Aide-mémoire C++

Matmeca 2014

Marc Duruflé

Table des matières

1. Commentaires

```
/*
  Ceci est un commentaires sur
  plusieurs lignes    (C et C++) */

// Ceci est un commentaire sur une ligne (C++ uniquement)

double x = 3.0; // qui peut etre dispose a la suite d'une instruction
```

2. Types prédéfinis

Type	Description
void	type vide
bool	booléen
int	entier (sur 4 octets)
char	caractère (1 octet)
float	réel simple précision (4 octets)
double	réel double précision (8 octets)
unsigned int	entier positif
short	entier court
long	entier long
int64_t	entier long (sur 8 octets)

2.1. Types de base

- **void** Type vide, utilisé le plus souvent pour signaler qu'une fonction ne retourne rien.

```
// exemple de function qui ne renvoie rien
void affiche_reel(double x)
{
  cout << " x = " << x << endl;
}
```

- **bool** Booleen (true or false)

```
bool test = true;
// on peut les utiliser directement dans les conditions
if (test)
  cout << "Test est vrai " << endl;
```

- **char** Caractère (codé sur un octet)

```
// la casse est importante, a et A sont des caracteres differents
char lettre = 'A';
// la correspondance caractere <-> nombre est assuree par
// la table ascii, par exemple '0' = 48
char zero = '0', zero_bis = 48;
// il y a aussi des caracteres speciaux
```

```
// \t (tabulation), \n (retour chariot), etc
lettre = '\n';
```

- **int** Entier (codé sur 4 octets)

```
// pas d'overflow (depassement) si n est compris entre
// -2 147 483 647  et  2 147 483 647
int n = 345;

// par defaut un entier (ici 345) est ecrit en base decimale
// on peut entrer un literal en base hexadecimale
// en commencant par 0x
n = 0xA34E; // n = A34E en hexadecimal

// pour les entiers en base 8, on commence par 0
n = 023472;  // n vaut 10042
```

- **float** Réel simple précision (codé sur 4 octets).

Ce type représente des nombres réels entre 1e-38 et 1e38 avec une précision de 1e-7.

```
// pour declarer un literal simple precision, ajouter f :
float x = 1.23f;
```

- **double** Réel double précision (codé sur 8 octets).
Ce type représente des nombres réels entre 1e-308 et 1e308 avec une précision de 2e-16.

```
// pour declarer un literal double precision :
double x = 1.23, y = -5.04e10;
```

- **int64_t** Entier (codé sur 8 octets)

```
// a inclure pour avoir le type int64_t
#include<stdint.h>

using namespace std;

// pas d'overflow (depassement) si n est compris entre
// -9e18  et  9e18
int64_t n = 345;
```

2.2. Qualificatifs additionnels

- **short** Pour déclarer une variable de plus faible précision

```
// il n'y a pas de regle, mais par exemple short int est en general
// stocke sur 2 octets (-32 768 a 32 767)
short int n = 1045;
```

- **long** Pour déclarer une variable de plus grande précision

```
// il n'y a pas de regle, mais par exemple long double est en general
// stocke sur 10 octets (double precision etendue)
// mettre L a la fin des literaux
long double x = 0.1234L;
```

- **signed / unsigned** Pour déclarer des entiers non-signés (positifs) ou signés

```
// Par defaut les entiers sont signes, pour declarer un entier positif:
unsigned int i = 23;
// l'avantage est alors que i peut aller de 0 a 4 milliards

// les caracteres non signes existent aussi (allant de 0 a 256)
unsigned char c = 23;
```

3. Pointeurs et références

Un pointeur est une variable contenant l'adresse d'une autre variable. Le contenu du pointeur q est alors accédé par ∗ q et l'adresse d'une variable x est accédée par & x :

```
double x = 1.0 ; // x vaut 1.0
double y = 2.0 ; // y vaut 2.0
double* q; // on declare un pointeur sur un double
q = NULL; // on specifie ici que q ne pointe vers rien
q = &x; // q pointe maintenant vers la variable x
*q = 3.0; // x vaut 3.0
q = &y; // q pointe maintenant vers la variable y
*q = 4.0;
// y vaut maintenant 4.0
x = *q;
// x vaut maintenant 4.0

// on peut allouer un nouvel emplacement memoire pour q
// (qui sera ni x, ni y)
q = new double;
*q = 2.0; // x et y sont inchanges

q = &x; // ici c'est tres dangereux
// car on perd l'adresse reservee au-dessus
// on a alors ce qu'on appelle une fuite de memoire

// on peut aussi allouer plusieurs cases (pas une seule)
q = new double[3]; // 3 cases ici
// et acceder aux autres cases avec []
```

```
x = q[1];

// pour liberer la memoire, delete[]
delete[] q;

// les references peuvent aussi etre utilisees
// pour definir des alias (sans avoir a ecrire de *)
double& alias_x = x;

// tout changement sur alias_x se repercutera sur x et vice versa
alias_x = -1.0; // x et alias_x valent -1
```

4. Tableaux en C

4.1. Tableau statique

C'est un tableau dont on connait la taille lors de la compilation.

```
// par exemple un tableau de 4 reels :
// par defaut les valeurs de y ne sont pas initialisees
double y[4];

// on utilise l'operateur [] pour acceder aux elements du tableau
// les indices commencent a 0
y[0] = 3.03; y[1] = -2e10; y[2] = 5.54e3; y[3] = 2.02e-5;

// on peut initialiser tout le tableau avec les { } :
int n[4] = {-31, 43, 3, 10};
```

4.2. Tableau dynamique

C'est un tableau dont on fixe la taille lors de l'execution

```
// par exemple on demande la taille du tableau
int n;
cout << "Entrez n " << endl;
cin >> n;

// on alloue le tableau avec l'operateur new
double* x; // x pointeur vers la premiere case du tableau
x = new double[n];

// on utilise l'operateur [] pour acceder aux elements du tableau
// les indices commencent a 0
x[0] = 3.03; x[1] = -2e10; x[2] = 5.54e3; x[6] = 2.02e-5;

// on desalloue le tableau avec l'operateur delete[]
delete[] x;
```

4.3. Tableau à plusieurs entrées

La syntaxe des tableaux statiques multidimensionnels est la suivante :

```
// exemple de tableau 3x2 qu'on initialise
double A[3][2] = {{0.5, 0.6}, {-0.3, 0.1}, {0.9, -0.9}};

// on accede aux elements en ecrivant A[i][j]
cout << " A[1][0] " << A[1][0] << endl;

// on peut ecrire un tableau avec un nombre arbitraire d'indices
// par exemple pour un tableau tridimensionnel
int B[5][4][3];

B[2][0][4] = 3.4;
```

En C++, il n'y a pas d'équivalent natif aux tableaux dynamiques multidimensionnels comme il existe en Fortran90. Toutefois il est possible de construire des vecteurs de vecteurs, avec le désavantage que ce type de structure ne stockera pas les éléments de manière contigüe. En voici un exemple :

```
// pour un vecteur de vecteurs, on met deux etoiles
// on met autant d'etoiles qu'il y a d'indices
double** V;

// on alloue la liste des lignes :
int m = 5;
V = new double*[m];

// ensuite on alloue chaque ligne
// chaque ligne peut avoir une taille differente
// et les elements de la seconde ligne ne seront pas stockes
// physiquement juste apres ceux de la premiere ligne (stockage non contigu)
int n = 7;
for (int i = 0; i < m; i++)
  V[i] = new double[n];

// ensuite on modifie un element de V avec V[i][j]
V[0][4] = 3.0;

// on doit desallouer chaque ligne separement
for (int i = 0; i < m; i++)
  delete[] V[i];

// puis desallouer le tout
delete[] V;
```

5. Structures, énumérations

Une structure contient plusieurs champs, c'est l'équivalent des types composés Fortran.

```
struct NomStructure
{
  // nombre des invites
  int nombre_invites;
```

```
  // tableau contenant la taille des invites
  double* taille_invites;

  // nom de l'evenement
  char* nom;
};

// on declare une variable de type NomStructure
NomStructure repas;

// pour modifier les membres de la structure, operateur .
repas.nombre_invites = 10;

// on peut aussi declarer a la volee une structure
// sans donner de nom :
struct{ double x, y;} var;

// et modifier les membres de la variable :
var.x = 3.0; var.y = -2.0;

// on peut aussi utiliser le mot cle typedef
typedef struct { int i; double x; } NomStruct;

// puis declarer la variable de type NomStruct
NomStruct x;
```

Une énumération permet de définir des constantes entières dans un programme et de les nommer, ce qui rend le code plus lisible.

```
// par exemple on enumere toutes les conditions aux limites
// qu'on sait traiter avec le code :
enum{DIRICHLET, NEUMANN, IMPEDANCE};

// ces constantes sont des entiers, qu'on peut utiliser :
int condition = NEUMANN;

// ensuite, au lieu de faire un test
// if (condition == 1), on met le nom
if (condition == DIRICHLET)
  {
    // on fait le traitement de la condition de Dirichlet
  }
```

6. Fonctions

Syntaxe :

```
type_retour nom_fonction(arg1, arg2, arg_default = value_default);
```

6.1. Surcharge de fonctions

Une fonction est définie par ses arguments et son type de retour. On peut surcharger une fonction, c'est à dire avoir deux fonctions portant le même nom mais dont les arguments diffèrent. Le mot clé "return" renvoie la valeur de retour de la fonction et quitte la fonction.

```
// exemple trivial d'une fonction renvoyant x+y
double Add(double x, double y)
{
  return x + y;
}

// on peut surcharger cette fonction pour des arguments differents
int Add(int x, int y)
{
  return x + y;
}

double Add(double x, double y, double z)
{
  return x + y + z;
}

// on definit une procedure en mettant void comme type de retour
void test_procedure(double x, double y)
{
  // return sert ici a quitter la procedure
  if (x == y)
    return;

  // le programme continuera que si x est different de y
  cout << "1/(x-y) = " << 1.0/(x-y) << endl;
}

// on peut specifier des arguments optionnels a la fin des arguments
// on leur donne une valeur par defaut, ici z est un argument optionnel
// s'il n'est pas rempli, il vaut 1.0
double Mlt(double x, double y, double z = 1.0)
{
  return x*y*z;
}

int main()
{
  // pour appeler une fonction, on ecrit son nom avec les arguments
  double x = Add(1.3, 1.4);
  double y = Add(x, 0.45, 1.23);

  test_procedure(x, y);

  // on peut appeler Mlt avec trois arguments
  double z = Mlt(x, y, 2.3);

  // ou avec deux arguments, le dernier prenant
  // la valeur par defaut (1.0 ici)
  z = Mlt(x, y);

  return 0;
```

```
}
```

6.2. Passage par référence

Par défaut, les arguments d'une fonction sont passés par valeurs et ne sont donc pas modifiés. Une copie locale de ces arguments est faite.

```
// exemple de fonction avec des arguments usuels
double Add(double x, double y)
{
  // on peut modifier les copies locales de x et y
  // la valeur de x et y ne sera pas modifiee en sortie de fonction
  x += y;
  y += x;
  return x;
}

int main()
{
  double x = 3.2, y = 2.3;
  double z = Add(x, y);

  // x et y valent toujours 3.2 et 2.3
  return 0;
}
```

Pour modifier les arguments, on utilise des références, afin de passer l'adresse des arguments. Ainsi on évite la copie locale des arguments, qui peut etre coûteuse si certaines variables sont volumineuses.

```
// exemple d'une structure volumineuse
struct BigArray
{
 double A[1000];
};

// on passe x et tab par reference, ils peuvent etre modifies
// de plus aucune copie locale de tab n'est effectuee
void modify_argument(double& x, BigArray& tab)
{
  for (int i = 0; i < 1000; i++)
    tab.A[i] *= x;

  x += 1.0;
}

// on peut utiliser le mot-cle const pour specifier qu'un argument
// est inchange, meme s'il est passe par reference
// le but est ici d'eviter la copie locale d'un gros objet
void test_argument(const BigArray& t)
{
  t.A[0] = 2.0; // erreur a la compilation
}

int main()
```

```
{
  double x = 2.0;
  BigArray tab;
  for (int i = 0; i < 1000; i++)
    tab.A[i] = double(i);

  modify_argument(x, tab);
  // x et tab ont ete modifies

  return 0;
}
```

6.3. Passage par pointeur

Une méthode moins élégante est de passer les adresses des arguments. C'est une technique surtout employée pour le passage de tableaux.

```
// on reprend le meme exemple que precedemment
void modify_argument(double* x, BigArray* tab)
{
  // on doit alors adapter la fonction
  for (int i = 0; i < 1000; i++)
    tab->A[i] *= *x;

  *x += 1.0;
}

// le passage par pointeur est naturel pour un tableau
double sum(double* x, int n)
{
  // on renvoie la somme des elements du tableau x
  double sum = 0.0;
  for (int i = 0; i < n; i++)
    sum += x[i];

  return sum;
}

int main()
{
  double x = 2.0;
  BigArray tab;

  // on passe ici les adresses des objets
  modify_argument(&x, &tab);

  double* vec;
  int n = 10;
  vec = new double[n];

  // pour un tableau, on le passe directement
  // puisque c'est deja un pointeur
  x = sum(vec, n);
```

```
  delete[] vec;

  return 0;
}
```

6.4. Pointeurs et références de fonctions

Grace à ces fonctionnalités on peut construire des tableaux de fonctions ou passer une fonction en paramètre.

```
// une reference de fonction s'ecrit sous la forme
//   type_retour (&nom_fct)(arguments)

// un pointeur de fonction s'ecrit sous la forme
//   type_retour (*nom_fct)(arguments)

// par exemple on construit une fonction generique
// pour calculer l'integrale de f entre a et b, ou f
// est une fonction qui est passee en parametre
double integrate(double a, double b, int N, double (&f)(double))
{
  double sum = 0.0;
  double h = (b-a) / N;
  for (int i = 0; i < N; i++)
    {
      double x = (double(i) + 0.5)*h + a;
      sum += h*f(x);
    }

  return sum;
}

// exemple de fonction dont on veut calculer l'integrale
double func(double x)
{
  return x*x + sin(0.5*x);
}

int main()
{
  cout << "Integrale de exp = " << integrate(0.0, 1.0, 20, exp) << endl;
  cout << "Integrale de func = " << integrate(0.0, 1.0, 20, func) << endl;

  // on declare un nouveau type fptr qui est un pointeur de fonction
  // renvoyant un double avec un double en parametre
  typedef double (*fptr)(double);

  // on declare un tableau contenant trois fonctions de ce type
  fptr tab_fct[3];

  // on initialise le tableau
  tab_fct[0] = &exp;
```

```
  tab_fct[1] = &func;
  tab_fct[2] = &asin;

  for (int i = 0; i < 3; i++)
    cout << "Integrale de f_i = "
    << integrate(0.0, 1.0, 20, *tab_fct[i]) << endl;

  return 0;
}
```

6.5. La fonction main

La fonction main est la fonction appelée lors de l'exécution, il ne doit donc y avoir qu'une seule fonction main lorsqu'on réalise l'édition des liens.

```
int main()
{
  // on renvoie 0 si le programme s'est bien deroule
  // un autre entier s'il y a eu une erreur
  return 0;
}
```

On peut également récupérer les arguments de la ligne de commande :

```
// Par exemple si vous avez execute votre programme en tapant :
// ./a.out  toto 2.5  -4
// alors argc vaudra 4, argv[0] vaudra "./a.out"
// argv[1] "toto", argv[2] "2.5" et argv[3] "-4"
int main(int argc, char** argv)
{
  // les arguments sont des chaines de caractere
  // qu'on peut convertir en double avec atof, et en int avec atoi
  int n = atoi(argv[3]);
  double x = atof(argv[2]);
}
```

7. Variables

Les variables peuvent etre déclarées à n'importe quel endroit du programme (on dit qu'elles sont déclarées à la volée). La variable est détruite lors de la fermeture de l'accolade. Les variables globales sont déclarées à l'extérieur de toute fonction.

```
// par exemple on declare une variable globale
int Nglob = 10;

// elle sera connue dans toutes les fonctions qui suivent
void f(double x)
{
  cout << "Nglob = " << Nglob << endl;
}

void g(double y)
{
```

```
  double z = 2.0*y;
  for (int i = 0; i < Nglob; i++)
    z *= double(i+1);

  // on peut aussi declarer des variable locales a la fonction g
  double w = -5.43 + z;

  cout << "w = " << w << endl;

  // toutes les variables locales (ici w et z) sont detruites a la fermeture
  // de l'accolade
}

// ici w et z ne sont pas connues
w = 3.0; // erreur de compilation

// on peut aussi declarer des variables constantes, dont on
// ne peut modifier la valeur apres initialisation
const double epsilon = 1e-10;

epsilon = 3.0; // erreur de compilation
```

8. Structures de contrôle

8.1. Opérateurs

()	opérateur d'accès
[]	opérateur d'accès
.	membre d'un objet ou structure
->	raccourci pour (*pointeur).
!	opérateur non
~	non bit à bit
++	incrémentation ($i++$ est équivalent à $i = i + 1$)
−−	décrémentation ($i--$ est équivalent à $i = i - 1$)
*	cible d'un pointeur ou multiplication
&	référence
/	division
−	opposé d'un nombre ou soustraction
+	addition de deux nombres
%	modulo (a%b équivaut à mod(a, b) en Fortran)
<<	flux de sortie ou décalage de bit
>>	flux d'entrée ou décalage de bit
<	signe strictement inférieur
>	signe strictement supérieur
<=	signe inférieur ou égal
>=	signe supérieur ou égal
==	signe égal (pour la comparaison)
!=	signe différent
&	et bit à bit
^	ou exclusif bit à bit
|	ou bit à bit
&&	et logique
||	ou logique
? :	structure if/else compacte
=	affectation
*=	$a* = b$ équivaut à $a = a * b$
+=	$a+ = b$ équivaut à $a = a + b$
/=	$a/ = b$ équivaut à $a = a/b$
%=	$a\% = b$ équivaut à $a = a\%b$
−=	$a- = b$ équivaut à $a = a - b$
^=	$a\hat{} = b$ équivaut à $a = a\hat{} b$
&=	$a\& = b$ équivaut à $a = a\&b$
|=	$a\| = b$ équivaut à $a = a\|b$

On notera qu'il n'y a pas d'opérateur puissance (comme ** en Fortran), il faut utiliser la fonction pow (dans le package cmath). Voici un exemple d'utilisation de ces opérateurs.

```
// on suppose qu'un objet matrice est cree avec une surcharge de l'operateur ()
// on peut utiliser A(i, j) pour modifier un element de la matrice
Matrice A;
A(0, 3) = 2.3;

// l'operateur [] est utilise pour les tableaux, vecteurs, etc
vector<double> v;
v[0] = 1.2e10;
```

```
// on peut declarer un pointeur sur une matrice
// on suppose que l'objet Matrice contient une fonction GetM()
Matrice* B;

// on peut faire pointer B sur A
B = &A;

// et appeler GetM() soit avec *, soit avec ->
int n = (*B).GetM();
n = B->GetM();

// ! est l'operateur non, !(i>3) equivaut a (i<=3)
int i = 5;
if (! (i > 3) )
  i = 7;

// && : et logique, || : ou logique
if ( ((i==3) || (i == 7)) && (i <= 5))
  {
    // ici seul i = 3 devrait passer la condition
  }

// != est l'operateur different
if ((i != 3) && (i >= 2))
  {
    // i peut etre egal a 2 ou superieur ou egal a 4
  }

// ++ a gauche signifie qu'on incremente avant d'utiliser le resultat
// ++ a droite signifie qu'on utilise d'abord le resultat avant d'incrementer
v[++i] = 3.4; // equivaut a i = i+1; v[i] = 3.4;
v[i++] = 3.4; // equivaut a v[i] = 3.4; i = i+1;

// idem pour la decrementation --
v[i--] = 3.4; // equivaut a v[i] = 3.4; i = i-1;

// reste d'une division euclidienne (pour les entiers uniquement):
int r = i%2; // vaut 1 si i est impair

// on peut faire les operations classiques
// par exemple ici y = (x + 2.0)*3.0 - 1.0/x;
double x = 1.2, y = 0.8;
y = (x + 2.0)*3.0 - 1.0/x;

// ~ : non bit a bit
// par exemple ~(00000001011) = 1111110100 soit -12 en decimal
i = ~11; // devrait faire i = -12

// & : et bit a bit, et | : ou bit a bit, ^ : ou exclusif
i = 6 | 12; // devrait faire 0110 | 1100 = 1110 (i=14)
i = 6 & 12; // devrait faire 0110 & 1100 = 0100 (i=4)
```

```
i = 6 ^ 12; // devrait faire 0110 ^ 1100 = 1010 (i=10)

// << pour les sorties, >> pour les entrees
cout << "coucou" << endl;
cin >> i;

// *=, +=, /=, -=, %=, etc fonctionnent sur le meme principe :
i += 4; // equivaut a faire i = i + 4

// ? : pour un if else compact
x = (i >= 3) ? 1.23 : 0.4;
// equivaut a if (i>= 3) { x = 1.23;} else { x = 0.4; }
```

8.2. Structure conditionnelle if

Le mot clé else est optionnel.

```
// si il y a une seule instruction, pas d'accolade
if (i == 3)
  cout << "i vaut 3" << endl;
else
  cout << "i est different de 3" << endl;

// sinon on met des accolades
double x;
if (i == 3)
  {
    x = 0.45;
    cout << "i vaut 3" << endl;
  }
else if (i == 5)
  {
    x = 0.52;
    cout << "i vaut 5" << endl;
  }
else
  {
    x = -0.2;
    cout << "i est different de 3 et de 5" << endl;
  }
```

8.3. Structure conditionnelle switch

On ne peut ici que comparer à des constantes (entiers ou caractères). C'est l'equivalent du select case en fortran.

```
// par exemple sur un caractere
char lettre;
switch (lettre)
  {
    case 'a' :
     // instructions dans le cas ou lettre='a'
     cout << " lettre = a" << endl;
```

```
      // mettre break a la fin (obligatoire)
      break;

    case 'b' :
    case 'c' :
    case 'd' :
      // ici lettre vaut b c ou d
      cout << " lettre = b c ou d" << endl;
      break;
    default :
      // les autres cas non traites
      cout << "lettre inconnue" << endl;
  }
```

8.4. Boucle for

Syntaxe :

for (initialisation ; test ; itération) opération ;

Example :

```
// Comme pour if, pas d'accolade si une seule instruction
// ici on fait la somme des entiers de 0 a 9
int sum = 0;
for (int i = 0; i < 10; i++)
  sum += i;

// sum devrait etre egale a 45
// lorsqu'il y a plusieurs instructions, on met des accolades
// ici on decremente i par pas de 2
for (int i = 10; i >= 0; i -= 2)
  {
    // on doit obtenir i = 10, 8, 6, 4, 2, 0
    cout << "i = " << i << endl;
    sum -= i;
  }
```

8.5. Boucle while

Syntaxe :

while (condition) operation ;

do operation while (condition) ;

Exemple :

```
// dans un premier cas on teste la condition avant
// d'effectuer l'operation
int i = 0;
while (i < 10)
  i++;
```

```
// dans un second cas, on fait l'operation avant de tester la condition
i = 20;
do
  {
    cout <<" i = " << i << endl;
    i++;
  }
while (i < 10);
```

8.6. Instructions break continue

Exemple :

```
// continue permet de sauter les instructions placees apres
for (int i = 0; i < 10; i++)
  {
    if (i == 0)
      continue;

    // toutes les lignes qui suivent ne seront pas
    // executees pour i = 0, mais le seront pour i > 0
    cout << "i = " << i << endl;
  }

// break sort de la boucle
for (int i = 0; i < 10; i++)
  {
    if (i == 5)
      break;

    // ici a cause du break, on sort de la boucle
    // pour i = 5, les valeurs 6, 7, 8, 9 ne seront
    // jamais parcourues
  }
```

9. C++ : la couche objet

9.1. Encapsulation des données

On peut ici utiliser le mot-clé struct, par défaut les membres d'une struct sont publiques, alors que les membres d'une class sont privées par défaut.

Exemple :

```
class NomClasse
{
  // par defaut les membres sont privees
  int ip; // ici une variable membre (appelee attribut)

  // ici une fonction membre (appelee methode)
  void CheckPrivate()
  {
    cout << "classe ok" << endl;
  }
```

```
 protected:
 // tous les membres places apres sont ''proteges''
 int jp;

 void CheckProtected()
 {
   cout << "classe ok" << endl;
 }

  public :
  // tous les membres places apres sont publics
  int kp;

  void CheckPublic()
 {
   cout << "classe ok" << endl;
 }

 // on peut inserer a tout moment le mot cle
 // private public ou protected pour specifier le type des membres qui suivent
 private :
 int ip2;

};

int main()
{
  // dans une fonction exterieure a la classe
  // on n'a acces qu'aux donnees publiques
  NomClasse var;

  var.CheckPublic(); // OK
  var.kp = 3; // OK

  // pas d'acces aux donnees privees ou protegees
  var.ip = 1; // erreur de compilation
  var.CheckProtected(); // erreur de compilation

  return 0;
}
```

9.2. Héritage

Trois types d'héritage (public, private, protected), voici comment les types de données des attributs de la classe mère sont transformés après héritage :

Données	héritage public	héritage protected	héritage private
public	public	protected	private
protected	protected	protected	private
private	interdit	interdit	interdit

Exemple

```
// classe mere
class Mother
{
  // donnees privees
  private :
  int i_private;

  // donnees protegees
  protected :
  int i_protected;

  // donnees publiques
  public :
  int i_public;

};

// classe fille, par exemple pour un heritage public
class Daughter : public Mother
{
  void Test()
  {
    // on peut acceder a tous les membres de la classe mere
    i_protected = 0;

    // sauf les donnees privees
    i_private = 0; // erreur de compilation
  }
};

// pour un heritage protege
class Fille : protected Mother
{
};

class Father
{
  public:
  int j_public;
};

// on peut faire aussi un heritage multiple
class Fils : public Mother, public Father
{
};
```

9.3. Fonctions et classes amies

Exemple :

```
class Mother
{
  private :
```

```
  int ip;
  // fonction amie
  friend void affiche_ip(const Mother &);

  // classe amie
  friend class Amie;
};

// dans la fonction amie on peut acceder aux membres privees de la classe
void affiche_ip(const Mother & var)
{
  cout << " i = " << var.ip << endl;
}

// pour les classes amies :
class Amie
{
  public :
  // les methodes ont acces aux donnees privees de Mother
  void ChangeIp(Mother & var, int j)
  {
    var.ip = j;
  }
};

int main()
{
  Mother var;
  Amie a;

  a.ChangeIp(var, 3);

  affiche_ip(var);
}
```

9.4. Constructeur, destructeur

```
class Base
{
  protected :
  int i;
  double x;

  // les constructeurs et destructeurs sont publics
  public :
  // un constructeur n'a pas de type de retour
  // et a le meme nom que la classe
  Base()
  {
    // un constructeur sert a initialiser les attributs
    i = 0;
    x = 1.0;
```

```
  }

  // deuxieme constructeur
  Base(int j, double y)
  {
    i = j;
    x = y;
  }

  // constructeur par copie
  Base(const Base& var)
  {
    i = var.i;
    x = var.x;
  }

  // un seul destructeur possible
  ~Base()
  {
    // un destructeur sert a liberer la memoire utilisee par l'objet
    // ici pas de tableau a detruire
  }

};

class Daughter : public Base
{
  double y;

  public :
  // on appelle le constructeur de la classe mere
  // et les constructeurs des attributs
  Daughter(double z) : Base(), y(z) {}

  Daughter(int j, double w, double z) : Base(j, w)
  {
    y = z;
  }

  // pour eviter les conversions implicites, mot cle explicit
  explicit Daughter(int j) : Base()
  {
    y = 0.0;
  }

  // pas besoin de declarer le destructeur
  // le destructeur par defaut est implicite et ne fait rien

};

int main()
{
```

```
  // des qu'on declare un objet, un constructeur sera appele
  // si pas d'argument comme ici, le constructeur par defaut Base() est appele
  Base b;

  // appel du second constructeur de Base
  Base a(4, 5.0);

  // appel du constructeur par copie (equivalent a initialiser c = a)
  Base c(a);

  // on peut aussi appeler un constructeur apres la declaration
  b = Base(6, -2.0);

  Daughter z; // erreur a la compilation
  // car le constructeur par defaut n'a pas ete defini

  Daughter w(2, 5.23, 0.43); // ok

  // lors de l'accolade fermante, les destructeurs seront appeles
  return 0;
}
```

9.5. Pointeur this

C'est un pointeur vers l'objet instancié.

```
class Base;

void affiche_objet(const Base& v);

class Base
{
  public :
  int ip;

  void ModifieIp(int j)
  {
    ip = j;

    // si on veut appeler une fonction qui prend en argument
    // l'objet courant, on utilise this
    affiche_objet(*this);
  }

  void SetIp(int j)
  {
    ip = j;
  }

};

void affiche_objet(const Base& v)
{
  cout << "i = " << v.ip << endl;
```

```
}

int main()
{
  Base var;

  var.ModifieIp(4);
  // il aurait ete equivalent de faire
  // var.SetIp(4); affiche_objet(var);

  return 0;
}
```

9.6. Attributs et méthodes statiques

Les attributs statiques sont partagés par toutes les instances d'une classe. Les méthodes statiques ne doivent pas modifier d'attribut dynamique, ni appeler des méthodes dynamiques. Les méthodes statiques sont complètement équivalentes à des fonctions usuelles (extérieures à la classe) excepté qu'elles sont encapsulées dans une classe.

```
class Exemple
{
  public :
  // attributs dynamiques
  int ip;
  double x;

  // attributs statiques precedes par le mot cle static
  static int jp;
  static double y;

  // methodes dynamiques
  void SetIp(int j)
  {
    ip = j;
  }

  // methodes statiques precedees par le mot cle static
  static void DisplayInfoClass()
  {
    cout << "Cette classe ne sert a rien" << endl;

    ip = 0; // interdit, erreur de compilation
    SetIp(3); // interdit, erreur de compilation

    jp = 3; // oui on a le droit
  }

};

// declaration obligatoire des attributs statiques a l'exterieur de la classe
int Exemple::jp(3);
double Exemple::y(2.03);
```

```
int main()
{
  // pour modifier un membre statique
  Exemple::jp = -10;
  Exemple::y = -0.45;

  // pour appeler une fonction statique
  Exemple::DisplayInfoClass();

  return 0;
}
```

9.7. Méthodes virtuelles, polymorphisme

Les méthodes virtuelles permettent de factoriser du code en écrivant des fonctions générales dans la classe mère, qui appelle des méthodes définies dans les classes filles.

```
class Forme
{
  public :
  // pour une classe abstraite le destructeur doit etre virtuel
  virtual ~Forme() {}

  // on peut definir des fonctions virtuelles
  // qui seront surchargees dans les classes derivees
  virtual void Draw()
  {
    // ne fait rien, on sait pas comment dessiner une forme generale
    cout << "Methode indefinie" << endl;
  }

  // on peut declarer des methodes virtuelles pures
  // qu'on a pas besoin de detailler, car on considere
  // qu'elle ne seront jamais appelees
  virtual int GetNombrePoints() = 0;


  // dans une methode generique, on peut utiliser les methodes virtuelles
  // qui sont specialisees pour chaque forme
  void AnalyseDessin()
  {
    // ici la fonction Draw appelee n'est pas forcement la fonction Draw()
    // definie dans Forme, cela peut etre une fonction d'une classe derivee
    Draw();

    int n = GetNombrePoints();
    cout << " Nombre de points de la forme = " << n << endl;
  }

};

class Carre : public Forme
```

```
{
  public :
  // on surcharge les fonctions virtuelles de Forme
  int GetNombrePoints()
  {
    return 4;
  }

  void Draw()
  {
    cout << "---" << endl;
    cout << "| |" << endl;
    cout << "---" << endl;
  }

};

class Triangle : public Forme
{
  public :
  // on surcharge les fonctions virtuelles de Forme
  int GetNombrePoints()
  {
    return 3;
  }

  void Draw()
  {
    cout << "/|" << endl;
    cout << "--" << endl;
  }

};

int main()
{
  Forme b; // erreur de compilation car la classe est abstraite
  // (abstraite = contient une methode virtuelle pure)

  Carre quad;

  // les fonctions Draw() et GetNombrePoints() du Carre seront appelees
  quad.AnalyseDessin();

  Triangle tri;

  // les fonctions Draw() et GetNombrePoints() du Triangle seront appelees
  tri.AnalyseDessin();

  // on peut aussi creer un objet polymorphique
  // (qui pourra etre un triangle ou un carre)
  Forme* forme;
```

```
  // ici c'est un carre
  forme = new Carre();

  delete forme;

  // et maintenant un triangle
  forme = new Triangle();
}
```

9.8. Surcharge des opérateurs

```
// par exemple si on veut pouvoir faire des operations
// directement sur les vecteurs
class Vector
{
  protected :
  // donnees
  double* data;
  // taille vecteur
  int m;

  public :

  // on surcharge dans la classe tous les operateurs internes

  // operateur =
  Vector& operator=(const Vector& v)
  {
    for (int i = 0; i < m; i++)
      data[i] = v(i);

    return *this;
  }

  // operateur d'acces ()
  double operator()(int i) const;

  // ici c'est l'operateur d'acces qui peut etre utilise pour V(i) = cte;
  double& operator()(int i);

  // l'operateur d'acces () peut etre surcharge
  // pour un nombre quelconque d'arguments (V(i, j) = quelque chose)
  double& operator()(int i, int j);

  // operateurs += *= -=
  Vector & operator +=(const Vector& v);
  Vector & operator *=(const Vector& v);
  Vector & operator -=(const Vector& v);

  // operateur de pre-increment ++v
  Vector & operator ++(void);
```

```
  // operateur de post-increment v++
  Vector operator ++(int);

  // operateurs de comparaison
  bool operator ==(const Vector&) const;
  bool operator !=(const Vector&) const;
  bool operator <=(const Vector&) const;
  bool operator >=(const Vector&) const;
  bool operator <(const Vector&) const;
  bool operator >(const Vector&) const;
};

// les operateurs externes sont definis a l'exterieur
Vector operator +(const Vector&, const Vector&);
Vector operator -(const Vector&, const Vector&);
Vector operator *(const Vector&, const Vector&);

// on peut gerer le cas mixte Vector/double
Vector operator *(const double&, const Vector&);
Vector operator *(const Vector&, const double&);

// ensuite dans le main, on peut utiliser ces surcharges
int main()
{
  Vector x, y, z;

  z = 2.0*x + y;
  ++x;

  x(0) = 3.0;
  // ...

  return 0;
}
```

10. Structure d'un code

En dispose les déclarations des fonctions et classes (appelées prototypes) dans des fichiers d'en-tête (headers) dont l'extension se termine par .hpp, .H, .h++ ou .hxx. Les définitions sont elles placées dans des fichiers .cpp, .C, .cxx, .c++ ou .cc.

10.1. Prototype de fonctions, de classes

Un prototype ne contient que la déclaration de la classe ou de la fonction

```
// exemple de prototype d'une fonction
// la fonction est definie dans un autre fichier
void MaFunction(const double&, int, double*);

// parfois deux classes sont entrelacees, il est alors utile
// de declarer l'existence d'une classe, avant de la detailler plus loin :
class MaClasse;
```

```
// on peut alors declarer des prototypes de fonctions
// avec MaClasse en argument
void FunctionClasse(const MaClasse &);

// exemple de prototype d'une classe
class MaClasse
{
  public :
  int n;
  double x;

  // on ne met que les prototypes des fonctions membres
  void Fonc1(double);
  int Fonc2(int);

};

// Pour definir les fonctions membres Fonc1 et Fonc2
// a l'exterieur de la classe :
void MaClasse::Fonc1(double y)
{
  x = 2.0*y;
}

int MaClasse::Fonc2(int m)
{
  return m + n;
}
```

10.2. Directives du préprocesseur

Les directives du préprocesseur commencent toutes par #.

```
// La directive la plus utilisee est la directive include
// qui inclut de maniere directe le contenu d'un fichier
#include <iostream>

// le fichier iostream.h sera directement insere dans le fichier

// pour un fichier personnel, on met des guillements
#include "MonFichier.hxx"

// pour eviter les multiples inclusions, les fichiers comportent
// toujours des lignes de ce type :
#ifndef FILE_MON_FICHIER_HEADER_HXX
#define FILE_MON_FICHIER_HEADER_HXX
// l'identificateur ici doit etre unique parmi tous les fichiers inclus
// le fait de mettre ce define empeche une seconde inclusion du meme fichier
// puisque FILE_MON_FICHIER_HEADER_HXX sera alors defini

// on met ici tout le contenu du fichier
class Patati
{};
```

```
// et on termine par endif
#endif

// la notion de flags est souvent utilisee, ce sont
// des indicateurs qu'on peut definir lors de la compilation
// (option -Dnom_flag)
// Par exemple si on veut specifier dans le code que
// la librairie Blas est utilisee, on va compiler :
// g++ -DUSE_BLAS toto.cc -lblas

// et dans le fichier toto.cc, on va traiter differemment le cas
// ou on peut utiliser Blas, et le cas ou on ne peut pas
#ifdef USE_BLAS
// partie du code qui ne sera activee que si le flag USE_BLAS est defini
// (soit lors de la compilation soit par une directive #define)
// par exemple on utilise une routine blas pour additionner deux vecteurs
daxpy_(n, alpha, x, 1, y, 1);

#else

// dans ce cas on a pas blas, on fait une somme standard
for (int i = 0; i < n; i++)
  y[i] += alpha*x[i];

#endif

// pour activer manuellement un flag
#define USE_BLAS

// pour desactiver le flag :
#undef USE_BLAS

// on peut tester aussi que le flag est non actif
#ifndef USE_BLAS
// dans ce cas, on a pas Blas

#endif


// les #define permettent aussi de definir des macros
// par exemple la macro DISP
// permet d'afficher le nom et contenu d'une variable :
#define DISP(x) std::cout << #x ": " << x << std::endl

// on peut alors utiliser DISP partout, et le remplacement sera effectue
// par le preprocesseur
DISP(x); // sera remplace par std::cout << " x: " << x << std::endl;
```

10.3. Espace de nommages (namespace)

Lorsqu'on écrit une librairie, toutes les fonctions et classes de la librairie sont encapsulées dans un espace de nommage. L'espace de nommage de la STL est std.

```
// avant d'ecrire toute fonction de la librairie, on met le mot cle namespace
namespace NomLibrairie
{

  int Fonc1(double y)
  {
    return int(y);
  };

  class MaClasse
  {
    public :
    double x;
  };

} // fin du namespace

// ensuite si on veut utiliser une fonction du namespace,
// soit on fait NomLibrairie::

void f()
{
  NomLibrairie::Fonc1(2.0);
}

// soit on utilise la commande using, et on a pas besoin de
// mettre NomLibrairie::
using namespace NomLibrairie;

int main()
{
  Fonc1(2.0);

  return 0;
}
```

11. Entrées-sorties

Exemple :

```
// iostream pour les entrees sorties standard
#include <iostream>

// fstream pour les fichiers
#include <fstream>

using namespace std;
```

```
int main()
{
  // cout pour le flux de sortie a l'ecran
  // cerr pour le flux des erreurs
  // endl est un retour chariot avec vidage du buffer
  cout << "Coucou " << endl;

  // cin pour le flux d'entree au clavier
  int test_input; double x;
  cout << "Entrez un entier suivi d un flottant" << endl;
  cin >> test_input >> x;

  // par defaut, cout affiche 5 chiffres significatifs
  // on peut en mettre plus avec precision
  cout.precision(15); // ici 15 chiffres significatifs
  cout << "x = " << x << endl;

  // pour ecrire dans un fichier : ofstream
  ofstream file_out("sortie.dat"); // on ecrit dans sortie.dat

  // la syntaxe est la meme que pour cout
  file_out << "x = " << x << endl;

  // une fois les ecritures terminees :
  file_out.close();

  // pour lire dans un fichier ifstream
  ifstream file_in("entree.dat"); // on lit dans entree.dat

  // premiere chose a faire : verifier que le fichier est ouvert
  if (!file_in.is_open())
    {
      cout << "Impossible de lire le fichier entree.dat" << endl;
      abort(); // abort quitte le programme, pratique pour debugguer
    }

  // ensuite on peut lire une donnee
  file_in >> x;

  // on peut verifier si la lecture de cette donnee a marche
  if (!file_in.good())
    {
      // un echec peut venir d'une mauvaise donnee
      // (par exemple si il y a un mot au lieu d'un nombre)
      cout << "Echec de lecture" << endl;
      abort();
    }

  // on ferme le flux une fois termine
  file_in.close();

  // on peut ouvrir un autre fichier avec la methode open
```

```
  file_in.open("autre.dat");
  file_in.close();

  // pour lire/ecrire en binaire on utilise write et read
  file_out.open("binary.dat");
  // write demande des char, donc on convertit tout en char*
  file_out.write(reinterpret_cast<char*>(&x), sizeof(double));
  file_out.close();

  file_in.open("binary.dat");
  // read demande des char, donc on convertit tout en char*
  file_in.read(reinterpret_cast<char*>(&x), sizeof(double));
  file_out.close();

  return 0;
}
```

12. Fonctions et classes génériques

Exemple de fonctions génériques :

```
// par exemple, si l'on veut ecrire une fonction
// qui marche a la fois pour les complexes et les reels
// plutot que de dupliquer le code, on va ecrire
// une fonction prenant en parametre un type generique,
// par exemple pour la norme d'un vecteur :
template<class Vector>
double Norm(const Vector& x)
{
  double norm = 0.0;
  for (unsigned int i = 0; i < x.size(); i++)
    norm += abs(x[i]);

  return norm;
}

// on peut avoir plusieurs types generiques
template<class Matrix, class Vector1, class Vector2>
void MltAdd(const Matrix& A, const Vector1& x, Vector2& y)
{
  // ...
}

int main()
{
  // les fonctions s'utilisent normalement
  vector<double> x;
  cout << Norm(x) << endl;

  // pour les vecteurs complexes
  vector<complex<double> > xc;
  cout << Norm(xc) << endl;
```

```
}
```

Exemple de classes génériques

```
// exemple d'une classe vecteur
// T peut etre un int, double, char, string, etc
template<class T>
class Vector
{
  protected:
  // donnees
  T* data;
  int m;

  public:
  int GetSize()
  {
    return m;
  }

};

// on peut mettre plusieurs parametres
template<class T, class Prop>
class Base
{
  public :
  // declarer des fonctions normales
  void FoncNormale(double, double);

  // et des fonctions template
  template<class Vector>
  void FoncGeneric(double, Vector&);
};

// definition des fonctions a l'exterieur de la classe
template<class T, class Prop>
void Base<T, Prop>::FoncNormale(double x, double y)
{
}

// pour la fonction template, on a en premier les parametres
// template de la classe puis ceux de la fonction
template<class T, class Prop> template<class Vector>
void Base<T, Prop>::FoncGeneric(double x, Vector& y)
{
}

// on peut ensuite faire une specialisation partielle de la classe
// la classe ainsi definie remplacera completement la classe
// generique pour le jeu de parametres choisi
template<class Prop>
class Base<int, Prop>
```

```
{
  public :
};

// on peut faire une specialisation totale
template<>
class Base<int, double>
{
};


int main()
{
  // lors de l'instanciation, il faut preciser le type des parametres
  Vector<int> v;

  // ici le compilateur choisira la classe Base generique
  Base<string, int> A;

  // ici le compilateur choisira la classe Base specialisee partiellement
  Base<int, string> B;

  // ici le compilateur choisira la classe specialisee totalement
  Base<int, double> C;
}
```

Les paramètres template peuvent être introduits par le mot-clé typename (au lieu de class), et on peut aussi utiliser des constantes dont on connaitra la valeur à la compilation.

```
// ici m designe la taille du vecteur
// m est connu a la compilation, ce qui permet de pouvoir ecrire
// des algos optimises (meta-programing)
template<typename T, int m>
class TinyVector
{
  // comme m est connu a la compilation, on peut
  // utiliser des tableaux statiques
  T data[m];

  public :
  T operator()(int i)
  {
    return data[i];
  }
};

int main()
{
  // dans le main, on doit preciser la taille
  TinyVector<double, 3> y;

}
```

13. Exceptions en C++

Exemple

```
#include <iostream>
#include <exception>

using namespace std;

// pour chaque nouvelle exception, on fait une classe :
class MonException
{
  public :
  string comment; // commentaire associe a l'exception

  // constructeur prenant un commentaire en argument
  MonException(const string& c) : comment(c) {}

};

int main()
{
  try
    {
      // on met ici le code qu'on veut executer
      // si a un moment on a une erreur, on peut lancer une exception
      MonException a(``Argument invalide'');
      throw a;

      // une exception peut etre de n'importe quel type
      // par exemple un entier
      int b = 3;
      throw b;
    }
  catch (MonException& err)
    {
      // cas ou une exception de type MonException a ete lancee
      // on fait le traitement associee a cette excpetion
    }
  catch (int a)
    {
      // cas ou une exception de type entier a ete lancee
    }
  catch (...)
    {
      // toutes les autres exceptions
      // on peut appeler abort par exemple si on ne sait pas gerer
      // ce cas la
      abort();
    }

  return 0;
}
```

14. Utilisation de la STL

14.1. Fonctions mathématiques

Les fonctions mathématiques sont contenues dans cmath.
Exemple

```
#include <cmath>

using namespace std;

int main()
{
  // calcul de x^y
  double x = 2.3, y = 0.8;
  double z = pow(x, y);

  // sqrt : racine carree
  // exp : exponentielle
  // log : logarithme neperien
  // log10 : logarithme en base 10
  // log2 : logarithme en base 2
  // sin, cos, tan : fonctions trigos usuelles
  // asin, acos, atan : les arc cos, sin et tan
  // sinh, cosh, tanh, asinh, acos, atanh : les cos, sin, etc hyperboliques
  // tgamma : fonction gamma
  z = asin(x);

  // arrondi au plus proche
  // ici on convertit en entier parce que round renvoie un reel
  int n = int(round(z));

  // arrondi inferieur
  n = int(floor(z));

  // arrondi superieur
  n = int(ceil(z));

  // d'autres fonctions existent
}
```

14.2. Chaînes de caractères (string)

Exemple

```
#include<string>
#include<iostream>
#include<sstream>

using namespace std;

// pour convertir un nombre en string
template<typename T>
inline std::string to_str(const T& input)
```

```
{
  std::ostringstream output;
  output << input;
  return output.str();
}


// pour convertir un string en nombre
template <class T>
inline T to_num(std::string s)
{
  T num;
  std::istringstream str(s);
  str >> num;
  return num;
}

int main()
{
  // pour manipuler les chaines de caracteres, ne pas utiliser char* (C)
  // mais plutot des string
  string s("toto");

  // nombre de caracteres : size()
  cout << "nombre de lettres = " << s.size() << endl;

  // pour acceder a chaque caractere, operateur []
  s[1] = 'y'; // s vaut maintenant tyto

  // on peut comparer des string
  if (s == "tata")
    cout << "incorrect" << endl;

  // on peut concatener avec l'operateur +
  s = s + ".dat"; // s vaut maintenant tyto.dat

  // on peut rechercher une chaine
  int pos = s.find("to"); // pos devrait valoir 2
  if (pos == string::npos)
    cout << "on n a pas trouve la chaine to dans s" << endl;

  // on peut extraire une sous-chaine
  string s_sub = s.substr(pos, 3);
  // pour extraire 3 caracteres a partir de s[pos]

  s_sub = s.substr(pos);
  // pour extraire tous les caracteres a partir de s[pos]

  // par exemple pour convertir i = 356 en string
  int i = 356;
  s = to_str(i);
```

```
  // et pour convertir "3.1456" en double
  s = string("3.1456");
  double x = to_num<double>(s);
}
```

14.3. Vecteurs

```
#include<vector>

using namespace std;

int main()
{
  // la classe vector stocke les elements de maniere contigue
  // acces rapide, mais insertion lente
  vector<double> x(5); // vecteur de 5 reels

  // acces avec []
  x[0] = 1.06e-7;
  x[1] = -3.5e8;

  // on peut agrandir le vecteur en gardant les anciens elements
  x.resize(6);

  // les nouveaux elements doivent etre initialises
  x[5] = 2.5;

  // on peut aussi rajouter un element a la fin du vecteur
  // (ie le vecteur est retaille et un element est rajoute)
  x.push_back(3.4); // equivalent a x.resize(7); x[6] = 3.4;

  // on peut aussi inserer un element avec insert, et detruire le vecteur avec clear
  x.clear();
}
```

14.4. Listes

```
#include<list>
#include <iostream>

using namespace std;

int main()
{
  // une liste ne stocke pas les elements de maniere contigue
  // (pensez a une liste simplement chainee)
  // l'insertion est rapide, mais l'acces est lent
  list<double> v;

  // push_back et insert existent
  v.push_back(3.4);
  v.push_back(0.3);
```

```
  // pour parcourir la liste, le mieux est d'utiliser un iterateur
  list<double>::iterator it;
  cout << ``v = ``;
  for (it = v.begin(); it != v.end(); ++it)
    cout << `` `` << *it;

  cout << endl;
}
```

14.5. Piles

```
#include<stack>

using namespace std;

int main()
{
  // stack est un conteneur correspondant a une pile
  // (premier entre, premier sorti)
  stack<double> pile;

  // on empile les elements avec push
  pile.push(0.33);
  pile.push(-0.25);
  pile.push(1.3); // la pile vaut (1.3, -0.25, 0.33)

  // on libere un element avec pop
  pile.pop(); // la pile vaut (-0.25, 0.33)

  // pour acceder au dernier element top
  double y = pile.top(); // y = -0.25 et la pile vaut (-0.25, 0.33)
}
```