PRG1 - Référence C++ v. 25.01.2023

```
Préprocesseur
#include <iostream>                  // insère un fichier d'en-tête standard
#include "monfichier.h"              // insère un fichier d'en-tête local

#ifndef _BLABLA_
#define _BLABLA_                     // empêche l'inclusion multiple de
…                                       fichiers d'en-tête
#endif
```

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, single/double quote
"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, 60 et 80 bits.
                                     // 23, 52 et 64 bits de mantisse.
bool a = true;                       // booléen. false = 0, true = 1
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 entiers 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 (synonyme) à x.
const int& r1=a; int const& r2=a;    // références constantes non affectables
enum HAlign {LEFT, CENTER, RIGHT};   // définit le type énuméré HAlign (C)
enum HAlign a; Align 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
typedef int Entier;                  // Entier est un synonyme du type int;
using Entier = int;                  // Entier est un synonyme du type int;
```

Classes de stockage

```
int x;                               // globale si hors de tout {}. Initialisée 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, initialisée à zéro ou
                                        autre au premier passage
static int x;                        // globale seulement dans ce fichier
extern int x;                        // globale définie dans un autre fichier
```

Instructions

```
x = y;                               // expression
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 (optionel)
else c;                              // sinon (ni x ni y), évaluer c (optionel)

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
}

break;                           // sort du while, do, for, switch le plus interne
continue;                        // saute à la fin de l'itération de while, do, for
return x;                        // sort de la fonction et retourne x à l'appelant

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 réel double
int f(int x) { instructions; }       // définit une fonction f. Scope global.
void f();                            // déclare f sans paramètre ni valeur retournée
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, T t);    // surcharge << pour l'affichage
```

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` chaines 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 surchage des fonctions est résolue via leurs paramètres selon l'ordre suivant : type exact, promotion numérique vers `int` ou `double`, conversions simples. 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 le namespace N
::x                         // Nom global x

t.x                         // Membre x de l'objet (classe ou struct) t
p->x                        // Membre x de l'objet pointé ou itéré par p
a[i]                        // (i+1)ème 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 x                    // Nombre de bytes pour stocker l'objet x
sizeof(T)                   // Nombre de bytes pour stocker un objet de type T
++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                          // addresse 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                       // modulo (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*pow(2,y)
x >> y                      // décalage des bits à droite, retourne x/pow(2,y)
                            // opérateurs surchargés pour ostream& et istream&

x < y                       // strictement plus petit
x <= y                      // plus petit ou égal. Équiv. à not(y<x);
x > y                       // strictement plus grand. Équiv à (y<x);
x >= y                      // plus grand ou égal. Équiv. à not(x<y);

x == y                      // égalité
x != y                      // inégalité

x & y                       // et appliqué bit par bit

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

x | y                       // ou appliqué bit par bit

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 {                       // Un nouveau type
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=3;
  operator int() const;         // 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'addresse de l'objet lui-même
int T::y = 2;                   // Initialisation d'un attribut static
T::k();                         // Appel à la méthode statique k de T
```

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);               // Instantiation explicite pour T=char
extern template int f<int>(int);           // Déclaration sans instantiation
f<unsigned>(3u);                           // Appel et instantiation implicite
f(3.0);                                 // Appel et déduction implicite 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 EXIT_SUCCESS;            // sortie normale du programme depuis main()
exit(EXIT_SUCCESS);             // sortie normale du programme depuis tout
atexit(f); atexit(g);           // g(); f(); seront appelés en sortie normale
abort();                        // sortie anormale, vers le debugger
terminate();                    // fonction appelée quand une exception n'est
                                   pas catchée. Elle appelle abort(), pas exit()
set_terminate(f);               // f() sera appelée par terminate() avant abort()
```

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::
```

<iostream>

```
cin >> x >> y;                             // Lit x et y depuis l'entrée standard
cout << "x=" << 3 << endl;                 // Affiche x=3 et passe à la ligne
cerr << x << y << flush;                   // Affiche x et y, force l'impression
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>

```
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
```

<cmath>

```
sin(x); cos(x); tan(x);                    // trigonométriques. x en radian
asin(x); acos(x); atan(x);                 // fonctions inverses
sinh(x); cosh(x); tanh(x);                 // 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, 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

<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>

```
assert(e);                              // si e est faux, arrête le programme
#define NDEBUG                         // Avant #include <cassert>, les désactive
```

<array>

```
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.empty();                              // a.size() == 0
a[2];                                   // référence à l'élément de a d'indice 2
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 itérateurs
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> comme vector<char>, mais aussi …

```
string s1, s2 = "hello", s3(4,'a');   // "", "hello", "aaaa"
string s4(s2,1,2), s5(s2,3);          // "el", "lo" : sous-chaines de s2
string s6("hello",4);                 // "hell" : 4 premiers char du const char[]
s1.length();                          // synonyme de s1.size()
s1 += s2 + ' ' + "world";             // concaténations
s.substr(m,n);                        // sous-chaine de n chars 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)
to_string(3.14);                      // convertit une valeur numérique en string
                                         avec le format par défaut. Ici "3.140000"
```

<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. E.g.,

```
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'occurence
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 random_shuffle(first, last, rand);           // mélange aléatoire
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, …
```