`.help`	Affiche la liste des commandes spéciales
`.bail ON|OFF`	S'arrêter à la première erreur rencontrée. OFF par défaut.
`.databases`	Lister les noms et fichiers bases liées
`.dump ?TABLE? ...`	Faire une image (*dump*) de la base sous forme de requêtes SQL dans un fichier texte

Commandes internes (interpréteur sqlite3) *(Suite)*

Commande	Description
`.echo ON|OFF`	Activer ou désactiver l'affichage
`.exit`	Quitter l'interpréteur
`.explain ON|OFF`	Activer ou désactiver le mode de sortie applicable à `EXPLAIN`
`.header(s) ON|OFF`	Activer ou désactiver l'affichage des en-têtes
`.help`	Affiche la liste et la description des commandes spéciales
`.import FILE TABLE`	Importer les données de `FILE` dans la `TABLE`
`.indices TABLE`	Montrer le nom de tous les index de `TABLE`
`.load FILE ?ENTRY?`	Charger une bibliothèque d'extensions
`.mode MODE ?TABLE?`	Choisir le mode de sortie, où `MODE` est parmi la liste suivante : – csv : données séparées par des points-virgules – column : colonnes alignées à gauche (voir aussi `.width`) – html : sous forme de <table> HTML – insert : sous forme d'instruction d'insertion SQL pour `TABLE` – line : une valeur par ligne – list : valeurs séparées par la chaîne de séparation `.string` – tabs : valeurs séparées par des tabulations – tcl : sous forme d'éléments de liste TCL
`.nullvalue STRING`	Affiche `STRING` à la place des valeurs nulles (vides) `NULL`
`.output FILENAME`	Envoyer la sortie vers le fichier FILENAME
`.output stdout`	Envoyer la sortie vers l'écran/console
`.prompt MAIN CONTINUE`	Remplacer les invites (*prompts*) standard

Commandes internes (interpréteur sqlite3) *(Suite)*

Commande	Description
`.quit`	Quitter l'interpréteur
`.read FILENAME`	Exécuter la requête SQL présente dans le fichier `FILENAME`
`.schema ?TABLE?`	Afficher les instructions `CREATE`
`.separator STRING`	Changer le séparateur utilisé pour la sortie et l'import
`.show`	Montrer la valeur courante des différentes options
`.tables ?PATTERN?`	Lister les noms des tables correspondant à un modèle `LIKE`
`.timeout MS`	Essayer d'ouvrir une table verrouillée pendant `MS` millisecondes
`.width NUM NUM ...`	Spécifier la largeur des colonnes en mode « colonne »

Créer/ouvrir une base (SQLite)

```
int sqlite3_open(
    const char *filename,  /* Database filename (UTF-8) */
    sqlite3 **ppDb         /* OUT: SQLite db handle */
);
int sqlite3_open16(
  const void *filename,    /* Database filename (UTF-16) */
  sqlite3 **ppDb           /* OUT: SQLite db handle */
);
int sqlite3_open_v2(
  const char *filename,    /* Database filename (UTF-8) */
  sqlite3 **ppDb,          /* OUT: SQLite db handle */
  int flags,               /* Flags */
  const char *zVfs         /* Name of VFS module to use */
);
```

Ces routines permettent d'ouvrir une base de données SQLite dont le nom de fichier est donné en paramètre. Le nom de fichier et l'encodage par défaut de la base sont considérés comme étant de l'UTF-8 pour `sqlite3open()` et `sqlite3_open_v2()`, mais de l'UTF-16 pour `sqlite3_open16()`. Un pointeur vers une instance de `sqlite3` est renvoyé par le paramètre `ppDb` même si une erreur apparaît (à l'exception de l'impossibilité d'allouer la mémoire pour cette instance, auquel cas un pointeur NULL est retourné).

Info

Si le nom de fichier est une chaîne vide ou «`:memory:`» alors une base privée est créée en mémoire. Cette dernière disparaîtra à sa fermeture.

Si une base est ouverte (et/ou créée) avec succès, alors `SQLITE_OK` est retourné par la fonction ; sinon un code d'erreur est renvoyé. Les fonctions `sqlite3_errmsg()` ou `sqlite3_errmsg16()` permettent d'obtenir un descriptif de l'erreur en anglais.

Attention

Selon le code d'erreur, il peut être nécessaire de fermer correctement la connexion avec `sqlite3_close()`.

Le paramètre *flags* supplémentaire de `sqlite3_open_v2()` permet de spécifier l'option d'ouverture de la base :

- `SQLITE_OPEN_READONLY`. Ouvrir la base en lecture seule. Si la base n'existe pas, une erreur est renvoyée ;
- `SQLITE_OPEN_READWRITE`. Ouvrir la base en lecture/écriture. Si le fichier est en lecture seule, la base sera ouverte en lecture seule. Dans les deux cas, le fichier doit exister sinon une erreur est renvoyée ;

- `SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE`. Ouvrir la base en lecture/écriture, et créer celle-ci s'il n'existe pas. C'est le comportement par défaut des fonctions `sqlite3_open()` et `sqlite3_open16()`.

Il est possible de combiner ces trois modes d'ouverture avec l'une des deux autres options suivantes :

- `SQLITE_OPEN_NOMUTEX`. La base est ouverte avec le support du multithread (du moins si l'option de compilation *single-thread* n'a pas été activée ou que cette même option n'a pas été spécifiée) ;
- `SQLITE_OPEN_FULLMUTEX`. La base est ouverte avec le support du multithread à travers un mécanisme de sérialisation.

Le dernier paramètre *zVfs* de `sqlite3_open_v2()` permet de définir l'interface des opérations système que la connexion à la base SQLite doit utiliser (reportez-vous au manuel de SQLite pour plus d'informations).

Un exemple de mise en œuvre de la fonction `sqlite3_open()` est disponible à la section « Lancer une requête avec SQLite ».

Codes d'erreur des fonctions sqlite3_open*()

Code	Valeur	Description
`SQLITE_OK`	0	Pas d'erreur
`SQLITE_ERROR`	1	Erreur SQL ou base manquante
`SQLITE_INTERNAL`	2	Erreur logique interne à SQLite
`SQLITE_PERM`	3	Permission d'accès refusée
`SQLITE_ABORT`	4	Une fonction *callback* a demandé un abandon

Codes d'erreur des fonctions sqlite3_open*() *(Suite)*

Code	Valeur	Description
`SQLITE_BUSY`	5	Le fichier de base de données est verrouillé
`SQLITE_LOCKED`	6	Une table de la base est verrouillée
`SQLITE_NOMEM`	7	Un `malloc()` a échoué
`SQLITE_READONLY`	8	Tentative d'écriture sur une base en lecture seule
`SQLITE_INTERRUPT`	9	Opération terminée par `sqlite3_interrupt()`
`SQLITE_IOERR`	10	Une erreur d'E/S disque est survenue
`SQLITE_CORRUPT`	11	L'image de la base est corrompue
`SQLITE_NOTFOUND`	12	*Non utilisé.* Table ou enregistrement non trouvé.
`SQLITE_FULL`	13	Échec de l'insertion car base pleine
`SQLITE_CANTOPEN`	14	Impossible d'ouvrir le fichier de base de données
`SQLITE_PROTOCOL`	15	*Non utilisé.* Erreur du protocole de verrouillage de la base.
`SQLITE_EMPTY`	16	Base de données vide
`SQLITE_SCHEMA`	17	Le schéma de la base a changé
`SQLITE_TOOBIG`	18	La chaîne ou le BLOB dépasse la taille maximale autorisée
`SQLITE_CONSTRAIN`	19	Abandon dû à la violation d'une contrainte
`SQLITE_MISMATCH`	20	Type de donnée inadapté
`SQLITE_MISUSE`	21	Bibliothèque utilisée incorrectement

Codes d'erreur des fonctions sqlite3_open*() *(Suite)*

Code	Valeur	Description
SQLITE_NOLFS	22	Utilisation de spécificité système non supportée sur l'hôte
SQLITE_AUTH	23	Autorisation refusée
SQLITE_FORMAT	24	Erreur de format de la base auxiliaire
SQLITE_RANGE	25	2e paramètre de sqlite3_bind() hors borne
SQLITE_NOTADB	26	Le fichier à ouvrir n'est pas une base de données
SQLITE_ROW	100	sqlite3_step() a une autre ligne prête
SQLITE_DONE	101	sqlite3_step() a fini son exécution

Lancer une requête avec SQLite

```
int sqlite3_exec(
  sqlite3*,                  /* An open database */
  const char *sql,           /* SQL to be evaluated */
  int (*callback)(void*,int,char**,char**),
                             /* Callback */
  void *,                    /* 1st argument to callback */
  char **errmsg              /* Error msg written here */
);
```

La fonction sqlite3_exec() permet d'exécuter une ou plusieurs instructions SQL sans avoir à écrire beaucoup de code. Les instructions SQL (encodées en UTF-8) sont passées en deuxième paramètre. Ces instructions sont exécutées une à une jusqu'à rencontrer une erreur, ou que toutes soient exécutées. Le troisième paramètre est un

callback optionnel appelé une fois pour chaque ligne produite par l'exécution de toutes les instructions SQL fournies. Le cinquième paramètre précise où écrire un éventuel message d'erreur.

Attention

Tout message d'erreur renvoyé a été alloué à l'aide de la fonction `sql3_malloc()`. Pour éviter des pertes mémoire, vous devez impérativement libérer cette mémoire avec `sqlite3_free()`.

L'exemple suivant illustre la manière de lire les données d'une base. Pour toute autre opération, il suffit de créer la requête SQL nécessaire et de l'exécuter.

```
#include <stdio.h>
#include <sqlite3.h>

static int callback(void *NotUsed, int argc, char **argv,
                    ➥char **azColName)
{
    int i;
    for(i=0; i<argc; i++)
    {
        printf("%s = %s\n", azColName[i], argv[i] ?
               ➥argv[i] : "NULL");
    }
    printf("\n");
    return 0;
}

int main(int argc, char **argv)
{
    sqlite3 *db;
    char *zErrMsg = 0;
```

```
    int rc;
    if( argc!=3 )
    {
        fprintf(stderr, "Usage: %s DATABASE
        ➥ SQL-STATEMENT\n", argv[0]);
        exit(1);
    }
    rc = sqlite3_open(argv[1], &db);
    if( rc )
    {
        fprintf(stderr, "Can't open database: %s\n",
              ➥sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }
    rc = sqlite3_exec(db, argv[2], callback, 0, &zErrMsg);
    if( rc!=SQLITE_OK )
    {
        fprintf(stderr, "SQL error: %s\n", zErrMsg);
        sqlite3_free(zErrMsg);
    }
    sqlite3_close(db);
    return 0;
}
```

Exemple d'utilisation de l'API de SQLite

Fermer une base (SQLite)

```
int sqlite3_close(sqlite3 *);
```

Cette fonction est le destructeur d'un objet `sqlite3`.

Attention

Votre application doit terminer toutes les instructions préparées et fermer tous les pointeurs de BLOB associés avec l'objet `sqlite3` avant de le fermer. Si `sqlite3_close()` est invoquée alors qu'une transaction était ouverte, celle-ci sera automatiquement annulée.

Un exemple de mise en œuvre de la fonction `sqlite3_close()` est disponible dans la section « Lancer une requête avec SQLite ».

Astuce

La fonction `sqlite3_next_stmt()` peut être utilisée pour connaître toutes les instructions préparées associées à une connexion. Un exemple type pourrait être :

```
sqlite3_stmt *pStmt;
while ( (pStmt=sqlite3_next_stmt(db,0)) != 0 )
{
    Sqlite3_finalize(pStmt);
}
```

Créer une table (requête SQL)

```
CREATE TABLE NomDeLaTable
(NomColonne1 Type,
NomColonne1 Type,
…);
```

Cette requête permet de créer une nouvelle table dans la base de données active. Le nom de la table est à préciser après `CREATE TABLE`. Si le moteur sous-jacent supporte les

espaces, il faut encadrer le nom par des guillemets doubles («). Vient ensuite la description des colonnes entre parenthèses.

Il est possible d'attribuer des valeurs par défaut à chaque colonne. Cette valeur est utile lorsqu'aucune valeur n'est spécifiée pour ladite colonne au moment d'une insertion. Pour spécifier une valeur par défaut, il suffit d'ajouter `default V` après la définition du type de donnée, où `V` est la valeur par défaut souhaitée.

```
CREATE TABLE client
 (Nom char(50),
 Prenom char(50),
 Addresse char(50) default 'Inconnue',
 Ville char(50) default 'Paris',
 Pays char(25),
 Date_de_naissance date)
```

Les tableaux suivants recensent les différents types de données disponibles pour les bases SQLite, MySQL et Oracle.

Type SQLite

Type	Description
`NUMERIC`	Si la donnée peut être convertie en entier alors elle est stockée en tant que `INTEGER`, si elle peut être convertie en réel alors elle est stockée en tant que `REAL`, sinon elle est stockée en tant que `TEXT`. Aucune conversion en `BLOB` (abréviation de *Binary Large Object*) ou valeur vide (`NULL`) n'est faite.
`INTEGER`	La valeur est un entier stocké sur 1 à 8 octets en fonction de sa valeur
`REAL`	La valeur est un réel, stocké en représentation IEEE 8 octets
`TEXT`	La valeur est un texte, stocké dans l'encodage de la base (UTF-8, UTF-16-BE ou UTF-16-LE)
`BLOB`	La valeur est un objet binaire BLOB stocké exactement comme fourni

Type MySQL

Type	Taille	Description
`TINYINT [M] [UNSIGNED]`	1 octet	Ce type peut stocker des nombres entiers de –128 à 127 s'il ne porte pas l'attribut `UNSIGNED`, dans le cas contraire il peut stocker des entiers de 0 à 255
`SMALLINT [M] [UNSIGNED]`	2 octets	Ce type de données peut stocker des nombres entiers de –32 768 à 32 767 s'il ne porte pas l'attribut `UNSIGNED`, dans le cas contraire il peut stocker des entiers de 0 à 65 535
`MEDIUMINT [M] [UNSIGNED]`	3 octets	Ce type de données peut stocker des nombres entiers de –8 388 608 à 8 388 607 s'il ne porte pas l'attribut `UNSIGNED`, dans le cas contraire il peut stocker des entiers de 0 à 16 777 215
`INT [M] [UNSIGNED]`	4 octets	Ce type de données peut stocker des nombres entiers de –2 147 483 648 à 2 147 483 647 s'il ne porte pas l'attribut `UNSIGNED`, dans le cas contraire il peut stocker des entiers de 0 à 4 294 967 295
`INTEGER [M] [UNSIGNED]`	.	Même chose que le type `INT`
`BIGINT [M] [UNSIGNED]`	8 octets	Ce type de données stocke les nombres entiers allant de –9 223 372 036 854 775 808 à 9 223 372 036 854 775 807 sans l'attribut `UNSIGNED`, et de 0 à 18 446 744 073 709 551 615 avec
`FLOAT [UNSIGNED]`	4 ou 8 octets	Stocke un nombre de type flottant, en précision simple de 0 à 24 ou précision double de 25 à 53 (occupe 4 octets si la précision est inférieure à 24 ou 8 au-delà)

`FLOAT[(M,D)] [UNSIGNED]`	4 octets	`M` est le nombre de chiffres et `D` est le nombre de décimales. Ce type de données permet de stocker des nombres flottants à précision simple.Va de –1,175494351E-38 à 3,402823466E+38. Si `UNSIGNED` est activé, les nombres négatifs sont retirés mais cela ne permet pas d'avoir des nombres positifs plus grands.
`DOUBLE PRECISION[(M,D)]`	8 octets	Même chose que le type `DOUBLE`
`DOUBLE [(M,D)]`	8 octets	Stocke des nombres flottants à double précision de –1,7976931348623157E+308 à 1,7976931348623157E+308 (jusqu'à ±2,2250738585072014E–308 en se rapprochant de 0). Si `UNSIGNED` est activé, les nombres négatifs sont retirés mais cela ne permet pas d'avoir des nombres positifs plus grands.
`REAL[(M,D)]`	8 octets	Même chose que le type `DOUBLE`
`DECIMAL[(M[,D])]`	`M`+2 octets si `D` > 0, `M`+1 octets si `D` = 0	Contient des nombres flottants stockés comme des chaînes de caractères
`NUMERIC [(M,D)]`	.	Même chose que le type `DECIMAL`
`DATE`	3 octets	Stocke une date au format «AAAA-MM-JJ» allant de «1000-01-01» à «9999-12-31»

Type	Taille	Description
`DATETIME`	8 octets	Stocke une date et une heure au format «AAAA-MM-JJ HH:MM:SS» allant de «1000-01-01 00:00:00» à «9999-12-31 23:59:59»
`TIMESTAMP [M]`	4 octets	Stocke une date sous forme numérique allant de «1970-01-01 00:00:00» à l'année 2037. L'affichage dépend des valeurs de `M` : AAAAMMJJHHMMSS, AAMMJJHHMMSS, AAAAMMJJ, ou AAMMJJ pour `M` égal respectivement à 14, 12, 8, et 6.
`TIME`	3 octets	Stocke l'heure au format «HH:MM:SS», allant de «-838:59:59» à «838:59:59»
`YEAR`	1 octet	Année à 2 ou 4 chiffres allant de 1901 à 2155 (4 chiffres) et de 1970-2069 (2 chiffres)
`[NATIONAL] CHAR(M) [BINARY]`	`M` octets	Chaîne de 255 caractères maximum remplie d'espaces à la fin. L'option `BINARY` est utilisée pour tenir compte de la casse.
`BIT`	1 octet	Même chose que `CHAR(1)`
`BOOL`	1 octet	Même chose que `CHAR(1)`
`CHAR (M)`	`M` octets	Stocke des caractères. Si vous stockez un caractère et que `M` vaut 255, la donnée prendra 255 octets. Autant donc employer ce type de données pour des mots de longueur identique.
`VARCHAR (M) [BINARY]`	L+1 octets	Ce type de données stocke des chaînes de 255 caractères maximum. L'option `BINARY` permet de tenir compte de la casse (L représente la longueur de la chaîne).

`TINYBLOB`	L+1 octets	Stocke des chaînes de 255 caractères maximum. Ce champ est sensible à la casse (L représente la longueur de la chaîne).
`TINYTEXT`	L+1 octets	Stocke des chaînes de 255 caractères maximum. Ce champ est insensible à la casse.
`BLOB`	L+1 octets	Stocke des Chaînes de 65 535 caractères maximum. Ce champ est sensible à la casse.
`TEXT`	L+2 octets	Stocke des chaînes de 65 535 caractères maximum. Ce champ est insensible à la casse (L représente la longueur de la chaîne).
`MEDIUMBLOB`	L+3 octets	Stocke des chaînes de 16 777 215 caractères maximum.
`MEDIUMTEXT`	L+3 octets	Chaîne de 16 777 215 caractères maximum. Ce champ est insensible à la casse.
`LONGBLOB`	L+4 octets	Stocke des chaînes de 4 294 967 295 caractères maximum. Ce champ est sensible à la casse.
`LONGTEXT`	L+4 octets	Stocke des chaînes de 4 294 967 295 caractères maximum.
`ENUM('valeur_possible1', 'valeur_possible2', 'valeur_possible3',...)`	1 ou 2 octets (la place occupée est fonction du nombre de solutions possibles : 65 535 valeurs maximum.	
`SET('valeur_possible1', 'valeur_possible2',...)`	1, 2, 3, 4 ou 8 octets selon de nombre de solutions possibles (de 0 à 64 valeurs maximum).	

Types de données internes (built-in) d'Oracle

Code interne Type	Description	Minimum	Maximum	Exemples de valeurs
`VARCHAR2 (taille)`	Chaîne de caractères de longueur variable	1 car.	2000 car.	'A'
`NVARCHAR2 (taille)`	Chaîne de caractères de longueur variable utilisant le jeu de car. national			'Bonjour DD'
`NUMBER [(taille[,precision])]`	Numérique (prec<=38, exposant max -84 +127)	10-84	10127	10.9999
`LONG (taille)`	Chaîne de caractères de longueur variable	1 car.	2 giga car.	'AAAHHHHHH...HHH'
`DATE`	Date (du siècle à la seconde)	01/01/-4712 (avant J.-C.)	31/12/9999	'10-FEB-04' '10/02/04' (dépend du format d'affichage ou du paramétrage local)
`RAW (taille)`	Données binaires devant être entrées en notation hexadécimale. Taille : 1 à 255 caractères	1 octet	2 000 octets	
`LONG RAW (taille)`	Données binaires devant être entrées en notation hexadécimale	1 octet	2 Go	
`ROWID`	Type réservé à la pseudo-colonne ROWID. Ne peut être utilisé.			
`CHAR (taille)`	Chaîne de caractères de longueur fixe	1 car.	255 car.	'AZERTY'
`NCHAR (taille)`	Chaîne de caractères de longueur fixe utilisant le jeu de car. national			'W'

`CLOC` `NCLOB`	LOB de type caractère mono-octet ou multi-octet utilisant le jeu de car. national	1 octet	4 giga car.
`BLOB`	BLOB ! gros objet binaire	1 octet	4 giga-octets
`BFILE`	Pointeur vers un fichier binaire externe	1 octet	4 giga-octets
`TIMESTAMP (fractional_seconds_precision)`	Date (année, mois, et jour plus heure, minute, secondes et quantité de fraction de seconde), où `fractional_seconds_precision` est le nombre de chiffres de la partie fractionnaire des secondes. Les valeurs vont de 0 à 9 (1 pour le dixième, 2 pour le centième, …). La valeur par défaut est 6 (le millionième).		
`TIMESTAMP (fractional_seconds_precision) WITH LOCAL TIME ZONE`	AVEC FUSEAU HORAIRE LOCAL (*LOCAL TIME ZONE*) Toutes les valeurs données dans le fuseau horaire local, avec l'exception suivante : les données sont normalisées dans le fuseau de la base lorsqu'elles y sont stockées. Lorsque les données sont relues, elles sont converties dans le fuseau horaire local.		

Code interne	Type	Description	Minimum	Maximum	Exemples de valeurs
`INTERVAL YEAR (year_precision) TO MONTH`		Stocke en période de temps en années et mois, où `year_precision` est le nombre chiffres pour le nombre d'années. De 0 à 9 chiffres après la virgule. La valeur par défaut est 2.			
`INTERVAL DAY (day_precision) TO SECOND (fractional_seconds_precision)`		Stocke une période de temps en jours, heures, minutes et secondes, où `day_precision` est le nombre maximum de chiffres pour le nombre de jours (de 0 à 9, et 2 par défaut) et `fractional_seconds_precision` est le nombre de chiffre après la virgule pour le nombre de secondes (de 0 à 9, et 6 par défaut).			
`UROWID [(size)]`		Adresse logique d'une ligne (chaîne de caractère en base 64) d'une table organisée en index. La taille (optionnelle) est celle de la colonne de type UROWID. La taille maximale est de 4 000 octets.			

Accéder aux données d'une table (requête SQL)

```
SELECT * FROM NomDeLaTable
SELECT NomDeColonne[, …] FROM NomDeLaTable
```

Pour parcourir les données d'une table, utilisez l'instruction `SELECT … FROM …` Pour garder toutes les colonnes, utilisez le caractère générique `*`.

Pour trier les données, utilisez l'option `ORDER BY` suivie du ou des noms de colonnes. L'instruction `GROUP BY` pour regrouper des données entre elles.

En supposant que la table interrogée contienne toutes les ventes (montant de la vente dans la colonne « Ventes ») effectuées par toutes les boutiques (nom de la boutique dans la colonne « Boutique » et ville de la boutique dans la colonne « Ville »), la requête suivante donne la recette de chaque boutique en les classant d'abord par ville puis par boutique :

```
SELECT Ville, Boutique, SUM(Ventes)
FROM InfoBoutique
GROUP BY Boutique
ORDER_BY Ville, Boutique
```

La clause `WHERE` permet de filtrer les données pour ne choisir que celles répondant à un certain critère. La clause `DISTINCT` permet de ne faire apparaître qu'une seule fois des entrées identiques.

Par exemple, la requête suivante permet de sélectionner tous les magasins ayant effectué une vente dont le montant se situe entre 500 et 700 euros. La clause `DISTINCT` permet

de n'obtenir qu'une seule fois le magasin (qui peut avoir effectué plusieurs ventes de ce montant) :

```
SELECT DISTINCT Boutique
FROM InfoBoutique
WHERE Ventes >= 500 AND Ventes <= 700
```

Cette requête peut aussi s'écrire à l'aide de l'instruction `BETWEEN` :

```
SELECT DISTINCT Boutique
FROM InfoBoutique
WHERE Ventes BETWEEN 500 AND 700
```

Enfin, à supposer qu'une autre table contienne la région dans laquelle se trouve une boutique, il est possible de connaître les ventes par région en utilisant une *jointure interne* avec la requête suivante :

```
SELECT DISTINCT A1.nom_de_region REGION, SUM(A2.
Ventes) VENTES
FROM Geographie A1, InfoBoutique A2
WHERE A1.Boutique == A2.Boutique
GROUP BY A1.nom_de_region
```

Ces quelques instructions SQL vous permettront de débuter rapidement. Pour aller plus loin, jetez un œil sur le site mentionné en introduction (**http://sql.1keydata.com/fr**).

Définir un environnement ODBC (wxWidgets)

```
#include <wx/dbtable.h>
class wxDbConnectInf;
```

Pour se connecter à une source ODBC, il faut fournir au moins trois informations : un nom de source, un identifiant d'utilisateur et un mot de passe. Une quatrième information, un dossier par défaut indiquant où sont stockées les données, est également à fournir pour des bases de données au format texte ou dBase. La classe `wxDbConnectInf` est conçue pour embarquer ces données, ainsi que d'autres nécessaires pour certaines sources ODBC.

Le membre `Henv` est la clé (*handle* d'environnement) pour accéder à une base. Le `Dsn` fourni doit impérativement et exactement correspondre au nom de la base, tel qu'il apparaît dans la source ODBC (dans le panneau d'administration ODBC sous Windows, ou dans le fichier .odbc.ini par exemple). L'`Uid` est l'identification utilisateur à utiliser pour se connecter à la base. Il doit exister dans la base vers laquelle vous souhaitez vous connecter. Il déterminera les droits et privilèges de la connexion. Certaines sources ODBC sont sensibles à la casse, faites donc attention… L'`AutStr` est le mot de passe de l'`Uid`, là encore sensible à la casse. Le membre `defaultDir` est utile pour les bases de données de type fichier. C'est par exemple le cas des bases dBase, FoxPro ou fichier texte. Il contiendra un chemin absolu, dans lequel vous aurez intérêt à utiliser des '/' plutôt que des '\' par souci de portabilité.

Se connecter à une base (wxWidgets)

```
class wxDb;
wxDb* wxDbGetConnection(wxDbConnectInf*);
```

Il y a deux façons de se connecter à une base. Vous pouvez :

- créer une instance de `wxDb` « à la main » et ouvrir vous-même la connexion ;
- utiliser les fonctions de mise en cache fournies avec les classes `wxODBC` pour créer, maintenir et détruire les connexions.

Quelle que soit la méthode choisie, il faut d'abord créer un objet `wxDbConnectInf`. Le code suivant montre comment lui fournir tous les paramètres nécessaires à la future connexion :

```
wxDbConnectInf dbConnectInf;
dbConnectInf.SetDsn("MonDsn");
dbConnectInf.SetUserID("MonNomDUtilisteur");
dbConnectInf.SetPassword("MonMotDePasse");
dbConnectInf.SetDefaultDir("");
```

Pour allouer l'environnement de connexion, la classe `wxDbConnectInf` possède une méthode à invoquer, comme le montre la suite de l'exemple. Une valeur de retour à faux signifie un échec de l'allocation et le *handle* est alors indéfini.

```
if (dbConnectInf.AllocHenv())
{
    wxMessageBox("Impossible d'allouer l'environnement ODBC",
    ➥"ERROR de CONNEXION", wxOK | wxICON_EXCLAMATION);
    return;
}
```

Un moyen plus concis consiste à utiliser directement le constructeur :

```
wxDbConnectInf *dbConnectInf = new wxDbConnectInf(NULL,
➥"MonDsn","MonNomDUtilisteur", "MonMotDePasse",  "" );
```

Dès que l'environnement est créé, vous n'avez plus à vous préoccuper de ce *handle*, mais seulement à l'utiliser. Il sera libéré lors de la destruction de l'instance de la classe.

Nous parlions tout à l'heure de deux manières de créer une connexion. La première, manuelle, consiste à créer une instance de `wxDb` et à l'ouvrir :

```
wxDb* db = newDb(dbConnectInf->GetHenv());
bool opened = db->Open(dbConnectInf);
```

L'autre solution, plus avantageuse, consiste à utiliser le système de mise en cache fourni. Il offre les mêmes possibilités qu'une connexion manuelle, mais gère automatiquement la connexion, de sa création au nettoyage lors de sa fermeture en passant par sa réutilisation. Et cela vous évite bien sûr de la coder vous-même. Pour utiliser ce mécanisme, appelez simplement la fonction `wxDbGetConnection()` :

```
wxDb* db = newDbGetConnection(dbConnectInf);
```

La valeur retournée pointe ainsi sur un `wxDb` à la fois initialisé et ouvert. Si une erreur survient lors de la création ou de l'ouverture, le pointeur retourné sera nul. Pour fermer cette connexion, utilisez `wxDbFreeConnection(db)`. Un appel à `wxDbCloseConnections()` vide le cache en fermant et détruisant toutes les connexions.

Créer la définition de la table et l'ouvrir (wxWidgets)

```
class wxDbTable;
wxDbTable::wxDbTable(wxDb *pwxDb, const wxString
➥&tblName, const UWORD numColumns, const wxString
➥&qryTblName = "", bool qryOnly = !wxDB_QUERY_ONLY,
➥ const wxString &tblPath = "")
```

Il est possible d'accéder au contenu des tables d'une base directement à travers les nombreuses fonctions membres de la classe `wxDb`. Heureusement, il existe une solution plus simple, grâce à la classe `wxDbTable` qui encapsule tous ces appels.

La première étape consiste à créer une instance de cette classe, comme le montre le code suivant :

```
wxDbTable* table = new wxDbTable(db, tableName,
 ➥numTableColumns, "", !wxDBQUERY_ONLY, "");
```

Le premier paramètre est un pointeur sur une connexion à une base (`wxDb*`) ; le deuxième est le nom de la table ; le troisième est le nombre de colonnes de la table (il ne doit pas inclure la colonne ROWID si vous utilisez Oracle). Viennent ensuite trois paramètres optionnels. `qryTableName` est le nom de la table ou de la vue sur laquelle les requêtes porteront. Il permet d'utiliser la vue au lieu de la table elle-même. Cela permet d'obtenir de meilleures performances dans le cas où les requêtes impliquent plusieurs tables avec plusieurs jointures. Néanmoins, toutes les requêtes `INSERT`, `UPDATE` et `DELETE` seront faites sur la table de base. `qryOnly` indique si la table sera utilisée uniquement

en lecture ou non. `tblPath` est le chemin indiquant où est stockée la base. Cette information est indispensable pour certains types de bases, comme dBase.

Info

Une table ouverte en lecture seule offre de meilleures performances et une utilisation mémoire moindre. Si vous êtes certain de ne pas avoir à modifier son contenu, n'hésitez pas à l'ouvrir dans ce mode.

De plus, une table en lecture seule utilise moins de curseurs, et certains types de base sont limités en nombre de curseurs simultanés – une raison supplémentaire de faire attention.

Lorsque vous définissez les colonnes à récupérer, vous pouvez conserver de une à toutes les colonnes de la table.

```
table->SetColDefs(0, "FIRST_NME", DB_DATA_TYPE_VARCHAR,
 ➥FirstName, SQL_C_WXCHAR, sizeof(FirstName), true, true);
table->SetColDefs(
    1, // numéro de colonne dans la table
    "LAST_NAME", // nom de la colonne dans la base/requête
    DB_DATA_TYPE_VARCHAR, // type de donnée
    LastName, // pointeur sur la variable à lier
    SQL_C_WXCHAR, // type de conversion
    sizeof(LastName), // taille max de la variable liée
    true, // optionnel, indique si c'est une clé (faux
    ➥ par défaut)
    true // optionnel, mise à jour autorisée (vrai
    ➥ par défaut)
    );
```

La définition des colonnes commence à l'indice 0 jusqu'au nombre – 1 de colonnes spécifié lors de la création de l'instance de `wxDbTable`. Les lignes de codes ci-avant lient les colonnes mentionnées de la table à des variables de l'application. Ainsi, lorsque l'application appelle `wxDbTable::GetNext()` (ou n'importe quelle fonction récupérant des données dans l'ensemble résultat), les variables liées aux colonnes contiendront les valeurs correspondantes.

Les variables liées ne contiennent aucune valeur tant qu'aucune `Get...()` n'est appelé. Même après un appel à `Query()` le contenu des variables liées est indéterminé. Vous pouvez avoir autant d'instances de `wxDbTable` que souhaité sur la même table.

L'ouverture de la table se contente de vérifier si la table existe vraiment, que l'utilisateur a au moins un privilège de lecture (clause `SELECT`), réserve les curseurs nécessaires, lie les colonnes aux variables comme spécifié et construit une instruction d'insertion par défaut (clause `INSERT`) si la table n'est pas ouverte en lecture seule.

```
if (!table->Open())
{
    // une erreur est survenue lors de l'ouverture
}
```

La seule raison « réelle » d'un échec est que l'utilisateur n'a pas les privilèges requis. D'autres problèmes, comme l'impossibilité de lier les colonnes aux variables seront plutôt dus à des problèmes de ressources (comme la mémoire). Toute erreur générée par `wxDbTable::Open()` est journalisée dans un fichier si les fonctions de journalisation (*logging*) SQL sont activées.

Utiliser la table (wxWidgets)

```
bool WxDbTable::Query();
```

Pour utiliser la table et la définition ainsi créée, il faut maintenant définir les données à collecter.

Le code suivant montre comment charger une requête SQL dans cette table :

```
table->SetWhereClause("FIRST_NAME = 'GEORGE'");
table->SetOrderByClause("LAST_NAME, AGE");
table->SetFromClause("");
```

Si vous souhaitez obtenir un classement inversé ajoutez `DESC` après le nom de la colonne, par exemple : `LAST_NAME DESC`. La clause `FROM` permet de créer des jointures. Lorsqu'elle est vide, on utilise directement la table de base.

Après avoir spécifié tous les critères nécessaires, il suffit de lancer la requête, comme le montre le code suivant :

```
if (!table->Query())
{
    // erreur survenue pendant l'exécution de la requête
}
```

Bien souvent, l'erreur provient d'un problème de syntaxe dans la clause `WHERE`. Pour trouver l'erreur, regardez le contenu du membre `erreurList[]` de la connexion. En cas de succès, cela signifie que la requête s'est bien déroulée, mais le résultat peut être vide (cela dépend de la requête).

Il est maintenant possible de récupérer les données. Le code suivant montre comment lire toutes les lignes du résultat de la requête :

```
wxString msg;
while (table->GetNext())
{
    msg.Printf("Line #%lu - Nom: %s Prénom: %s",
    ➥table->GetRowNum(), FirstName, LastName);
    ➥ wxMessageBox(msg, "Données", wxOK | wxICON_
    ➥ INFORMATION, NULL);
}
```

Fermer la table (wxWidgets)

```
if (table)
{
    delete table;
    table = NULL;
}
```

Fermer une table est aussi simple que de détruire son instance. La destruction libère tous les curseurs associés, détruit les définitions de colonnes et libère les *handle*s SQL utilisés par la table, mais pas l'environnement de connexion.

Fermer la connexion à la base (wxWidgets)

```
wxDb::Close();
wxDbFreeConnection(wxDb*);
wxDbCloseConnections();
```

Lorsque toutes les tables utilisant une connexion donnée ont été fermées, vous pouvez fermer cette connexion. La méthode à employer dépend de la façon dont vous avez créé cette connexion.

Si la connexion a été créée manuellement, vous devez la fermer ainsi :

```
if (db)
{
    db->Close();
    delete db;
    db = NULL;
}
```

Si elle a été obtenue par la fonction `wxDbGetConnection()` qui gère un système de cache, vous devez alors fermer cette connexion comme suit :

```
if (db)
{
    wxDbFreeConnection(db);
    db = NULL;
}
```

Notez que le code ci-dessus libère juste la connexion de telle sorte qu'elle puisse être réutilisée par un prochain appel à `wxDbGetConnection()`. Pour libérer toutes les ressources, vous devez appeler le code suivant :

```
wxDbCloseConnections();
```

Libérer l'environnement ODBC (wxWidgets)

```
wxDbConnectInf::FreeHenv();
```

Une fois toutes les connexions fermées, vous pouvez libérer l'environnement ODBC :

```
dbConnectInf->FreeHenv();
```

Si vous avez utilisé la version du constructeur avec tous les paramètres, la destruction de l'instance libère l'environnement automatiquement :

```
delete dbConnectInf;
```

16

XML

XML (*Extensible Markup Language*) est un langage informatique de balisage générique de plus en plus répandu. Il existe principalement deux types d'API pour la manipulation de fichiers XML :

- **DOM** (*Document Object Modeling*). Constitue un objet mémoire de la totalité d'un document XML. Cette API permet un accès direct à tous les nœuds de l'arbre (éléments, texte, attributs) pour les lire, les modifier, les supprimer ou en ajouter. Il est par exemple très utilisé avec JavaScript dans les navigateurs web.
- **SAX** (*Simple API for XML*). Système de traitement séquentiel. Il représente une réelle alternative à DOM pour les fichiers XML volumineux. Lors de la lecture d'un fichier XML avec SAX, il est possible de réagir à différents événements comme l'ouverture et la fermeture d'une balise. Il est également possible de générer des événements SAX, par exemple lors de la lecture du contenu d'une base de données, afin de produire un document XML.

Il existe différentes bibliothèques permettant de manipuler des documents XML. En voici quelques-unes parmi les plus répandues, en rapport avec le C++ :

- **MSXML** (*Microsoft Core XML Services*). Très répandue, ne fonctionne toutefois que sous Windows.
- **libXML2** (**http://www.xmlsof.org**). Bibliothèque C très répandue dans le monde GNU/Linux. Disponible pour Linux, Winodws, Solaris, Mac OS X, HP-UX, AIX. Vous pouvez bien sûr l'utiliser avec C++.
- **Xerces-C++** (**http://xerces.apache.org/xerces-c**). Implémente des analyseurs (*parser*) DOM, SAX et SAX2. Disponible pour Windows, Unix/Linux/Mac OS X et Cygwin.
- **eXpat** (**http://expat.sourceforge.net**). Bibliothèque écrite en C utilisée par Firefox (disponibles sur les mêmes plate-formes que Firefox). Il existe des ponts (*wrappers*) C++ de cette bibliothèque.

D'autres bibliothèques haut niveau comme QT (**http://www.trolltech.com**) et wxWxWidgets (**http://www.wxwidgets.org**) fournissent également leur API pour le traitement des fichiers XML. Ce chapitre présente l'utilisation des objets fournis par wxWidgets. Ils utilisent un modèle DOM.

Charger un fichier XML

```
#include <wx/xml/xml.h>
class WXDLLIMPEXP_XML wxXmlDocument : public wxObject
{
public:
    wxXmlDocument();
    wxXmlDocument(const wxString& filename,
                  const wxString& encoding = wxT("UTF-8"));
```

```
    wxXmlDocument(wxInputStream& stream,
             const wxString& encoding = wxT("UTF-8"));
    virtual ~wxXmlDocument();

    wxXmlDocument(const wxXmlDocument& doc);
      wxXmlDocument& operator=(const wxXmlDocum
     // Lit un fichier XML et charge son contenu.
    // Retourne TRUE en cas de succès, FALSE sinon.
    virtual bool Load(const wxString& filename,
                 const wxString& encoding = wxT("UTF-8"),
                 int flags = wxXMLDOC_NONE);
    virtual bool Load(wxInputStream& stream,
                 const wxString& encoding = wxT("UTF-8"),
                 int flags = wxXMLDOC_NONE);

    // Enregistre le document dans un fichier XML.
    virtual bool Save(const wxString& filename,
                 int indentstep = 1) const;
    virtual bool Save(wxOutputStream& stream,
                 int indentstep = 1) const;
    bool IsOk() const;

    // Retourne le nœud racine du document.
    wxXmlNode *GetRoot() const;

    // Retourne la version du document (peut être vide).
    wxString GetVersion() const;

    // Retourne l'encodage du document (peut être vide).
    // Note : il s'agit de l'encodage du fichier,
    // et non l'encodage une fois chargé en mémoire !
     wxString GetFileEncoding() const;

     // Méthodes d'écriture :
     wxXmlNode *DetachRoot();
     void SetRoot(wxXmlNode *node);
     void SetVersion(const wxString& version);
     void SetFileEncoding(const wxString& encoding);

#if !wxUSE_UNICODE
    // Retourne l'encodage de la représentation mémoire du
```

```
    // document (le même que fourni à la fonction Load
    // ou au constructeur, UTF-8 par défaut).
    // NB : cela n'a pas de sens avec une compilation
    // Unicode où les données sont stockées en wchar_t*
    wxString GetEncoding() const;
    void SetEncoding(const wxString& enc);
#endif

private:
    // ...
};
```

La classe `wxXmlDocument` utilise la bibliothèque *expat* pour lire et charger les flux XML. L'exemple suivant charge un fichier XML en mémoire :

```
wxXmlDocument doc;
if (!doc.Load(wxT("fichier.xml")))
{
    // Problème lors du chargement
}
```

Après cela, votre fichier se trouve en mémoire sous forme arborescente. Si vous souhaitez garder tous les espaces et les indentations, vous pouvez désactiver leur suppression automatique. L'exemple suivant montre comment :

```
wxXmlDocument doc;
if (!doc.Load(wxT("fichier.xml"), wxT("UTF-8"),
            ➥wxXMLDOC_KEEP_WHITESPACE_NODES))
{
    // Problème lors du chargement
}
```

Vous pouvez faire de même lors de l'enregistrement du document dans un fichier :

```
doc.Save(wxT("fichier.xml"), wxXMLDOC_KEEP_WHITESPACE
➥_NODES);
```

Manipuler des données XML

```
// Propriété(es) d'un nœud XML.
// Exemple : in <img src=»hello.gif» id=»3»/>
// «src» est une propriété ayant «hello.gif» pour
// valeur et «id» une autre ayant la valeur «3».

class WXDLLIMPEXP_XML wxXmlProperty
{
public:
    wxXmlProperty();
    wxXmlProperty(const wxString& name, const
    ➥ wxString& value, wxXmlProperty *next = NULL);
    virtual ~wxXmlProperty();

    wxString GetName() const;
    wxString GetValue() const;
    wxXmlProperty *GetNext() const;

    void SetName(const wxString& name);
    void SetValue(const wxString& value);
    void SetNext(wxXmlProperty *next);
private:
    // ...
};
```

```
// Un nœud d'un document XML.
class WXDLLIMPEXP_XML wxXmlNode
{
public:
    wxXmlNode();
    wxXmlNode(wxXmlNode*      parent,
              wxXmlNodeType   type,
              const wxString& name,
              const wxString& content = wxEmptyString,
              wxXmlProperty*  props   = NULL,
              wxXmlNode*      next    = NULL);
    wxXmlNode(wxXmlNodeType   type,
              const wxString& name,
              const wxString& content = wxEmptyString);
    virtual ~wxXmlNode();

    // Constructeur par copie et opérateur de copie.
    wxXmlNode(const wxXmlNode& node);
    wxXmlNode& operator=(const wxXmlNode& node);

    virtual void AddChild(wxXmlNode *child);
    virtual bool InsertChild(wxXmlNode *child,
                             wxXmlNode *followingNode);
#if wxABI_VERSION >= 20808
    bool InsertChildAfter(wxXmlNode *child,
                          wxXmlNode *precedingNode);
#endif
    virtual bool RemoveChild(wxXmlNode *child);
    virtual void AddProperty(const wxString& name,
                             const wxString& value);
    virtual bool DeleteProperty(const wxString& name);

    // Accesseurs :
    wxXmlNodeType GetType() const;
    wxString GetName() const;
    wxString GetContent() const;

    int GetDepth(wxXmlNode *grandparent = NULL) const;

    bool IsWhitespaceOnly() const;
```

```
    // Récupère le contenu d'un nœud wxXML_ENTITY_NODE
    // En effet, <tag>contenu<tag> est représenté comme
    // wxXML_ENTITY_NODE name="tag", content=""
    //    __ wxXML_TEXT_NODE or
    //       wxXML_CDATA_SECTION_NODE name=""
           content="contenu"
    wxString GetNodeContent() const;

    wxXmlNode *GetParent() const;
    wxXmlNode *GetNext() const;
    wxXmlNode *GetChildren() const;

    wxXmlProperty *GetProperties() const;
    bool GetPropVal(const wxString& propName,
                    wxString *value) const;
    wxString GetPropVal(const wxString& propName,
                        const wxString& defaultVal)
const;
    bool HasProp(const wxString& propName) const;

    void SetType(wxXmlNodeType type);
    void SetName(const wxString& name);
    void SetContent(const wxString& con);

    void SetParent(wxXmlNode *parent);
    void SetNext(wxXmlNode *next);
    void SetChildren(wxXmlNode *child);

    void SetProperties(wxXmlProperty *prop);
    virtual void AddProperty(wxXmlProperty *prop);

private:
    // ...
```

Un nœud XML (`wxXmlNode`) a un nom et peut avoir un contenu et des propriétés. La plupart des nœuds sont de type `wxXML_TEXT_NODE` (pas de nom ni de propriétés) ou `wxXML_ELEMENT_NODE`. Par exemple, avec `<title>hi</title>`,

on obtient un nœud élément ayant pour nom « title » (sans contenu) possédant pour seul fils un nœud texte ayant pour contenu « hi ».

Si `wxUSE_UNICODE` vaut `0`, toutes les chaînes seront encodées avec la page de codes spécifiée lors de l'appel à la fonction `Load()` du document XML (UTF-8 par défaut).

Info

Le constructeur par copie affecte toujours le pointeur sur le parent et le pointeur sur le suivant à NULL.

L'opérateur d'affectation (=) ne recopie ni le pointeur sur le parent ni le pointeur sur le suivant, mais les laissent inchangés. Par contre, il recopie toute l'arborescence du nœud concerné (fils et propriétés).

L'exemple suivant illustre comment utiliser ces différentes classes XML :

```
wxXmlDocument doc;
if ( ! doc.Load(wxT("myfile.xml")) )
    return false;

// Début du traitement du fichier XML
if ( doc.GetRoot()->GetName() != wxT("mon-noeud-racine") )
    return false;

wxXmlNode *child = doc.GetRoot()->GetChildren();
while (child)
{
    if (child->GetName() == wxT("tag1"))
    {
        // process text enclosed by <tag1></tag1>
        wxString content = child->GetNodeContent();

        ...

        // process properties of <tag1>
        wxString propvalue1 =
```

```
            ➥child->GetPropVal(wxT("prop1"),
                                wxT("default-value"));
        wxString propvalue2 =
            ➥child->GetPropVal(wxT("prop2"),
                                wxT("default-value"));

        ...
    }
    else if (child->GetName() == wxT("tag2"))
    {
        // process tag2 ...
    }
    child = child->GetNext();
}
```

Le contenu d'un nœud diffère selon son type. Le tableau suivant vous aidera à savoir de quel type de nœud il s'agit.

wxXmlNodeType : types de nœud XML

Nom*	Valeur *
wxXML_ELEMENT_NODE	1
wxXML_ATTRIBUTE_NODE	2
wxXML_TEXT_NODE	3
wxXML_CDATA_SECTION_NODE	4
wxXML_ENTITY_REF_NODE	5
wxXML_ENTITY_NODE	6
wxXML_PI_NODE	7
wxXML_COMMENT_NODE	8
wxXML_DOCUMENT_NODE	9
wxXML_DOCUMENT_TYPE_NODE	10
wxXML_DOCUMENT_FRAG_NODE	11
wxXML_NOTATION_NODE	12
wxXML_HTML_DOCUMENT_NODE	13

* Ces noms et valeurs correspondent à l'énumération `xmlElementType` de la bibliothèque libXML.

Annexe A

Bibliothèques et compilateurs

Compilateurs

Borland Turbo C++ et C++ Builder

Turbo C++ : **http://cc.codegear.com/free/turbo**, gratuit

C++ Builder : **http://www.codegear.com/products/cppbuilder**, commercial

Systèmes : Windows

La qualité des compilateurs Borland est bien connue. C++ Builder est un vrai IDE (environnement de développement intégré) RAD, comme Delphi mais en C++. Un must. La version 2009 inclut le support du C++0x.

Microsoft Visual Studio

Site : **http://msdn.microsoft.com/en-us/visualc/default.aspx**

Systèmes : Windows

Depuis quelque temps maintenant, la version express de Visual Studio C++ est gratuite. Elle peut même être utilisée pour créer des applications commerciales.

GCC – GNU Compiler Collection

Site : **http://gcc.gnu.org**

Systèmes : Windows (cygwin et mingw), Linux

Le compilateur libre par excellence.

Intel C++

Site : **http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm**

Systèmes : Windows, Linux, Mac OS X

Processeur : Intel seulement

Pour ceux dont les performances du code produit sont essentielles…

IDE et RAD

Dev-C++ – IDE et compilateur

Site : **http://www.bloodshed.net/devcpp.html**

http://wxdsgn.sourceforge.net (version RAD avec wxWidgets)

Licence : Sources de l'application (en Delphi) disponibles en GPL

Systèmes : Windows

Compilateurs : IDE pour Mingw ou GCC

Dev-C++ est un IDE libre pour programmer en C/C++. Facile d'installation (une version inclut même le compilateur Mingw) et pratique (intégration du débogueur GDB),

il est le compagnon idéal pour ceux qui veulent un IDE simple et rapide. Et il est parfait pour ceux qui veulent débuter rapidement.

WxDevC++ est plus à jour que DevC++. De plus, cette version contient les packs WxWindows installés par défaut.

KDevelop – IDE

Site : **http://www.kdevelop.org**

Licence : GPL

Systèmes : Linux, Solaris, Unix

Compilateurs : GCC

Le projet KDevelop a été mis en place en 1998 pour bâtir un IDE pour KDE facile à utiliser. Depuis, KDevelop est disponible pour le public sous la licence GPL et supporte beaucoup de langages de programmation.

KDE étant développé avec la bibliothèque QT, cet environnement va peut-être enfin devenir disponible pour Windows. À suivre… car c'est un IDE très largement utilisé et d'une grande qualité.

Ultimate++ – Bibliothèque graphique et suite RAD

Site : **http://www.ultimatepp.org**

Licence : BSD

Systèmes : Windows, Linux, (version Mac OS X *via* X11 en cours de développement ; WinCE prévue ; Solaris et autres système BSD envisagées)

Compilateurs : MingGW,Visual (2003+), GCC

Ultimate++ est plus qu'une bibliothèque, c'est une suite de développement ayant pour ambition la productivité du développeur. Cette suite comprend un ensemble de bibliothèque (IHM, SQL, etc.) et un IDE (environnement de

développement intégré). La rapidité de développement que procure cette bibliothèque provient d'un usage « agressif » des possibilités qu'offre le C++, plutôt que de miser sur un générateur de code (comme QT le fait, par exemple).

TheIDE, l'IDE RAD de cette suite, utilise la technologie *BLITZ – build* pour réduire jusqu'à quatre fois le temps de compilation. Elle propose également : un outil de conception visuel d'interface ; *Topic*++, un outil de documentation de code et de documentation d'application ; *Assist*++, un analyseur de code C++ apportant un système de complétion automatique de code, de navigation dans le code, et une approche de transformation (*refactoring*) de code.

Si cet environnement vous tente, n'hésitez pas… Il a tout pour devenir incontournable (il ne lui manque que le support de Mac OS X en natif Cocoa).

Autres

- Borland C++ Builder (voir Compilateurs)
- Code::Block (**http:// www.codeblocks.org**)
- Eclipse (**http://www.eclipse.org**)
- XCode (fourni avec le système d'exploitation Mac OS X)

Bibliothèques

POCO C++ – Développement réseau et XML

Site : **http://pocoproject.org/poco/info/index.html**

Licence : *Boost Software* (licence libre pour le commercial et l'open source)

Systèmes : Windows, Mac OS X, Linux, HP UX, Tru64, Solaris, QNX, plus Windows CE et tout système embarqué compatible POSIX.

Compilateurs : Dec CC, Visual C++, GCC, HP C++, IBM xlC, Sun CC

POCO C++ (*C++ Portable Components*) est une collection de bibliothèques de classes pour le développement d'applications portables, orientées réseau. Ses classes s'intègrent parfaitement avec la bibliothèque standard STL et couvrent de multiples fonctionnalités : threads, synchronisation de threads, accès fichiers, flux, bibliothèques partagées et leur chargement, sockets et protocoles réseau (HTTP, FTP, SMTP, etc.), serveurs HTTP et parseurs XML avec interfaces SAX2 et DOM.

Blitz++ – Calcul scientifique en C++

Site : **http://www.oonumerics.org/blitz**

Licence : GPL ou Blitz artistic License

Systèmes : Linux, Sparc, SGI Irix, Cray, IBM AIX, Solaris, HP UX, Sun, Unix, Dec Alpha, Dec Ultrix.

Compilateurs : GCC, Visual C++ (2003+), Intel C++, CRI C++ (Cray), HP C++, KAI C++, etc.

Blitz++ est une bibliothèque C++ pour le calcul scientifique. Elle utilise les *templates* pour atteindre un niveau de performances proche du Fortran 77/90 (et parfois même meilleur).

ColDet – Détection de collision 3D

Site : **http://photoneffect.com/coldet**

Licence : LGPL

Systèmes : Linux, Windows, Mac OS X, etc.

Compilateurs : Visual C++, Borland C++ Builder, GCC, MetroWerks CodeWarrior, etc.

Cette bibliothèque apporte une solution libre au problème de détection de collision entre polyèdres génériques. Elle vise

la programmation des jeux 3D où l'exactitude de la détection entre deux objets complexes est requise. Cette bibliothèque fonctionne sur tout type de modèles, y compris des soupes de polygones. Elle utilise une hiérarchie de boîtes englobantes pour optimiser la détection, puis un test d'intersection sur les triangles pour l'exactitude. Elle fournit même, sur demande, le point exact de la collision et les paires de triangles s'intersectant. Un système de *timeout* peut être mis en place pour interrompre des calculs trop longs. Il est également possible de faire des tests d'intersection de type lancé de rayon-modèle, segment-modèle, et d'utiliser directement les primitives de test de collision lancé de rayon-sphère et sphère-sphère.

CGAL – Computational Geometry Algorithms Library

Site : **http://www.cgal.org**

Licence : LGPL/QPL (selon les parties utilisées) ou commerciale

Systèmes : Windows, Linux, Mac OS X, Solaris, SGI Irix

Compilateurs : Visual C++, Borland C++, GCC, Intel C++, SunPro, SGI CC, KAI C++

CGAL est une bibliothèque de structures et de calculs géométriques sûrs et efficaces. Parmi ceux-ci on trouve : les triangulations (2D contraintes ou de Delaunay 2D/3D), les diagrammes de Voronoï (points 23/3D, points massiques 2D, segments), les opérations booléennes sur les polyèdres, les arrangements de courbes et leurs applications (enveloppes 2D/3D, sommes de Minkowski), la génération de maillage (maillages de Delaunay 2D et 3D, peaux), le calcul de géométries (simplification de maillage de surface, subdivision et paramétrisation, estimation des propriétés différentielles locales, approximation de crêtes et d'ombiliques), alpha-formes, interpolations, collages, distances, structures de recherche, etc.

Dinkum Compleat Library – Standard C++, C99, and Embedded C++

Site : **http://www.dinkumware.com**

Licence : Commercial

Systèmes : Windows, Linux, Mac OS X, Solaris

Compilateurs : Visual C++, GCC, SunC++, Embedded Visual C++

Cette bibliothèque est une réimplémentation de la bibliothèque standard STL, en y ajoutant le support/émulation du C99 en plus de l'ISO 14882:1998/2003 et du TR1. Elle met l'accent sur la portabilité et les performances. Elle rassemble également d'autres fonctionnalités qu'il faut glaner dans d'autres bibliothèques. C'est une bonne solution pour ceux qui en ont les moyens, à condition qu'elle ne fasse pas double emploi avec une autre solution.

GC – Garbage Collector for C/C++

Site : **http://www.hpl.hp.com/personal/Hans_Boehm/ gc**

Systèmes : Linux, *BSD, Windows, Mac OS X, HP-UX, Solaris, Tru64, IRIX, etc.

Si vous êtes fatigué de gérer la mémoire et avez la possibilité de mettre en place un système de ramasse-miette, alors essayez cette bibliothèque. Elle est utilisée par le projet Mozilla (comme détecteur de perte de mémoire), le projet Mono, le compilateur statique Java GCJ, le runtime Objective C de GNU, et bien d'autres.

GMP – GNU Multiprecision Package

Site : **http://gmplib.org**

Licence : LGPL

Processeurs : AMD64, POWER64, POWER5, PPC970, Alpha, Itanium, x86, …

Compilateurs : Compilateur C/C++ standard

GMP, ou GNUmp, est une bibliothèque implémentant des nombres entiers signés, des nombres rationnels et des nombres à virgule flottante en précision arbitraire. Toutes les fonctions ont une interface normalisée. GMP est conçue pour être aussi rapide que possible en utilisant les mots machine comme type arithmétique de base, en utilisant des algorithmes rapides, en optimisant soigneusement le code assembleur pour les boucles intérieures les plus communes, et par une attention générale portée à la vitesse (par opposition à la simplicité ou à l'élégance).

LEDA – Library of Efficient Data types and Algorithms

Site : **http://www.algorithmic-solutions.com/leda**

Licence : Gratuite, professionnel, recherche (le contenu diffère selon la licence)

Systèmes : Windows, Linux, Solaris

Compilateurs : Visual C++ (2005+), GCC (3.4+), SunPRO (5.8)

LEDA est une immense bibliothèque de structures de données et d'algorithmes géométriques et combinatoires. Elle est utilisée par certains industrielles pour réaliser des bancs d'essais sur de grands jeux de données. Elle fournit une collection considérable de structures de données et d'algorithmes sous une forme qui leur permet d'être employés par des non-experts. Dans la version en cours, cette collection inclut la plupart des structures et algorithmes classiques du domaine. LEDA contient des implémentations efficaces pour chacun de ces types de données, par exemple, piles de Fibonacci pour des files d'attente prioritaires, tables

dynamiques d'adressage dispersé parfait (*dynamic perfect hashing*) pour les dictionnaires, etc. Un atout majeur de LEDA est son implémentation des graphes. Elle offre les itérations standard telles que « pour tous les nœuds v d'un graphe G » ou encore « pour tous les voisins W de v »; elle permet d'ajouter et d'effacer des sommets et des arêtes, d'en manipuler les matrices d'incidence, etc.

Pantheios – C++ Logging

Site : **http://www.pantheios.org**

Licence : type BSD

Systèmes : Windows, Linux, Mac OS X, Unix

Compilateurs : Borland (5.5.1+), Comeau (4.3.3+), Digital Mars (8.45+), GCC (3.2+), Intel (6+), Metrowerks (8+), Microsoft Visual C++ (5.0+)

Pantheios est une bibliothèque de journalisation (*logging*) offrant un bon équilibre entre contrôle des types, performances, généricité et extensibilité. La portabilité de cette bibliothèque est également un atout.

Elle offre un système de filtrage des messages en fonction des huit niveaux de sévérité définis par le protocole *SysLog* (RFC 3164, voir **http://fr.wikipedia.org/wiki/Syslog** et **http://tools.ietf.org/html/rfc3164**), sans pour autant vous limiter à ceux-ci (vous pouvez définir les vôtres). Elle fournit un grand nombre de plug-in d'écriture : fichier, stderr/stdout, SysLog (avec une implémentation personnalisée du protocole SysLog pour Windows), débogueur Windows, journal d'événement Windows, objet d'erreur COM, et vous pouvez en écrire d'autres.

La société Synesis Software (**http://synesis.com.au**) offre de personnaliser Pantheios selon vos besoins.

STLport – Bibliothèque standard alternative

Site : **http://www.stlport.org**

Licence : libre

Compilateurs :

- SUN Workshop C++ Compilers 4.0.x-5.0, 5.1, 5.2 (Forte 6, 6u1, 6u2), for Solaris
- GNU GCC 2.7.2-2.95, 3.0 Compilers, for all GCC platforms
- IBM xlC 3.1.4.0, for AIX
- IBM OS/390 C/C++ 1.x - 2.x Compiler, for OS/390 (STLport is the only available and official ANSI library for this platform)
- IBM VisualAge/ILE C++ for AS/400 Version 4 Release 4, for AS/400
- IBM Visual Age C++ 3.x-5.x, for OS/2 and Win32
- HP ANSI C++ Compiler, for HP-UX
- SGI MIPSpro C++ 7.x, for IRIX
- SGI MIPSpro C++ 4.0, for IRIX
- DEC C++ V5.5-004, V6.x for OSF 3.2/4.0, for Digital Unix.
- Tandem NCC, for Tandem OS
- SCO UDK C++, for UnixWare
- Apogee C++ 4.0, for SUN Solaris
- KAI C++ 3.1-4.x, for all KAI platforms
- Portland Group C++ compiler (pgCC), for all pgCC platforms
- Microsoft Visual C++ 4.x - 6.x, for Win32
- Microsoft Embedded Visual C++ 3.0
- Borland C++ 5.02-5.5, for Win16/32

- Metrowerks CodeWarrior Pro 1.8-5.3, for Mac OS, Win32
- Intel (R) C++ 4.0/4.5/5.0 Reference Compiler, for Win32
- Watcom C++ 10.x-11.0, for DOS, Win16, Win32
- Powersoft's Power++ 2.0, for Win32
- Symantec C++ 7x/8.x, for MacOS and Win32
- Apple MPW MrCpp and SCpp compilers, for Mac OS

STLport se distingue des STL fournit par la plupart de compilateurs, notamment en intégrant un mode de débogage à la STL à l'aide « d'itérateurs sûrs » et de préconditions permettant un contrôle rigoureux lors de l'exécution.

Ne cherchez pas plus loin si votre compilateur ne contient pas la bibliothèque standard STL.

Bibliothèques à dominante graphique

SDL – Simple DirectMedia Layer

Site : **http://www.libsdl.org**

Licence : LGPL (et aussi commerciale avec support pour ceux qui le souhaitent)

Systèmes : Linux, Windows, Windows CE, BeOS, Mac Os, Mac Os X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX et QNX. Le support d'autres systèmes (AmigaOS, Dreamcast, AIX, OSF/Tru64, RISC OS, SymbianOS et OS/2) semble exister mais non officiellement

Compilateurs : Tout compilateur C

Simple DirectMedia Layer est une bibliothèque multimédia multiplate-forme. Elle fournit un accès bas niveau au matériel audio, clavier, souris, joystick, 3D (à travers OpenGL) et tampon vidéo 2D. Elle est utilisée par des applications de lecture MPEG, des émulateurs, bon nombre de jeux populaires (dont le port Linux de « Civilization : Call To Power »).

SDL est écrite en C, mais fonctionne nativement en C++. Il existe aussi des ponts vers quantité de langages tels Ada, C#, D, Eiffel, Erlang, Euphoria, Guile, Haskell, Java, Lisp, Lua, ML, Objective C, Pascal, Perl, PHP, Pike, Pliant, Python, Ruby, Smalltalk et Tcl.

wxWidgets – Développement multiplate-forme et IHM

Site : **http://www.wxwidgets.org**

Licence : wxWidgets Library Licence (proche de la LPGL)

Systèmes : Mac OS X, GNU/Linux et Unix, Microsoft Windows, OS/2, ainsi que pour du matériel embarqué (embedded) sous GNU/Linux ou Windows CE

Compilateurs : Tout compilateur C++ standard

wxWidgets (anciennement wxWindows) est une bibliothèque graphique libre utilisée comme boîte à outils de programmation d'interfaces utilisateur multiplate-formes. À la différence d'autres boîtes à outils qui tentent de restituer une interface utilisateur identique sur toutes les plate-formes, wxWidgets restitue des abstractions comparables, mais avec l'apparence native de chaque environnement cible, ce qui est moins dépaysant pour les utilisateurs finaux.

La bibliothèque originale est écrite en C++ mais il existe de nombreux ponts vers les langages de programmation courants : Python – wxPython ; Perl – wxPerl ; BASIC – wxBasic ; Lua – wxLua ; OCaml – wxCaml ; JavaScript – wxJavaScript ; Java – wxJava ou wx4j ; Ruby – wxRuby ; Eiffel –

wxEiffel ; Haskell – wxHaskell ; C#/.NET – wx.NET ; Euphoria – wxEuphoria ; D – wxD. Certains sont plus développés que d'autres et les plus populaires restent wxPython, wxPerl et wxBasic.

Sous le nom « wx », wxWidgets est la base de l'interface utilisateur des applications développées avec C++BuilderX (qui n'est malheureusement plus disponible) de Borland.

QT – Développement multiplate-forme et IHM

Site : **http://trolltech.com/products**

Licence : QPL

Systèmes : Linux, Windows, Windows CE, Mac OS X, Unix

Compilateurs : Tout compilateur C++ standard

Qt est une bibliothèque logicielle orientée objet et développée en C++. Elle offre des composants d'interface graphique (*widgets*), d'accès aux données, de connexions réseau, de gestion des fils d'exécution, d'analyse XML, etc. Qt permet la portabilité des applications qui n'utilisent que ses composants par simple recompilation du code source. Les environnements supportés sont les Unix (dont Linux) utilisant le système graphique X Window System, Windows et Mac OS X.

Qt est notamment connue pour être la bibliothèque sur laquelle repose l'environnement graphique KDE, l'un des environnements de bureau les plus utilisés dans le monde Linux.

De plus en plus de développeurs utilisent Qt, y compris parmi de grandes entreprises. On peut notamment citer Google, Adobe Systems ou encore la NASA.

ILOG VIEWS – Bibliothèque C++ et éditeur pour concevoir de très grosses IHM

Site : **http://www.ilog.com/products/views**

Licence : Commerciale

Systèmes : Windows, Unix (SUN-Solaris, HP-UX, IMB-AIX), Linux

ILOG VIEWS est une bibliothèque C++ haut niveau. Elle inclut un système de ressource portable, un gestionnaire d'événements, le support PostScript, une bibliothèque de composants graphiques très complète (dont un gérant les diagrammes de Gantt), des outils de conception d'IHM interactifs très puissants (avec gestion d'état), des composants gérant la connexion et l'utilisation de bases de données (Oracle, IBM DB2, etc.)

La réputation de cette bibliothèque n'est plus à faire.

Utilitaires

Understand for C++ – Reverse engineering, documentation and metrics tool

Site : **http://www.scitools.com/products/understand**

Licence : Commerciale

Systèmes : Windows, Linux, Solaris, HP-UX, IRIX, plug-in pour Eclipse

Langages : C, C++. Il existe aussi des versions pour C# et Java

Understand for C++ est un outil d'analyse, de documentation et de métrique (mesure) de code source. Il permet de naviguer dans le code grâce à un système de références croisées, d'éditer le code (affiché avec mise en forme), et de le visualiser sous diverses vues graphiques.

Cet environnement inclut des API PERL et C permettant d'inclure votre propre système de documentation de votre code source.

Cet outil, difficile d'utilisation au premier abord, justifie l'effort d'apprentissage par ses qualités.

Ch C/C++ interpreter – quand le C/C++ devient script...

Site : **http://www.softintegration.com**

Licence : Commerciale, une version gratuite existe (mais elle n'est pas embarquable)

Systèmes : Windows, Linux, Mac OS X, Solaris, HP-UX, FreeBSD et QNX

Compilateurs : GCC, HP C++, Sun CC,Visual C++

Ch est l'interpréteur C/C++ le plus complet. Il peut être embarqué dans vos applications. Il supporte la norme ISO C (C90), la plupart des nouvelles fonctionnalités apportées par le C99, les classes en C++, POSIX, X/Motif, Windows, OpenGL, ODBC, GTK+, C LAPACK, CGI, XML, le dessin graphique 2D/3D, le calcul numérique avancé et la programmation en shell. De plus, Ch possède quelques autres fonctionnalités que l'on retrouve dans d'autres langages et logiciels.

Ch Standard Edition est gratuit, même pour des applications commerciales. Dommage que la version embarquable (Embedded Ch) ne le soit pas, même pour les projets non commerciaux…

Annexe B

Les ajouts de la future norme C++ (C++0x)

C++0x, la future norme du langage C++, remplacera l'actuelle norme C++ (ISO/IEC 14882), plus connue sous les acronymes C++98 et C++03, publiée en 1998 et mise à jour en 2003. Celle-ci inclura non seulement plusieurs ajouts au langage lui-même, mais étendra également la bibliothèque standard C++ en lui incorporant une bonne partie des rapports techniques C++ numéro 1 (les fameux TR1). Cette norme n'est pas encore publiée, mais cette annexe en présente quelques extraits issus du rapport N2597 datant de mai 2008. Vous pourrez ainsi vous familiariser avec quelques-unes des avancées majeures du C++.

g++ a déjà commencé à implémenter certaines parties de cette future norme. Vous pouvez vous référer à l'adresse web **http://gcc.gnu.org/projects/cxx0x.html** pour en suivre la progression. Les autres acteurs du secteur des compilateurs (Intel, Borland, etc.) attendent certainement que cette norme soit figée pour finir (ou commencer) leur propre implémentation. Il faudra donc attendre encore avant de pouvoir en utiliser tout le potentiel…

Variable locale à un thread

```
thread_local
```

En environnement multithread, chaque thread possède souvent ses propres variables. C'est le cas par exemple des variables locales d'une fonction, mais pas des variables globales ou statiques.

Un nouveau type de stockage, en plus des nombreux existants (`extern`, `static`, `register` et `mutable`) a été proposé pour le prochain standard : le stockage local au thread. Le nouveau mot-clé correspondant est `thread_local`.

Un objet thread est analogue à un objet global. Les objets globaux existent durant toute l'exécution d'un programme tandis que les objets thread existent durant la vie du thread. À la fin du thread, les objets thread sont inaccessibles. Comme pour les objets statiques, un objet thread peut être initialisé par un constructeur et détruit avec un destructeur.

Unicode et chaînes littérales

```
char s[] = "Chaine ASCII standard";
wchar_t s[] = L"Chaine wide-char ";
```

```
char s[] = u8"chaine UTF-8";
char16_t s[] = u"Chaine UTF-16";
char32_t s[] = U"Chaine UTF-32";
```

Le C++ actuel offre deux types de chaînes littérales. La première, composée entre guillemets, permet de coder des chaînes de caractères ASCII à zéro terminal. La deuxième (`L""`) permet de le faire avec un jeu de caractères large (`wchar_t`). Aucun de ces deux types ne supporte les chaînes Unicode.

L'internationalisation des applications devenant la norme aujourd'hui, le C++0x permettra au langage de supporter Unicode, à l'aide de trois encodages : UTF-8, UTF-16 et UTF-32. De plus, deux autres types de caractères feront leur apparition : `char16_t` et `char32_t`. Vous l'aurez compris, ils permettront le support de UTF-16 et UTF-32. UTF-8 sera traité directement avec le type `char`.

L'écriture de chaînes littérales utilise souvent les séquences d'échappement pour préciser certains caractères, comme par exemple `"\x3F"`. Les chaînes Unicode possèdent aussi leur séquence d'échappement. La syntaxe est illustrée dans l'exemple suivant :

```
u8"caractère Unicode : \u2018.";
u"caractère Unicode : \u2018.";
U"caractère Unicode : \u2018.";
```

Le nombre après `\u` est donné sous forme hexadécimale. Il ne nécessite pas la présence du préfixe `0x`. Si vous utilisez `\u`, il est sous-entendu qu'il s'agit d'un caractère Unicode 16 bits. Pour spécifier un caractère Unicode 32 bits, utilisez `\U` et une valeur hexadécimale 32 bits. Seules des valeurs Unicode valides sont autorisées. Par exemple, des valeurs entre U+D800 et U+DFFF sont interdites car elles sont réservées aux paires de substitution en encodage UTF-16.

```
R"[Une chaine \ bas niveau avec ce qu'on veut " ]";
R"delimiteur[Encore \ ce qu'on veut " ]delimiteur";
```

Il est parfois utile de désactiver les séquences d'échappement. C++0x fournit un moyen de coder des chaînes « bas niveau ». Dans la première ligne de l'exemple précédent, tout ce qui est entre les crochets (`[` et `]`) fait partie de la chaîne. Dans la deuxième ligne, `"delimiteur[` marque le début de la chaîne et `]delimiteur"` en marque la fin.

Le `delimiteur` est arbitraire et peut être défini à votre convenance ; il doit simplement être le même au début et à la fin.

Comme le montre l'exemple suivant, les chaînes bas niveau sont également disponibles en Unicode :

```
u8R"XXX[une chaine UTF-8 "bas niveau".]XXX";
uR"*@[une chaine UTF-16 "bas niveau".]*@";
UR"[une chaine UTF-32 "bas niveau".]";
```

Délégation de construction

```
class UnType
{
    int nombre ;
public :
    UnType(int unNombre) : nombre(unNombre) { }
    UnType() : UnType(42) { }
};
```

Les classes sont un élément essentiel du C++. Les changements apportés par la norme C++0x faciliteront et sécuriseront la mise en œuvre d'une hiérarchie de classes.

En C++ standard, un constructeur ne peut pas en appeler un autre : chaque constructeur doit construire tous les membres de la classe lui-même, ce qui implique souvent de dupliquer le code d'initialisation. Grâce au C++0x, un constructeur pourra en appeler d'autres. D'autres langages, comme Java, utilisent déjà ce mécanisme (connu sous le nom de « délégation ») pour réutiliser le comportement d'un constructeur existant.

Attention

C++03 autorisera la syntaxe suivante pour l'initialisation des membres :

```
class MaClasse
{
public:
    MaClasse() {}
    explicit MaClasse(int i) : m_i(i) {}
private:
    int m_i = 10;
};
```

Tout constructeur initialisera `m_i` avec la valeur `10`, sauf si ce constructeur surcharge l'initialisation de `m_i` avec sa propre valeur. Ainsi, le constructeur vide de l'exemple précédent utilise l'initialisation de la définition, mais l'autre constructeur initialise la variable avec la valeur donnée en paramètre.

Héritage des constructeurs

```
class B : A
{
    using A::A;
    // ...
};
```

On aimerait souvent pouvoir initialiser une classe dérivée avec le même ensemble de constructeurs que la classe de base. Jusqu'à présent, la seule manière de faire cela consiste à redéclarer tous les constructeurs en les redirigeant sur ceux de la classe de base. Si le compilateur pouvait faire le travail, il en résulterait moins d'erreur et l'intention serait rendue bien visible. La nouvelle norme C++ va étendre la signification du mot-clé `using` en ce sens. Il acquiert ainsi une nouvelle signification : permettre d'hériter tous les

constructeurs d'une classe mère donnée, évitant ainsi d'avoir à tous les coder de nouveau.

```
struct B1
{
    B1( int );
};

struct B2
{
    B2( int = 13, int = 42 );
};

struct D1 : B1
{
    using B1::B1;
};

struct D2 : B2
{
    using B2::B2;
};
```

Dans l'exemple précédent, les constructeurs de `B1` candidats à l'héritage pour `D1` sont :

- `B1( const B1 & )` ;
- `B1( int )`.

L'ensemble des constructeurs de `D1` est :

- `D1()` constructeur par défaut implicite, mal formé si utilisé ;
- `D1( const D1 & )` constructeur par copie implicite, non hérité ;
- `D1( int )` constructeur hérité implicitement.

Les constructeurs de `B2` candidats à l'héritage pour `D2` sont :

- `B2( const B2 & )` ;
- `B2( int = 13, int = 42 )` ;
- `B2( int = 13 )` ;
- `B2()`.

L'ensemble des constructeurs de `D2` est :

- `D2()` constructeur par défaut implicite, non hérité ;
- `D2( const D2 & )` constructeur par copie implicite, non hérité ;
- `D2( int, int )` constructeur hérité implicitement ;
- `D2( int )` constructeur hérité implicitement.

Info

Si deux déclarations `using` héritent de constructeurs ayant la même signature, le programme est mal formé ; il est alors nécessaire de le surcharger dans la classe fille.

```
struct B1
{
    B1( int );
};

struct B2
{
    B2( int );
};

struct D1 : B1, B2
{
    using B1::B1;
    using B2::B2;  // mal formé: double déclaration
                   // implicite d'un même constructeur
};

struct D2 : B1, B2
{
    using B1::B1;
    using B2::B2;
    D2( int );    // ok : déclaration utilisateur
                  // prévient le conflit
};
```

Constructeur et destructeur par défaut

```
struct Type
{
    Type() = default;
};
```

```
struct Type
{
    Type() = delete;
};
```

En C++ standard, le compilateur peut fournir, pour les objets qui ne les fournissent pas explicitement, un constructeur par défaut, un constructeur par copie, un opérateur de copie (`operator=`) et un destructeur. Le C++ définit également des opérateurs globaux (comme `operator=` et `operator new`) qui peuvent également être surchargés.

Le problème est qu'il n'y a aucun moyen de contrôler la création de ces versions par défaut. Par exemple, pour rendre une classe non copiable, il faut déclarer le constructeur par défaut et l'opérateur de copie en tant que membres privés et ne pas les implémenter. De cette façon, l'utilisation de ces fonctions provoquera soit une erreur de compilation, soit une erreur d'édition de liens. Cette technique, si elle fonctionne, est toutefois loin d'être idéale.

C++0x permettra de forcer, ou empêcher, la génération de fonctions par défaut. Le code suivant déclare explicitement le constructeur par défaut d'un objet :

```
struct Type
{
    Type() = default; // déclaration explicite
    Type(AutreType valeur);
};
```

Inversement, il est possible de désactiver la génération automatique de ces fonctions. Le code suivant montre comment créer un type non copiable :

```
struct Type // non copiable
{
    Type() = default;
    Type(const Type&) = delete;
    Type& operator=(const Type&) = delete;
};
```

Le code suivant montre comment empêcher un type d'être alloué dynamiquement. La seule façon d'allouer un tel objet est de créer une variable temporaire (allocation sur la pile). Il est impossible d'écrire `Type *ptr = new Type ;` (allocation sur le tas).

```
struct Type // allocation dynamique interdite
{
    void* operator new(std::size_t) = delete;
};
```

La spécification `= delete` peut être utilisée sur n'importe quelle fonction membre pour empêcher son utilisation. Cela peut être utile, par exemple, pour empêcher l'appel d'une fonction avec certains paramètres. L'exemple suivant montre comment empêcher une conversion implicite de `double` vers `int` lors de l'appel de la fonction membre `f()` avec un nombre réel. Un appel tel `obj->f(1.0)` provoquera une erreur de compilation.

```
struct Type // pas de double
{
    void f(int i);
    void f(double) = delete;
};
```

La technique employée dans l'exemple précédent peut être généralisée pour forcer le programmeur à n'utiliser que le type explicitement défini par la déclaration de la fonction membre. Le code suivant n'autorise des appels à la fonction membre `f()` qu'avec le type `int` :

```
struct Type // que des int
{
    void f(int i);
    template<T> void f(T) = delete;
};
```

union étendue

```
struct Point
{
    Point() {}
    Point(int x, int y) : m_x(x), m_y(y) {}
    int m_x, m_y;
};
union
{
    int i;
    double d;
    Point p; // autorisé avec C++0x
};
```

En C++, le type des membres d'une `union` est soumis à certaines restrictions. Par exemple, les unions ne peuvent pas contenir des objets définissant des constructeurs non triviaux. Plusieurs des limitations imposées sur les unions semblant être superflues, le prochain standard enlèvera toutes les restrictions sur le type des membres des unions à l'exception des références sur les types. Cela rendra les unions plus faciles à utiliser.

Opérateurs de conversion explicites

```
struct Type
{
    explicit operator bool() { ... }
};
```

La nouvelle norme permettra d'appliquer le mot-clé `explicit` à tout opérateur de conversion, en plus des constructeurs, interdisant au compilateur de convertir automatiquement un type en un autre. Le cas ci-dessus empêchera par exemple la conversion implicite en booléen.

Type énumération fort

```
enum class Enumeration
{
    Val1,
    Val2,
    Val3 = 10,
    Val4 // 11
};
```

Les énumérations sont par nature des *non-types* : ils ont l'apparence d'un type (et presque la signification) mais n'en sont pas réellement. Ce sont en effet de simples entiers. Cela permet de comparer deux valeurs provenant d'énumérations distinctes. La dernière norme (datant de 2003) a apporté un peu de sécurité en empêchant la conversion d'un entier en énumération ou d'une énumération vers une autre. Autre limitation : le type d'entier sous-jacent ne peut être configuré ; il est dépendant de l'implémentation du compilateur et de la plate-forme.

La norme C++0x permettra, en ajoutant le mot-clé `class` après `enum`, de créer de vrais types énumération à part entière. Ainsi avec l'exemple générique ci-avant, une comparaison du type `Val4 == 11` générera une erreur de compilation.

Par défaut, le type sous-jacent d'une classe d'énumération est toujours le type `int`. Il pourra néanmoins être personnalisé. Dans l'exemple ci-après, vous voyez comment le changer en `unsigned int` ou `long`. De plus, il sera nécessaire de préciser le nom de l'énumération : `Enum1::Val1`.

```
enum class Enum1 : unsigned int { Val1, Val2 };
enum class Enum2 : long { Val1, Val2 };
```

Listes d'initialisation

```
class Sequence
{
public:
    Sequence(std::initializer_list<int> list);
};
Sequence s = { 1, 2, 3, 5, 7, 0 };
```

```
void Fonction(std::initializer_list<float> list);
Fonction({1.0f, -3.45f, -0.4f});
```

Le concept de liste d'initialisation existe déjà en C. L'idée est de fournir une liste d'arguments pour initialiser un tableau ou une structure (dans ce dernier cas, les éléments doivent être donnés dans le même ordre que ceux de la structure). Le système est récursif et permet d'initialiser des structures de structures. C++0x étend ce concept aux classes. Le concept de liste d'initialisation se nomme `std::`

`initializer_list<>`. Il pourra être utilisé non seulement avec les constructeurs mais aussi avec les fonctions.

Une liste d'initialisation est constante. Ses valeurs ne peuvent pas être changées ou modifiées par la suite.

Les conteneurs standard pourront être initialisés grâce à cette nouvelle fonctionnalité.

```
std::vector<std::string> v = { "abc", "defgh", "azerty" };
```

Initialisation uniformisée

```
struct Structure
{
    int   x;
    float y;
};

struct Classe
{
    Classe(int x, float y) : m_x(x), m_y(y) {}
private:
    int   m_x;
    float m_y;
};

Structure var1{5, 3.2f};
Classe    var2{2, 4.3f};
```

C++0x met en place une uniformisation de l'initialisation des types. L'initialisation de `var1` fonctionne exactement comme une liste d'initialisation de style C (mais le = n'est plus nécessaire). Une conversion implicite est automatiquement utilisée si nécessaire (et si possible), sinon il en résulte une erreur de compilation. L'initialisation de `var2` appelle simplement le destructeur.

Cette uniformisation va permettre d'omettre le type dans certains cas. Le code suivant vous montre en quel sens :

```
struct StrindAndID
{
    std::string text;
    int id;
};
StringAndID makeDefault()
{
    return {"UnTexte", 12}; // Notez l'omission du type
}
```

Cette uniformisation ne remplace pas la syntaxe habituelle des constructeurs. En effet, il peut être indispensable d'y avoir recours. Imaginez qu'une classe implémente un constructeur par liste d'initialisation (`Classe(std::initializer_list<T>);`). Ce constructeur aura priorité sur tous les autres avec la syntaxe unifiée. On rencontre ce cas de figure avec la nouvelle implémentation de la STL, comme par exemple `std::vector<>`. Cela implique que l'exemple suivant ne crée pas un tableau de quatre éléments mais un tableau contenant `4` pour seul élément :

```
std::vector<int> V{4}; // V contient la valeur 4
```

Pour créer un vecteur de quatre éléments, il faut utiliser la syntaxe de constructeur standard, comme ceci :

```
std::vector<int> V(4); // V contient 4 éléments " vides "
```

Type automatique

```
auto variable = ...;
decltype(variable)
```

Jusqu'à présent, pour déclarer une variable en C++, il était obligatoire de spécifier son type. Cependant, avec l'arrivée des templates et des techniques de métaprogrammation associées, le type de quelque chose n'est pas toujours aisé à définir. C'est particulièrement le cas des valeurs de retour des fonctions templates. De même, il est parfois pénible de déterminer le type de certaines variables que l'on souhaite sauvegarder, obligeant à fouiller parfois loin dans les définitions de classes.

Grâce au mot-clé `auto`, C++0x permettra la déclaration d'une variable sans en connaître le type, à condition toutefois d'initialiser celle-ci explicitement. Le type est ainsi spécifié implicitement par la valeur d'initialisation. Le code suivant illustre cette technique avec l'appel d'une fonction (le compilateur connaît forcément son type de retour) et avec une constante littérale (en l'occurrence un `int`).

```
auto varDeTypeObscure = std::bind(&uneFonction, _2,
➥_1, unObjet);
auto varEntiere = 5;
```

C++0x offrira encore un mot-clé supplémentaire, `decltype(...)`, permettant d'obtenir le type d'une variable, et surtout de pouvoir l'utiliser comme un type traditionnel.

```
int unEntier;
decltype(unEntier) unAutreEntier = 33;
```

Astuce

Le mot-clé `auto` permettra aussi d'obtenir un code plus aisé à lire. Par exemple, le code suivant :

```
for (std::vector<int>::const_iterator it = V.begin();
it != V.end(); ++it)
{
     // ...
}
```

pourra s'écrire :

```
for (auto it = V.begin(); it != V.end(); ++it)
{
    // ...
}
```

Boucle `for` ensembliste

```
int tableau[5] = {1, 2, 3, 4, 5};
for(int &x : tableau)
{
  x *= 2;
}
```

La bibliothèque BOOST définie la notion d'intervalle (*range*). Les conteneurs ordonnés sont un surensemble de la notion d'intervalle, et deux itérateurs sur un tel conteneur peuvent définir un intervalle. Ces concepts, et les algorithmes qui les utilisent, seront inclus dans la bibliothèque C++0x standard. Cependant, l'intérêt du concept d'intervalle est tel que le C++0x en fournira une implémentation au niveau du langage lui-même.

Dans l'exemple précédent, la variable `x` a une portée égale à celle de la boucle `for`. La partie après de `:` est un intervalle

sur lequel s'effectue l'itération. Les tableaux de type C sont automatiquement convertis en intervalle. Tout objet respectant le concept d'intervalle peut y être utilisé. Les `std::vector<>` de la STL du C++0x en sont un exemple.

Syntaxe de fonction unifiée

```
[] Fonction(paramètres) -> TypeRetour;
```

La syntaxe à laquelle nous sommes habitués jusqu'à présent, issue du C, commençait à faire sentir ses limites. Par exemple, il est actuellement impossible d'écrire le code suivant :

```
template <typename A, typename B>
T plus(const A& a, const B& b)
{
    return a+b;
}
```

On pourrait être tenté d'utiliser la nouvelle fonctionnalité de type automatique que le C++0x fournira, en remplaçant `T` par `decltype(A+B)`. Mais cela ne suffit pas, car il faudrait que les types `A` et `B` soient définis, et cela n'est vrai qu'après l'interprétation du prototype de la fonction. D'où l'ajout à la norme C++0x d'une nouvelle syntaxe de fonction unifiée. Le code suivant montre comment écrire une telle fonction en C++0x :

```
template <typename A, typename B>
[] plus(const A& a, const B& b) -> decltype(a+b)
{
    return a+b;
}
```

Pour les fonctions membres, cette syntaxe s'utilise ainsi :

```
struct Structure
{
    []FonctionMembre(int x) -> int;
};
[] Structure::FonctionMembre(int x) -> int
{
    return x + 2;
}
```

Concepts

```
concept NomDuConcept<typename T> { ... }
auto concept NomDuConcept<typename T> { ... }
```

```
template <typename T> requires NomDUnConcept<T> ...
concept_map NomDuConcept<Type> { ... }
template<typename T> concept_map NomDuConcept<T> { ... }
```

L'écriture de classes et de fonctions templates nécessite presque obligatoirement de présupposer certaines conditions sur les paramètres templates. Par exemple, la STL actuelle suppose que les types utilisés avec ses conteneurs soient assignables. Contrairement au contrôle de type inhérent au polymorphisme, il est possible de passer n'importe quel type comme paramètre template, à condition bien sûr qu'il supporte toutes les opérations employées dans l'implémentation de ce template. Du coup, les prérequis sur le type du paramètre sont implicites au lieu d'être explicites. Les concepts vont permettre de codifier l'interface qu'un paramètre template nécessite.

La première motivation à la création des concepts est l'amélioration des messages d'erreur du compilateur. En effet, jusqu'à présent, une erreur liée à un manque de fonctionnalité d'un paramètre template provoquait un message d'erreur souvent très long (le type template est souvent très long). De plus, ce message survient bien plus loin dans le code qu'au moment de l'instanciation de la classe ou de la fonction template (au moment de l'utilisation de la fonctionnalité manquante), rendant difficile la localisation et la compréhension de l'erreur.

```
template <LessThanComparable T>
const T& min(const T& x, const T& y)
{
    return y < x ? : y : x;
}
```

La fonction template suivante, en employant `LessThanComparable` plutôt que `class` ou `typename`, exprime comme prérequis que le type `T` doit respecter le concept `LessThanComparable` précédemment défini. Grâce à cela, si vous utilisez cette fonction template avec un type qui ne respecte pas ce concept, vous obtiendrez un message d'erreur clair indiquant que le type utilisé lors de l'instanciation du template ne respecte pas le concept `LessThanComparable`. Le code suivant est une forme d'écriture plus générique exprimant la même chose, mais permettant aussi de spécifier plusieurs concepts au lieu d'un :

```
template <typename T> requires LessThanComparable<T>
const T& min(const T& x, const T& y)
{
    return y < x ? : y : x;
}
```

Chose intéressante, il sera possible d'empêcher l'utilisation d'un modèle template avec un type conforme à un concept donné. Par exemple, `requires ! LessThanComparable<T>` :

```
auto concept LessThanComparable<typename T>
{
  bool operator<(T, T);
}
```

Le mot-clé `auto` signifie ici que tout type qui supporte les opérations mentionnées dans le concept est considéré comme supportant le concept. Sans ce mot-clé, le type doit utiliser une carte de concept pour signifier lui-même qu'il supporte ce concept. Le concept précédent indique que tout type `T` tel qu'il existe un opérateur `<` prenant deux objets de type `T` et retournant un `bool` sera considéré comme `LessThanComparable`. Cet opérateur ne doit pas nécessairement être un opérateur global mais peut aussi être un opérateur membre du type `T`.

```
auto concept Convertible<typename T, typename U>
{
  operator U(const T&);
}
```

Les concepts peuvent impliquer plusieurs types différents. Par exemple, le concept précédent exprime qu'un type peut être converti en un autre. Pour utiliser un tel concept, il est obligatoire d'utiliser la syntaxe généralisée :

```
template<typename U, typename T> requires Convertible<T, U>
U convert(const T& t)
{
    return t;
}
```

Les concepts peuvent aussi être composés. La définition du concept suivant indique qu'il doit aussi satisfaire un autre concept, `Regular`, précédemment défini :

```
concept InputIterator<typename Iter, typename Value>
{
    requires Regular<Iter>;
    Value operator*(const Iter&);
    Iter& operator++(Iter&);
    Iter operator++(Iter&, int);
}
```

Les concepts peuvent également être dérivés d'un autre concept, comme avec l'héritage. Voici un exemple de syntaxe :

```
concept ForwardIterator<typename Iter, typename Value> :
    InputIterator<Iter, Value>
{
    // Autres contraintes
}
```

Des `typename`s peuvent aussi être associés à un concept. Cela impose au template respectant ce concept de définir ces types. Le code suivant illustre cela avec le concept d'itérateur :

```
concept InputIterator<typename Iter>
{
    typename value_type;
    typename reference;
    typename pointer;
    typename difference_type;
    requires Regular<Iter>;
    requires Convertible<reference, value_type>;
    reference operator*(const Iter&); // déréférencement
    Iter& operator++(Iter&);          // pré-incrément
    Iter operator++(Iter&, int);      // post-incrément
    // ...
}
```

Les cartes de concept, que nous avons évoquées précédemment, permettent d'associer explicitement un type à un concept. L'exemple suivant permet de spécifier tous les types du concept d'`InputIterator` pour le type `char*` :

```
concept_map InputIterator<char*>
{
    typedef char value_type ;
    typedef char& reference ;
    typedef char* pointer ;
    typedef std::ptrdiff_t difference_type ;
};
```

Les cartes de concept peuvent elles-mêmes bénéficier de la notion de template. L'exemple suivant généralise la carte de concept précédente à tous les pointeurs :

```
template <typename T>
concept_map InputIterator<T*>
{
    typedef T value_type ;
    typedef T& reference ;
    typedef T* pointer ;
    typedef std::ptrdiff_t difference_type ;
};
```

Autorisation de `sizeof` sur des membres

```
struct UnType { AutreType membre; };
sizeof(UnType::membre);
```

Avec la norme actuelle, cet exemple produit une erreur de compilation. Avec la norme C++0x, vous obtiendrez la taille de `AutreType`.

Amélioration de la syntaxe pour l'utilisation des templates

```
Type<Autre<int>> obj;
```

Jusqu'à aujourd'hui, les compilateurs confondent >> avec l'opérateur de redirection. Il est obligatoire de séparer les deux > par un (ou plusieurs) espace. Cette limitation sera levée avec l'apparition du support de C++0x.

Template externe

```
extern template class ...;
```

En C++, le compilateur doit instancier un template chaque fois que celui-ci est rencontré. Cela peut considérablement augmenter le temps de compilation, en particulier si le template est instancié dans de nombreux fichiers avec les mêmes paramètres. Il n'existe aucun moyen de dire de ne pas provoquer l'instanciation d'un template.

Le C++ dispose déjà d'un moyen de forcer l'instanciation d'un template. Le code suivant en montre un exemple :

```
template class std::vector<MaClasse>;
```

C++0x introduit l'idée de template externe, permettant ainsi de prévenir son instanciation. Le code suivant montre comment, en indiquant au compilateur de ne pas instancier le template dans cette unité de traduction :

```
extern template class std::vector<MaClasse>;
```

Expressions constantes généralisées

constexpr

Le C++ dispose déjà du concept d'expression constante. C'est le cas par exemple de `3 + 4` qui correspond toujours au même résultat. Les expressions constantes sont des opportunités d'optimisation pour le compilateur. Il exécute l'expression à la compilation et stocke le résultat dans le programme. D'un autre côté, il y existe un certain nombre d'endroits où les spécifications du langage C++ requièrent des expressions constantes : définir un tableau nécessite une constante, et une valeur d'énumération doit être une constante.

Pourtant, alors qu'une expression constante est toujours le résultat d'un appel de fonction ou d'un constructeur, il est impossible d'écrire (actuellement) le code ci-après. Ce n'est pas possible car `cinq() + 5` n'est pas une expression constante. Le compilateur n'a aucun moyen de savoir que `cinq()` est une constante.

```
int cinq() { return 5; }
int tableau[cinq() + 5]; // illégal
```

C++0x introduit le mot-clé `constexpr`, qui permet au programmeur de signifier qu'une fonction ou un objet est une constante à la compilation. L'exemple précédant peut-être réécrit comme ci-après. Cela permet au compilateur de comprendre, et vérifier, que `cinq()` est une constante.

```
constexpr int cinq() { return 5; }
int tableau[cinq() + 5]; // ok : crée un tableau de
➥ 10 entiers
```

L'utilisation de `constexpr` sur une fonction impose des limitations très strictes sur ce que la fonction peut faire.

Premièrement, la fonction ne peut pas retourner `void`. Deuxièmement, le contenu de la fonction doit être de la forme `return` *`expression`*. Troisièmement, *`expression`* doit être une expression constante, après substitution des paramètres. Cette expression constante ne peut qu'appeler d'autres fonctions définies comme `constpexpr`, ou utiliser d'autres variables ou données constantes elles aussi. Quatrièmement, toute forme de récursion est interdite. Enfin, une fonction ainsi déclarée ne peut être appelée que dans sa portée.

Les variables peuvent aussi être des expressions constantes. Les variables ainsi déclarées sont implicitement `const`. Elles peuvent uniquement stocker des résultats d'expressions ou de constructeurs constants.

```
constexpr double graviteTerrestre = 9.8;
constexpr double graviteLunaire   = graviteTerrestre / 6
```

Afin de pouvoir créer des objets constants, un constructeur peut être déclaré `constexpr`. Dans ce cas, son corps doit être vide et ses membres doivent être initialisés avec des expressions constantes. Le destructeur d'un tel objet doit également être trivial.

Tout membre d'une classe, comme le constructeur par copie, la surcharge d'opérateur, etc., peut être déclaré comme `constexpr`, à condition qu'il vérifie les contraintes d'une fonction constante. Cela permet au compilateur de copier des classes ou de faire des opérations sur celles-ci pendant le processus de compilation.

Bien entendu, les fonctions et constructeurs constants peuvent être appelés avec des variables non constantes. Dans ce cas, aucune optimisation ne sera faite à la compilation : le code sera appelé lors de l'exécution. Du coup, le résultat ne peut évidemment pas être stocké dans une variable constante.

Les variadic templates

```
template <class ... Types> ...;
```

Les *variadic templates* sont le pendant pour les templates des fonctions variadiques, dont la plus connue est `printf()`. Les templates pourront ainsi recevoir un nombre indéterminé de paramètres, au sens de non fixé à l'avance.

La première notion à connaître pour utiliser cette nouvelle fonctionnalité est le *template parameter pack*. C'est le paramètre template qui représente zéro ou plusieurs arguments. Dans la définition générique précédente, il correspond à `class ... Types`.

La deuxième notion est le *function parameter pack*. C'est l'équivalent de la première notion pour ce qui est des paramètres d'une fonction template. Dans `template<class ... Types> void f(Types ... args);` le *function parameter pack* est `Types... args`.

L'expression *parameter pack* désigne indifféremment l'une ou l'autre des deux notions précédentes.

Enfin, la troisième et dernière notion est le *pack expansion*. C'est une expression qui permet de transmettre à une fonction la liste des arguments du pack. Dans le code suivant, `&rest...` est un *pack expansion* :

```
template <ckass ... Types> void f(Types ... rest);
template <ckass ... Types> void f(Types ... rest)
{
    f(&rest ...);
}
```

Le code suivant est utilisé dans la nouvelle définition de tuple (n-uplet). Elle permet de récupérer le dernier argument

d'un pack et illustre bien l'utilisation de cette nouvelle fonctionnalité :

```
// récupération du type du dernier argument
template <class... Args>
struct last;

template <class T>
struct last<T> { typedef T type; }

template <class T, class... Args>
struct last<T, Args...> : last<Args...> {};

// récupération du dernier argument
template <class T> const T& last_arg(const T& t) {
➥ return t; }
template <class T, class... Args>
typename last<Args...>::type const &
    last_arg(const T&, const Args&... args)
    {
        return last_arg(args...);
    }
```

Pointeur nul

`nullptr`

Depuis près d'une trentaine d'années, l'écriture du pointeur nul est sujette à polémique : `NULL`, `0` ou `(void*)0` ? Le C++0x va enfin clore le chapitre ; `nullptr` va devenir un nouveau mot-clé désignant le pointeur nul.Voici quelques exemples illustrant son utilisation et son intérêt.

```
char* p = nullptr; // ok
char c = nullptr; // erreur de type
int i = nullptr; // erreur de type
```

Pointeurs et types numériques

```
int (A::*pmf)(char*) = nulltpr; // ok
int A::*pmi = nullptr; // ok, pointeur sur une donnée
                          ➥ membre
```

Pointeurs et membres

```
if (nullptr == 0) // erreur de type
if (nullptr) // erreur, pas de conversion implicite
               ➥ vers bool
if (p == nullptr) // ok
```

Pointeurs et comparaison

```
void f(int);
void f(char*);
f(0); // appelle f(int)
f(nullptr); // appelle f(char*)
```

Pointeurs et surcharge

Tuple

```
template <class... Types> class tuple;
```

Un tuple est une collection de dimension fixe d'objets de types différents. Tout type d'objets peut être élément d'un tuple. Cette nouvelle fonctionnalité est implémentée dans

un nouveau fichier en-tête et bénéficie des extensions du C++0x telles que : paramètres templates infinis (*variadic templates*), référence sur référence et arguments par défaut pour les fonctions templates.

```
typedef tuple<int, double, long&, const char*> tuple_test;
long longueur = 43;
tuple_test essai( 22, 3.14, longueur, "Bonjour");
longueur = get<0>(essai);   // longueur vaudra 22
get<3>(essai) = "Au revoir"; // modifie la 4e valeur du tuple
```

Il est possible de créer le tuple `essai` sans définir son contenu si les éléments du tuple possèdent tous un constructeur par défaut. De même, il est possible d'assigner un tuple à un autre si les deux tuples sont de même type (dans ce cas, chaque élément doit posséder un constructeur par copie) ou si chaque élément de droite est convertible avec le type de l'élément de gauche (grâce à un constructeur approprié).

```
tuple <int, double, std::string> t1;
tuple <char, short, const char*> t2;
t1 = t2; // Ok car int = char et double = short possible
         // et std::string(const char*) existe
```

Les opérateurs relationnels sont disponibles pour les tuples ayant le même nombre d'arguments. Deux expressions sont ajoutées pour vérifier les caractéristiques d'un tuple à la compilation :

- `tuple_size<T>::value` retourne le nombre d'éléments du tuple `T` ;
- `tuple_element<i, T>::type` retourne le type de l'objet en position `i` du tuple `T`.

Lambda fonctions et lambda expressions

```
[](paramètres){code}
```

Les lambda fonctions sont bien agréables pour ceux qui utilisent les algorithmes STL, car il devient enfin possible de les utiliser sans pour autant écrire une fonction qui ne sera utilisée qu'une seule fois.

Par exemple, si vous voulez aujourd'hui afficher le contenu d'un `std:vector<>` sur la console, deux possibilités s'offrent à vous : la boucle classique et la fonction `std::copy` avec les itérateurs de flux. Le code suivant vous rappelle celles-ci :

```
// à l'ancienne
for( std::vector<Objet>::iterator it = V.begin(); it
➥!= V.end() +it)
    std::cout << *it << " ";
// avec les algorithmes STL
std::copy( V.begin(), V.end(), std::ostream_iterator
➥ <const Objet>(cout, " ") );
```

Grâce aux lambda fonctions, vous pourrez désormais écrire le code suivant :

```
for_each( V.begin(), V.end(), [](const Objet& o) {
    std:cout << o << " ";
});
```

Leur intérêt est encore renforcé lorsque cela évite de coder des foncteurs ou des fonctions. Par exemple, le code suivant montre à quel point il sera simple de rechercher un élément de conteneur respectant certains critères :

```
std::find_if( V.begin(), V.end(), [](const Objet& o) {
    return w.poids() > 98;
});
```

En fin de compte, tous les algorithmes STL de type boucle pourront être utilisés comme des boucles. Le `});` va devenir un classique…

Annexe C

Conventions d'appel x86

Les bibliothèques de codes, sous forme de bibliothèques dynamiques (.DLL sous Windows, .so sous Unix/Linux) ou de bibliothèque statiques (souvent .lib sous Windows et .a sous Unix/Linux), ne sont pas toutes écrites en C ou C++. C'est pourquoi vous rencontrerez certainement des mots-clés tels que `stdcall`, `safecall`, `cdecl` ou `WINAPI` (ce dernier est en fait une macro déclarée par les API Windows). Ils correspondent à des conventions d'appel et définissent comment appeler des fonctions – souvent compilées séparément, peut-être même à l'aide d'un compilateur différent du vôtre – au moment de l'exécution du programme.

Une convention d'appel détermine :

- l'ordre dans lequel les paramètres sont transmis à la fonction ;
- où les paramètres sont placés (dans la pile ou dans des registres) ;
- quels registres peuvent être utilisés par la fonction ;
- qui de l'appelé (la fonction) ou de l'appelant (la partie de code qui l'appelle) doit nettoyer la pile après l'appel.

Un sujet lié à la convention d'appel est la décoration de nom (ou *name mangling*). Elle détermine comment lier les noms utilisés dans le code avec les symboles de noms utilisés par

le lieur (*linker*) en ajoutant automatiquement le nombre et le type des paramètres associés au nom d'une méthode.

Si vous voulez en savoir un peu plus sur l'architecture x86, n'hésitez pas à aller jeter un œil sur cette page web : **http://en.wikipedia.org/wiki/X86**.

Info

Il existe souvent de subtiles différences dans l'application de ces conventions par les divers compilateurs en usage. Il est donc souvent ardu d'interfacer des codes compilés par des compilateurs différents. Heureusement, les conventions utilisées pour les API standard (comme `stdcall`) sont implémentées de manière uniforme.

Microprocesseurs Intel *

Famille	Processeurs
40xx, 80xx et 80xx	*4004*, *4040*, *8080*, *8085*, 8086, 8088, 80186, 80188, 80286, 80386, 80486, 486SL, 486SX, 486DX
Pentium	Pentium, Pentium MMX
P6	Pentium Pro, Pentium II, Pentium III
NetBurst	Pentium 4, Pentium 4-M, Pentium D, Pentium Extreme Edition
Mobile Architecture	Pentium M, Core Solo et Duo
Core Architecture	Core 2 Solo, Duo et Quad, Core 2 Extreme
Nehalem	Core i7
Serveur / Calculateur	Xeon, *Itanium*, *Itanium 2*
Autres	Atom, Celeron, Centrino, Pentium Dual-Core
XScale	*PXA250*, *PXA255*, *PXA260*, *PXA270*, *PXA290*
RISC	*iAPX 432*, *i860*, *i960*

* Processeurs non x86 en italiques

Microprocesseurs AMD

Famille	Processeurs
Architecture	K5, K6, K7, K8, K8L, K10
Socket	Socket 7, Socket Super 7, Socket A, Socket 563, Socket S1, Socket 754, Socket 939, Socket 940, Socket AM2, Socket AM2+, Socket F, Socket F+, Socket AM3
Processeurs antérieurs à K7	Am2900, AMD 29000, 8086, 8088, 80286, Am286, Am386, Am486, Am5x86, SSA5, 5k86, AMD K6, AMD K6-2, AMD K6-III
Athlon	Athlon, Athlon XP, Athlon 64, Athlon64 X2, Athlon FX
Phenom	Phenom 8000, Phenom 9000, Phenom FX
Duron / Sempron	Duron, Sempron
Mobile	Mobile Athlon XP, Mobile Athlon 64, Turion 64, Turion 64 X2, Turion 64 Ultra
Serveur	Athlon MP, Opteron
Autres	Geode, Alchemy
Chipset	ATI, AMD

Convention d'appel cdecl

```
void __cdecl f(float a, char b);
                          // Borland et Microsoft
void __attribute__((cdecl)) f(float a, char b);
                          // GNU GCC
```

Convention d'appel par défaut en C, elle l'est aussi en C++. L'appelant est responsable du désempilement des arguments, permettant ainsi d'utiliser des fonctions à nombre d'arguments dynamiques (définies à l'aide des points de suspension ...).

Les paramètres de la fonction concernée sont placés sur la pile de droite à gauche (dans l'ordre inverse de celui de la déclaration de la fonction). La valeur de retour est placée dans le registre EAX, sauf pour les valeurs flottantes qui se trouveront dans le registre flottant FP0. Les registres EAX, ECX et EDX sont libres d'être utilisés par la fonction. La fonction appelante nettoie la pile après le retour de l'appel.

Il existe quelques différences dans l'implémentation de `cdecl`, notamment vis-à-vis de la valeur de retour. Du coup, des programmes compilés pour différents systèmes d'exploitation et/ou par différents compilateurs peuvent être incompatibles, même s'ils utilisent la convention `cdecl` et ne font pas appel à l'environnement sous-jacent.

Microsoft C++ sous Windows retournera les structures de données simples de moins de 8 octets dans EAX:EDX. Les plus grandes structures ou celles nécessitant un traitement particulier lié aux exceptions (tels un constructeur, un destructeur ou un opérateur d'affectation particulier) sont retournées en mémoire.

G++ sous Linux transmet toutes les structures ou classes en mémoire, quelle que soit leur taille. En outre, les classes qui ont un destructeur sont toujours passées par référence, même pour les paramètres par valeur. Pour ce faire, l'appelant alloue la mémoire nécessaire et utilise un pointeur sur celle-ci comme premier paramètre caché. L'appelé la remplit et retourne ce pointeur tout en supprimant le paramètre caché.

Convention d'appel pascal

```
void __pascal f(float a, char b);
                              // Borland et Microsoft
```

Les paramètres sont placés sur la pile de la gauche vers la droite, à l'opposé de `cdecl`, et l'appelé est responsable du nettoyage de la pile.

Dans notre cas, l'appel `f(5,d);` correspondrait au code assembleur suivant :

```
PUSH EBP
MOV EBP, ESP
PUSH $00000005
PUSH d
CALL f
```

Astuce

Que faire si votre compilateur ne supporte pas cette directive et que vous deviez employer une bibliothèque utilisant la convention `pascal` ? Vous pouvez ruser en déclarant les en-têtes des fonctions qui vous intéressent en `stdcall` mais en inversant les paramètres. Par exemple, avec gcc :

```
// int __pascal f(int a, int b, int c);
int __attribute__((stdcall)) f(int c, int b, int a);
```

Convention d'appel `stdcall`

```
void __stdcall f(float a, char b);
                          // Borland et Microsoft
void __attribute__((stdcall)) f(float a, char b);
                          // GNU GCC
```

Cette convention est une variante de la convention `pascal` dans laquelle les paramètres sont passés par la pile, de droite à gauche. Les registres EAX, ECX et EDX sont réservés à l'usage de la fonction. La valeur de retour se trouve dans le registre EAX.

Convention d'appel `fastcall`

```
void __fastcall f(float a, char b);
                           // Borland et Microsoft
void __attribute__((fastcall)) f(float a, char b);
                          // GNU GCC
```

Cette convention n'est pas toujours équivalente suivant les compilateurs. Utilisez-la avec précaution.

Les deux premiers (parfois plus) arguments 32 bits sont transmis à la fonction dans des registres : le plus souvent dans EDX, EAX et ECX. Les autres arguments sont transmis par la pile, en ordre inverse (de droite à gauche) comme `cdecl`. La fonction appelante est responsable du nettoyage de la pile s'il y a lieu.

Attention

En raison de l'ambiguïté induite par les différentes implémentations de cette convention, il est recommandé d'utiliser `fastcall` uniquement pour les fonctions ayant 1, 2 ou 3 (selon le compilateur) arguments 32 bits et lorsque la vitesse d'exécution est essentielle.

Convention d'appel thiscall

`thiscall`

Cette convention est utilisée par les fonctions membres non statiques. Deux versions de `thiscall` se côtoient en fonction du compilateur et de l'utilisation du système de nombre d'arguments variable.

Avec GCC, `thiscall` est pratiquement identique à `cdecl` : la fonction appelante nettoie la pile, et les paramètres sont passés de droite à gauche. La différence tient dans l'ajout du pointeur `this`, ajouté en dernier à la pile comme si c'était le premier argument de la fonction.

Avec Microsoft Visual C++, le pointeur `this` est transmis par le registre ECX et c'est l'appelé qui nettoie la pile, à l'instar de la convention `stdcall` utilisée en C pour ce compilateur et de l'API Windows. Si la fonction utilise la syntaxe … des paramètres variables, c'est à l'appelant de nettoyer la pile (comme pour `cdecl`).

Conventions d'appel x86

Mot-clé	Ordre/Nettoyage	Remarque
`cdecl`	DàG/Appelant	Souvent la convention par défaut
`pascal`	GàD/Appelé	Convention OS/2 16 bits, Windows 3.x et certaines versions de Borland Delphi
`stdcall`	DàG/Appelé	Convention de l'API Win32
`fastcall`	DàG/Appelé/GCC DàG/Appelé/MSVC GàD/Appelé/Borland DàG/Appelé/Watcom	2 registres (ECX et EDX) avec __msfastcall 2 registres (ECX et EDX) avec `__fastcall` 3 registres (EAX, EDX et ECX) jusqu'à 4 registres (EAX, EDX, EBX et ECX) en ligne de commande (voir **http://www.openwatcom.org/index.php/Calling_Conventions#Specifying_Calling_Conventions_the_Watcom_Way**)
`thiscall`	DàG/Appelant/GCC DàG/Appelé/MSVC	 versions 2005 ou supérieurs

Conventions d'appel spécifiques

Mot-clé	Ordre/Nettoyage	Remarque
`register`	DàG/Appelé	Anciennement utilisé par Borland pour le `fastcall`
`safecall`	------/Appelé	Utilisé en Delphi Borland pour encapsuler les objets COM/OLE
`syscall`	DàG/Appelant	Standard pour les API OS/2
`optlink`	DàG/Appelant	Utilisé par les compilateurs VisualAge d'IBM

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

LE GUIDE DE SURVIE

C++

L'ESSENTIEL DU CODE ET DES COMMANDES

Ce *Guide de survie* est le compagnon indispensable pour programmer en C++ et utiliser efficacement les bibliothèques standard STL et BOOST, ainsi que QT, wxWidget et SQLite. Cet ouvrage prend en compte la future norme C++0x.

CONCIS ET MANIABLE

Facile à transporter, facile à utiliser — finis les livres encombrants !

PRATIQUE ET FONCTIONNEL

Plus de 150 séquences de code pour programmer rapidement et efficacement en C++.

Vincent Gouvernelle est ingénieur en informatique, diplômé de l'ESIL à Marseille, et titulaire d'un DEA en informatique. Il travaille actuellement chez Sescoi R&D, société éditrice de logiciels spécialisés dans la CFAO (conception et fabrication assistée par ordinateur).

Niveau : Intermédiaire / Avancé
Catégorie : Programmation
Configuration : Multiplate-forme

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4011-5

9 782744 040115