PRG1 - Référence C++ v. 06.10.2025

Préprocesseur

```
#include <iostream>                 // Insère un fichier d'en-tête standard
#include "monfichier.hpp"           // Insère un fichier d'en-tête local
#ifndef NOM_HPP                     // Empêcher l'inclusion multiple de
#define NOM_HPP                     // fichiers d'en-tête
…                                   // Code à inclure si non déjà inclu
#endif
#pragma once                        // Inclusion unique, non standard
#define N 42                        // Macro
#undef N                            // Supprime une macro
#if / #elif / #else / #endif        // Compilation conditionnelle
```

Constantes littérales

```
255, 0377, 0xff                     // Entiers (décimal, octal, hex)
123L, 234UL, 345LL, 456ull          // Entiers (un)signed (long) long
123.0f, 123.0, 1.23e2L              // Réels (float, double, long double)
true, false                         // Constantes booléennes 1 et 0
'a', '\141', '\x61'                 // Caractères (littéral, octal, hex)
'\n', '\\', '\'', '\"'              // newline, backslash, apostrophe, guillemet
"hello" "world"                     // Chaînes concaténées de type const char*
"hello"s                            // Chaîne littérale de type std::string
```

Note: "hello" est en réalité `const char t[] { 'h', 'e', 'l', 'l', 'o', '\0' };`

Déclarations

```
int x;                              // Déclare x entier
int x = 42; int x(42) ; int x{42};  // Déclare et initialise x entier
signed char a; unsigned char b;     // Entier sur 8 bits, [-128,127] et [0,255]
char a = 'A';                       // Caractère sur 8 bits, ASCII jusque 127
short a; unsigned short b;          // ⩾ 16 bits, [-32768,32767] et [0,65535]
int a; unsigned b;                  // ⩾ 16 bits, typ. 32, [-2E9,2E9] et [0,4E9]
long a; unsigned long b;            // ⩾ 32 bits, 32 ou 64 selon OS
long long a; unsigned long long b;  // ⩾ 64 bits, [-9E18,9E18] et [0,1.8E19]
float a; double b; long double c;   // Réel sur 32, 64 et généralement 80 bits
                                    // 23, 52 et 64 bits de mantisse
bool a = true;                      // Booléen. false vaut 0, true (1 en int)
int a, b, c = 3, d;                 // Déclaration multiple
auto a = 1u;                        // Type de a = type de sa valeur initiale
int a[10];                          // Tableau de 10 entiers, non initialisé
int a[] = {1, 2, 3};                // Tableau de 3 entiers initialisé
int a[5] = {1, 2};                  // Tableau de 5 int initialisé à {1,2,0,0,0}
const int a = 0; int const b = 1;   // Constantes, doivent être initialisés,
                                    // ne peuvent être affectées.
constexpr int a = 0;                // constexpr sont évaluées à la compilation
int& r = x;                         // r est une référence à x (synonyme)
const int& r1=a; int const& r2=a;   // Références constantes non modifiables
enum HAlign {left, center, right};  // Définit le type énuméré HAlign (C)
enum HAlign a; HAlign b;            // Variables de type HAlign
enum class VAlign {up, down};       // Définit le type énuméré VAlign (C++11)
VAlign a = VAlign::up;              // Les valeurs enum class sont qualifiées
using Entier = int;                 // Entier est un synonyme du type int;
typedef int Entier;                 // Même chose, antérieur à C++11
```

Classes de stockage

```
int x;                                  // Globale si hors de tout {}. Init. à 0
{ int x; };                             // Locale au bloc {}. Pas initialisée,
                                        // n'existe que pendant l'exécution du bloc
{ static int x; static int y = a;}      // Durée de vie globale, init. à zéro ou
                                        // autre au premier passage
static int x;                           // Globale, visible seulement dans ce fichier
extern int x;                           // Doit être définie ailleurs
```

Instructions

```
x = y;                                  // Affectation
int x;                                  // Déclaration
;                                       // Instruction vide
{ int x; a; }                           // Un bloc équivaut à une instruction.

if (x) a;                               // Si x est vrai (true), évaluer a
else if (y) b;                          // Sinon, si y est vrai, évaluer b
else c;                                 // Sinon (ni x ni y), évaluer c (optionnel)

while (x) a;                            // Répéter 0 fois ou plus, tant que x est vrai
for (x ; y ; z) a;                      // Équivalent à x; while (y) { a; z; } sauf si
                                        // a inclut une instruction continue
do a; while (x);                        // Équivalent à a; while(x) a;

switch(x) {                             // x doit être énumérable (entier, enum)
  case X1: a;                           // Si x==X1 (expression constante), saute ici
  case X2: b;                           // Sinon, si x==X2, saute ici
  default: c;                           // Sinon, saute ici; pas de break implicite
}

break;                                  // Sort le while,do,for,switch le plus interne
continue;                               // Passse à l'itération suivante while,do,for
return;                                 // Sort de la fonction sans valeur de retour
return x;                               // Sort de la fonction et retourne x

throw a;                                // Lève une exception. sort de toute fonction
                                        // jusqu'à ce qu'elle soit attrapée
Try { a; }                              // Évalue a et contrôle ses exceptions
catch (T& t) { b; }                     // Si a lève une exception de type T, évalue b
catch (…) { c; }                        // Si a lève une autre exception, évalue c
```

Fonctions

```
double f(int x, short s);               // Déclare la fonction f avec 2 paramètres int
                                        // et short et retournant un double
int f(int x) { instructions; }          // Définit une fonction f. Portée globale.
void f();                               // Déclare f sans param. ni valeur de retour
void f(int a = 42);                     // f() appelle f(42)
void f(int a, int& b, const int& c); // Paramètres passés par valeur,
                                        // référence ou référence constante
void f(int t1[], const int t2[]);       // Tableaux passés variables ou constants
T operator+(T lhs, T rhs);              // a+b de type T appelle operator+(a,b)
T operator-(T x);                       // -a de type T appelle operator-(a)
T& operator++();                        // ++i retourne une référence à i
T operator++(int);                      // i++ retourne la valeur prédente de i
ostream& operator<<(ostream& o, const T& t);    // surcharge << pour l'affichage
constexpr                               // Valeur de retour connue à la compilation
noexcept                                // La fonction ne lève pas d'exception
```

Paramètres et valeurs de retour peuvent être de tout type. Toute fonction doit être déclarée ou définie avant utilisation. Elle peut être déclarée avant et définie ensuite.

Un programme consiste en un ensemble de déclarations de variables et de définitions de fonctions globales (éventuellement dans plusieurs fichiers), dont exactement une doit être

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

`argv` étant un tableau de `argc` chaînes de caractères contenant les arguments de la ligne de commande. Par convention, main retourne 0 en cas de succès, 1 ou plus en cas d'erreur.

Chaque fichier `.cpp` doit contenir ou inclure une seule fois la déclaration de toute fonction qu'il utilise. Chaque fonction utilisée doit être définie une et une seule fois dans l'ensemble des fichiers `.cpp`

La surcharge des fonctions est résolue via leurs paramètres selon l'ordre suivant : type exact, promotion numérique vers **`int`** ou **`double`**, conversions de type. Si elle est appelable, une référence variable est préférée à une référence constante. Si la fonction a plusieurs paramètres, l'ensemble des fonctions candidates est établi séparément pour chaque paramètre et l'intersection de ces ensembles est considéré. Si la résolution de surcharge ne peut choisir une unique fonction, l'appel est ambigu et ne compile pas.

Expressions

Les opérateurs sont groupés par ordre de précédence. Opérateurs unaires et affectation s'évaluent de droite à gauche, les autres de gauche à droite. La précédence n'affecte pas l'ordre des évaluations qui est indéfini.

```
T::x                        // Nom x défini dans la classe T
N::x                        // Nom x défini dans l'espace de noms N
::x                         // Nom global x

t.x                         // Membre x de l'objet (classe ou structure) t
p->x                        // Membre x de l'objet pointé ou itéré par p
a[i]                        // (i+1)ième élément de a, les indices commencent à 0
f(x, y)                     // Appel à la fonction f avec les arguments x et y
T(x, y)                     // Objet de classe T construit avec arguments x et y
x++                         // Incrémente x, retourne une copie du x original
x--                         // Décrémente x, retourne une copie du x original
dynamic_cast<T>(x)          // Conversion au type T, vérifié à l'exécution
static_cast<T>(x)           // Conversion au type T, non vérifié
reinterpret_cast<T>(x)      // Interprète les bits de x comme étant de type T
const_cast<T>(x)            // Transforme variable T en constante ou vice-versa

sizeof(T)                   // Nombre de bytes pour stocker un objet de type T
sizeof x                    // Idem pour l'objet x, qui n'est pas évalué
++x                         // Incrémente x, retourne une référence à x
--x                         // Décrémente x, retourne une référence à x
-x                          // Moins unaire
+x                          // Plus unaire. Sans effet sauf promotion numérique
!x , not y                  // Négation: vrai si x est faux, faux sinon
*x                          // Déréférence le pointeur / l'itérateur x
&x                          // Adresse en mémoire de la variable x
(T) x                       // Conversion au type T. (obsolète)

x * y                       // Multiplication
x / y                       // Division (réelle ou entière selon arguments)
x % y                       // Reste (x,y entiers ; x%y de même signe que x)

x + y                       // Addition
x - y                       // Soustraction

x << y                      // Décalage des bits à gauche, retourne x·2^y
x >> y                      // Décalage des bits à droite, retourne x/2^y
                            // Opérateurs surchargés pour ostream& et istream&
x < y                       // Strictement plus petit
```

```
x <= y                      // Plus petit ou égal. Équivalent à not(y<x)
x > y                       // Strictement plus grand. Équivalent à (y<x)
x >= y                      // plus grand ou égal. Équivalent à not(x<y);
x == y                      // Égalité
x != y                      // Inégalité

x & y                       // et appliqué bit par bit : autre syntaxe: x bitand
x bitand y                  // et bit par bit (syntaxe alternative)

x ^ y                       // ou exclusif appliqué bit par bit (alternat. xor)

x | y                       // ou appliqué bit par bit (alternative : bitor)

x && y                      // et logique. y n'est évalué que si x est vrai
x and y                     // et logique (syntaxe alternative)

x || y                      // ou logique. y n'est évalué que si x est faux
x or y                      // ou logique (syntaxe alternative)

x = y                       // Affecte la valeur de y à x, retourne référence à x
x += y                      // x=x+y; aussi -=, *=, /=, %=, <<=, >>=, &=, |=, ^=

x ? y : z                   // y si x est vrai, z sinon. L'autre n'est pas évalué

throw x                     // lève l'exception x

x , y                       // évalue x puis y. retourne y
```

Classes

```
class T {                       // Déclare/définit un nouveau type T
private:                        // Accessible seulement depuis les méthodes de T
public:                         // Accessible depuis partout
  int x;                        // Attribut
  void f();                     // Méthode, aussi appelée fonction membre
  void g() { }                  // Méthode définie en ligne
  void h() const;               // Méthode ne modifiant pas les attributs
  int operator+(int y);         // t+y appelle t.operator+(y)
  int operator-();              // -t appelle t.operator-()
  T() : x(1) {}                 // Constructeur avec liste d'initialisation
  T(const T& t) : x(t.x) {}     // Constructeur de copie
  T& operator=(const T& t) { x = t.x; return *this;} // Opérateur d'affectation
  ~T();                         // Destructeur
  explicit T(int a);            // Permet T t(3); mais pas T t=3;
  operator int() const;         // Conversion vers le type int; permet int(t);
  friend void i();              // La fonction i() a accès aux membres privés
  friend class U;               // Les méthodes de U ont accès aux membres privés
  static int y;                 // Attribut partagé par tous les objets de type T
  static void k();              // Fonction T::k(), a accès à y mais pas à x
  class Z{};                    // Classe imbriquée T::Z
  using V = int;                // T::V est synonyme de int
};
void T::f() {                   // Définition de la méthode f de la classe T
  this-> x = x; }               // this est l'adresse de l'objet lui-même
int T::y = 2;                   // Initialisation d'un attribut statique
T::k();                         // Appel à la méthode statique k de T
… = delete                      // Suppression de constructeur
… = default                     // Opérateur par défaut
```

S'ils ne sont pas explicitement définis ou effacés, toute classe a un constructeur de copie et opérateur d'affectation par défaut qui copient tous les attributs. Si aucun constructeur n'est explicitement défini, il y a aussi un constructeur par défaut sans paramètre.

Templates

```
template <typename T> T f(T t);              // Surcharge de f pour tout type T
template<> bool f(bool);                     // Spécialisation de f pour le T=bool
template char f<char>(char);                 // Instanciation explicite pour T=char
extern template int f<int>(int);             // Déclaration sans instanciation
f<unsigned>(3u);                             // Appel et instanciation implicite
f(3.0);                                      // Appel et déduction de T=double
template <typename T> class X {              // Classe dont le type T est générique
  X(T t);                                    // Déclaration du constructeur de X<T>
  void g();                                  // Méthode de X<T>
  template <typename U> void h(U);           // Méthode générique de X<T>
  friend f<>(T t); }                         // Fonction générique amie
template <typename T> X<T>::X(T t) {}        // Définition du constructeur
template <typename T> void X<T>::g() {}      // Définition de la méthode g
template<> void X<bool>::g() {}              // Spécialisation de X<T>::g pour T=bool
template<> class X<char> { … }               // Spécialisation de toute la classe X
X<int> x(3);                                 // Un objet de type « X de int »
template <typename T, class U=T, int n=0>    // Template avec paramètres par défaut
  class Y {};                                // Classe générique à param. multiples
template <typename T> class Y<T,int,3> {}; // Spécialisation partielle
template <typename T, template<typename> class Conteneur>
  class Pile { Conteneur<T> data };          // Permet d'écrire Pile<int,vector> p;

template <typename T> using Tab = array<T,10> ;    // Tab<int> est array<int,10>
template <typename T> const T PI = T(3.1415926535897932385);
```

Exceptions et fin de programme

```
void f();                          // f peut lever des exceptions
void g() noexcept;                 // g ne lève pas d'exception
void h() noexcept(c);              // h ne lève pas d'exception si c est vrai
return 0;                          // Sortie normale du programme depuis main()
exit(0);                           // Sortie normale du programme depuis tout
                                   // ne détruit pas les objets automatiques
atexit(f); atexit(g);              // g(); f(); seront appelés en sortie normale
abort();                           // Sortie anormale, vers le débogueur
terminate();                       // Fonction appelée quand une exception n'est
                                   // pas capturée. Elle appelle abort(), pas exit()
set_terminate(f);                  // f() sera appelée par terminate() avant abort()
```

<stdexcept>

Met à disposition des sous-classes dérivées de `std::exception` pour la gestion d'un certain nombre d'exceptions spécifiques

```
throw logic_error("Pas d'opérateur < disponible"); // Lève une exception standard
try { f();}                                        // Bloc protégé
catch (logic_error & e) {cerr << e.what();}        // Traitement d'une exception
catch(...) {cerr << "Autre";}                 // Récupère tous les types d'exceptions

invalid_argument("constructeur faux") // Erreur spécifique, dérive de logic_error
length_error("Dépassement taille max")             // Dérive de logic_error
out_of_range("Indices hors tableau")               // Dérive de logic_error
bad_alloc()      // Retourné par new (mémoire insuffisante), dérive de exceptions
range_error("Valeur trop grande")                  // Dérive de runtime_error
overflow_error("Dépassement du type int")          // Dérive de runtime_error
```

Namespaces

```
namespace N { void f() {}; }     // Cache le nom f
N::f();                          // Utilise f dans l'espace de nom N
using namespace N;               // Rend f visible sans N::
```

<fstream>

```
fstream fs("f.txt", ios::in | ios::out); // Crée fs et ouvre f.txt en lecture
                                        // et écriture
ios::app, ios::trunc                    // Ajout à la fin, écrasement
fs >> x >> y;                           // Lit x et y depuis le flux fs
fs << x << y;                           // Écrit x et y dans le flux fs
fs.close();                             // Fermeture du fichier
fs.open("f.txt", ios::in | ios::out);   // Connecte le flux fs au fichier f.txt
ifstream is("f.txt");                   // Création du flux is (en lecture)
c = is.get();                           // Lit 1 char
is.get(c);                              // Lit 1 char. Retourne is
is.unget();                             // Remettre le dernier caractère lu
getline(is, string& s, char d = '\n');  // Lit jusqu'à d, stocke le résultat
                                        // dans s
is.ignore(size_t n = 1, int delim = EOF);// Ignore au max n char
ofstream os("f.txt");                   // Création du flux os (en écriture)
os.put(c);                              // Écrit le caractère c dans le flux os
os.flush();                             // Force l'écriture
```

<iostream>

```
cin >> x >> y;                // Lit x et y depuis l'entrée standard
cout << "x=" << 3 << flush ;  // Affiche x=3 et force l''impression
cerr << x << y << endl;       // Affiche x et y, passe à la ligne, sans tampon
c = cin.get();                // Lit 1 char
if(cin)                       // Teste si cin.good()
cin.clear()                   // Met cin à l'état good
cin.ignore(n,'\n');           // Ignore n caractères ou jusqu'au saut de ligne
```

<iomanip> et <ios>

```
cout << setw(6) << setprecision(3) << setfill('*') << 31.41592;     // **31.4
cout << fixed << setprecision(3) << 31.41592;                       // 31.416
cout << scientific << setprecision(3) << 31.41592;                  // 3.142e+01
cout << setfill('*') << left << setw(4) << +2;                      // 2***
cout << setfill('*') << right << setw(4) << -2;                     // **-2
cout << showpos << setfill('*') << internal << setw(4) << +2;       // +**2
cout << oct << 16 << ' ' << hex << 16 << ' ' << hex << 16;          // 20 16 10
```

<cmath>

```
sin(x); cos(x); tan(x);               // Fct trigonométriques. x en radian
asin(x); acos(x); atan(x);            // Fonctions trigonométriques inverses
sinh(x); cosh(x); tanh(x);            // Fonctions hyperboliques
exp(x); log(x); log10(x); log2(x);    // e^x, logarithmes en base e, 2, 10
pow(x,y); sqrt(x); hypot(x, y);       // x^y (imprécis sur entier), x^1/2, (x^2+y^2)^1/2
ceil(x); floor(x); trunc(x)           // Entier supérieur / inférieur / tronqué
round(x); round(1.5); round(-3.5);    // Entier le plus proche / 2 / -4
fabs(x); fmod(x,y);                   // Valeur absolue, x modulo y
```

<limits>

```
numeric_limits<T>::max();             // Plus grande valeur du type T
numeric_limits<T>::min();             // 1.17549e-38  pour T=float
numeric_limits<T>::lowest();          // -3.40282e+38 pour T=float
numeric_limits<T>::digits10;          // Chiffres significatifs. 6 pour T=float
numeric_limits<T>::epsilon();         // Précision relative: 1.2e-07 pour float
```

<numbers>

```
constexpr T e_v<T>; numbers::e;        // Constante e pour le type T; e_v<double>
log2e, log10e                          // Logarithme de e en base 2, en base 10
pi, inv_pi, inv_sqrtpi                 // π, 1/π, π^-1/2
sqrt2, sqrt3, inv_sqrt3                // 2^1/2, 2^1/3, 2^-1/3
egamma, phi                            // Gamma, nombre d'or
```

<cctype>

```
isalpha(c); isalnum(c);                // c est une lettre ? lettre ou chiffre
isdigit(c); isxdigit(x);               // c est un chiffre décimal? hexadécimal?
isblank(c); isspace(c);                // ' ' ou '\t'? ou '\n','\v','\r','\f'?
ispunct(c); iscntrl(c);                // Ponctuation ? contrôle (<32 ou 127)?
isprint(c); isgraph(c);                // Pas contrôle ? isprint, sauf ' ' ?
islower(c); isupper(c);                // c est minuscule ? majuscule ?
tolower(c); toupper(c);                // Convertit en minuscule / majuscule
```

<cassert> Macro d'arrêt du programme si une condition n'est pas remplie

```
assert(e);                             // Si e est faux, appelle abort();
#define NDEBUG                         // Désactivé si la macro est définie
                                       // avant #include <cassert>
```

<array> Tableaux de taille connue à la compilation

```
array<int,4> a;                        // Tableau de 4 entiers {0,0,0,0}
array<int,4> b(a);                     // b est une copie de a
array<int,4> c = {1, 2};               // Tableau {1,2,0,0}
a.size();                              // Nombre d'éléments de a
a.data();                              // Pointeur vers le premier élément
a.empty();                             // a.size() == 0
a[2];                                  // Référence au troisième élément de a
a.at(3);                               // a[3], throw si 3 >= a.size()
a.back();                              // a[a.size()-1];
a.front();                             // a[0];
for(auto i = a.begin(); i != a.end(); ++i) *i = 0;     // Parcours avec itérateur
for(auto i = a.cbegin(); i != a.cend(); ++i) cout << *i;     // Parcours constant
for(int e : a) cout << e;              // Parcours constant de tout a
for(int& e : a) e = 0;                 // Parcours permettant de modifier a
a.fill(42);                            // for(int& e : a) e = 42;
a = b;                                 // Copie de tout a par affectation
a == b; a < b; …                       // Comparaison lexicographique
```

<vector> Comme `array` (sauf fill), mais de taille et capacité variables, donc aussi …

```
vector<int> v(3);                      // Tableau de 3 entiers {0,0,0}
vector<int> w(v.begin(), v.end());     // w copie la séquence des éléments de v
vector<int> x(n,1);                    // x rempli de n fois la valeur 1
v.push_back(7);             // Ajoute l'élément 7 à droite. Incrémente v.size()
v.pop_back();               // Supprime l'élément de droite. Décrémente v.size()
v.insert(v.begin()+i, val);            // Insère val à l'indice i
v.insert(v.begin()+i, first, last);    // Insère la séquence [first,last[
v.erase(v.end()-2);                    // Retire l'avant dernier élément
v.erase(first,last);        // Retire les éléments aux emplacements [first,last[
v.assign(…);                           // v = vector<int>(…);
v.resize(n, 1);             // Modifie la taille, remplit avec 1 si elle augmente
v.clear();                             // v.resize(0);
v.capacity();                          // Nombre d'emplacements mémoire réservés
v.reserve(n);                          // Augmente la capacité si n>v.capacity()
v.shrink_to_fit();                     // Réduit la capacité à v.size()
```

<string>

Instanciation de `basic_string<char>`, comme `vector<char>`, mais avec fonctions-membres prenant en paramètre des `size_t` spécifiant position et longueur et non des itérateurs, et aussi …

```
string s1, s2 = "hello", s3(4,'a');   // "", "hello", "aaaa"
string s4(s2,1,2), s5(s2,3);          // "el", "lo" : sous-chaînes de s2
string s6("hello",4);                 // "hell" : 4 premiers char du const char[]
s1.length();                          // Synonyme de s1.size()
s1 += s2 + ' ' + "world";             // Concaténations
s1.append(s2, pos, len);              // Concaténation d'une sous-chaîne
s1.append("abc", 2); s1.append(4,'x');// Concaténation de "ab"; de "xxxx"
s1.compare(s2);                       // <0 si s1 < s2, 0 si s1 == s2, >0 sinon
s1.compare(pos, len, s2, subpos, sublen); // Comparaison de 2 sous-chaînes
s.substr(m,n);                        // Sous-chaîne de n char commençant à s[m]
size_t i = s.find(x,pos);             // Cherche depuis l'indice pos, retourne
                                      // string::npos si absent
size_t i = s.rfind(x,pos);            // Cherche de droite à gauche
size_t i = s.find_first_of(x,pos);    // x contient plusieurs char
string::npos;                         // size_t(-1)
s1.replace(pos, len, "xxx", 2);       // Remplace len char depuis pos par "xx"
to_string(3.14);                      // Convertit une valeur numérique en string
                                      // au format par défaut. Ici "3.140000"
```


Permet de gérer différents types de tableaux comme des `array<T, size_t n>`

```
int t[24]; span s(t); s[0] = s.back(); // Équivalent à t[0] = t[23];
span s(t, len);                         // s couvre les len premiers éléments de t
span u = s.first(len);                  // u couvre les len premiers éléments de s
span v = s.last(len);                   // v couvre les len derniers éléments de s
span w = s.subspan(pos, len);           // w couvre len élément à partir de pos
```

<algorithm>

Les séquences à traiter sont spécifiées avec 2 itérateurs `(first,last)` qui correspondent à la boucle

```
for(; first!=last; ++first) { *first; }
```

Si plusieurs séquences ont la même longueur, un seul `last` est demandé. `(first1, last1, first2)` correspond à

```
for(; first1!=last1; ++first1, ++first2) { *first1; *first2; }
```

Si nécessaire, un itérateur sur le premier élément non utilisé est retourné pour donner la longueur de la sortie. Par exemple,

```
vector<int> v{1,2,3,2,4,2};
auto it = remove(v.begin(),v.end(),2);
assert(it == v.begin()+3);    // et v contient {1,3,4,2,4,2}
```

Pour les autres paramètres, pour des séquences d'éléments de type `T`, on a

```
T val; bool upred(T); bool bpred(T,T); bool compare(T,T);
T uop(T); T bop(T,T); void f(T); T g(); int n; int rand(int);
```

Les paramètres optionnels sont en *italique*.

```
void for_each(first, last, f);                // Parcourt une séquence

int count(first, last, val);                  // Compte un nombre d'occurrence
int count_if(first, last, upred);

bool all_of(first, last, upred);              // Teste les éléments
bool any_of(first, last, upred);
bool none_of(first, last, upred);
bool equal(first1, last1, first2, bpred);     // Égalité de 2 séquences
```

```
Iter find(first, last, val);                        // Recherches, retournent last
Iter find_if(first, last, upred);                   // si pas trouvé
Iter find_if_not(first, last, upred);
Iter search(first1, last1, first2, last2, bpred);
Iter search_n(first1, last1, n, val, bpred);

Iter min_element(first, last, compare);             // position du min / max
Iter max_element(first, last, compare);

Iter transform(first1, last1, d_first, uop);        // transforme une séquence
Iter transform(first1, last1, first2, d_first,bop);


void fill(first, last, val);                        // Remplit avec une valeur
Iter fill_n(first, n, val);

void generate(first, last,g);                       // Remplit avec une
Iter generate_n(first, n,g);                        // fonction génératrice

Iter copy(first, last, d_first);                    // Copie une séquence
Iter copy_if(first, last, d_first, upred);
Iter copy_n(first, n, d_first);
Iter copy_backward(first, last, d_last);

Iter remove(first, last, val);                      // Supprime certains éléments
Iter remove_if(first, last, upred);
Iter remove_copy(first, last, d_first, val);
Iter remove_copy_if(first, last, d_first, upred);

void replace(first, last, oldval, newval);          // Remplace certains éléments
void replace_if(first, last, upred, val);
Iter replace_copy(first, last, d_first, oldval, newval);
Iter replace_copy_if(first, last, d_first, upred, newval);

Iter unique(first, last, bpred);                    // Supprime les doublons qui
Iter unique_copy(first, last, d_first, bpred);      // se suivent

void reverse(first, last);                          // Inverse la séquence
Iter reverse_copy(first, last, d_first);

void rotate(first, n_first, last);                  // n_first passe premier par
Iter rotate_copy(first, n_first, last, d_first);    // rotation

void sort(first, last, compare);                    // Tri rapide instable
void stable_sort(first, last, compare);             // Tri stable, plus lent
```

<numeric>

Certains algorithmes ont besoin d'une valeur initiale `T i;` Les opérateurs `+,-,*` peuvent optionnellement être remplacés par des fonctions binaires `T bop(T,T); e{e0, e1, …}`

```
T accumulate(first, last, i, bop+);                     // i + somme(e)
T inner_product(first1, last1, first2, i, bop+, bop*);  // i + somme((e1).*(e2))
void iota(first, last, i);                              // i, i+1, i+2, …
Iter adjacent_difference(first, last, d_first, bop-);   // e0, e1-e0, e2-e1, …
Iter partial_sum(first, last, d_first, bop+);           // e0, e0+e1, e0+e1+e2, …
```