Langage C++

Polytech Angers Bertrand Cottenceau

Table des matières

1. Fiches de synthèse

Les premières pages de ce document fournissent des fiches de synthèse sur des éléments de syntaxe du langage C++. Le langage C++ est une extension du langage C avec la notion de classes. Il reprend donc la syntaxe du C pour les instructions et ajoute des mots-clé et des éléments de syntaxe pour les classes, la dérivation, les paramètres de type et la ligature dynamique de méthode.

Enfin, le langage C++ est fourni avec une librairie de classes appelée STL dont proviennent les classes `string`, `vector<T>` (tableaux) et `list<T>` (listes chaînées) .

En connaissant le langage C (les pointeurs sont utilisés en C++) et un langage objet comme C# ou Java, cette synthèse devrait permettre de rentrer au plus vite dans le sujet.

1.1. Le C++ en dehors des classes

Les références sont des ***alias*** : plusieurs noms pour une même variable, un même objet.

Déclaration d'une référence

```
int varI=3;
int & refI=varI;    //  refI est un autre nom pour la variable varI
```

Un alias fournit la même adresse que la variable

```
int * p1=&refI;     // refI est une référence à varI
int * p2=&varI;
cout << p1 << p2 << endl; // les deux pointeurs contiennent la même adresse
```

⚠ **Ne pas confondre les deux sens de l'opérateur &** : adresse de ≠ déclaration de référence

```
int & refI=varI;    // déclaration d'une référence
int * p=&varI;      // opérateur & qui fournit l'adresse d'une variable
```

Paramètre passé "par référence"

```
void reset(int & pFormel) /// pFormel = [sortie]
{
 pFormel=0;  // pFormel est un alias d'une variable du programme appelant
}
...
reset(pAppel);      // après la fonction, pAppel vaut 0
```

Paramètre passé "par référence constante"

```
void f(const GrosType & pEntree) /// pEntree =[entrée]
{   // si le paramètre est gros, et seulement une entrée, privilégier
    // le passage par référence constant, c'est plus rapide
}
```

Les paramètres de fonction en C++ (synthèse)

Un *petit paramètre* d'entrée (char, int, double, etc.) peut être *passé par valeur* (ou *par valeur constante.*)
Un *gros paramètre* d'entrée (objet,tructure) doit être passé *par référence* constante (↗performance).
Un paramètre de *sortie* doit être passé *par référence* (remplace le passage par adresse du langage C)
Pour les tableaux : fournir l'adresse de la première case et la taille (comme en C).

```
void f(int * tab, unsigned taille){ ...}
```

Fonction qui retourne une référence (ex: operator[ ])

```
int & f(int pF){ ... }  // f retourne une référence à une variable int
```
⚠ *seulement pour retourner une référence à une variable non locale à f*

```
...
f(3)=2;       // f(3) est modifiable, puisque c'est un alias d'une variable!
```

Surcharge de fonctions Plusieurs fonctions peuvent avoir le même nom mais pas la même signature

Arguments avec valeurs par défaut : valeur attribuée quand un argument explicite fait défaut

```
void f(int pI, double d=2.1);    // f(3) ≈ f(3,2.1)

void g(char *, int i=4);         // g("toto") ≈ g("toto",4)
```

1.2. Allocation Automatique vs Dynamique

Allocation: réserver de la mémoire pour une variable/un objet (ou un tableau de variables/objets)

Deux modes en C++: allocation ***automatique*** / allocation ***dynamique*** (opérateurs new/delete)

Portée de l'allocation automatique : jusqu'à la fin de bloc de déclaration de la variable
Portée de l'allocation dynamique : de {**`ptr = new T}`** à **`{delete ptr}`** avec `ptr` un pointeur `T*`

```
#include <iostream>
#include<vector>
using namespace std;

int main()
{   int A1=12;      //allocation auto de A1 – désalloué en fin de bloc
    int * ptr=NULL; //allocation auto de ptr (pointeur)

      // sous-bloc créé artificiellement pour illustrer la portée
     {
           cout << "--Sous-bloc--" << endl;
           vector<int> v={4,7,3};  // alloc auto de v (taille 3)
           int A2=27;             // alloc auto de A2
           ptr= new int(7);       // alloc dyn de *ptr et init *ptr=7

           for(int x:v) cout << x << ",";  //foreach

           cout << "*ptr=" << *ptr << endl;

      } // fin de bloc => désallocation de A2 et v

   // Les variables v et A2 n'existent plus ici
   // La variable *ptr allouée dynamiquement existe toujours

    cout << "--Bloc main--"<<endl;
    cout << "A1=" << A1 << " *ptr=" << *ptr << endl;
    delete ptr;    // désallocation de l'entier *ptr

  // allocation dynamique d'un tableau
    int * tab = new int[3]{4,5,6};
    for(int i=0;i<3;i++)
        cout << tab[i] << " ";

    delete[] tab; //désallocation du tableau

    return 0;
}
//fin de bloc main => désallocation de A1 et ptr
```

Sortie

```
--Sous-bloc--
4,7,3,*ptr=7
--Bloc main--
A1=12 *ptr=7
4 5 6
```

1.3. Les classes en C++

```
#pragma once        //MaClasse.h [HEADER]
class MaClasse
{
public:
      MaClasse();                // constructeur sans argument [default constructor]
      MaClasse(int var);               // constructeur à un argument de type int
      MaClasse(const MaClasse & obj);  // constructeur de copie
      ~MaClasse();                     // destructeur
      MaClasse & operator=(const MaClasse & obj);   // opérateur d'affectation

      void SetVal(int val);            // Modificateur => méthode pas constante
      int GetVal() const;              // Accesseur => méthode constante

private:
      int _val;
      double _autre;
};
```

Source des méthodes dans un fichier séparé (fichier d'extension .cpp)

```
#include "MaClasse.h"     //MaClasse.cpp

MaClasse::MaClasse()  // [default constructor]
{
      _val=0; _autre = 2.1;
}

MaClasse::MaClasse(const MaClasse & obj) //[copy constructor]
{     // initialisation comme copie de obj
      ...
}

  // méthode constante: const en fin de signature
int MaClasse::GetVal() const  {  return _val; }

void MaClasse::SetVal(int val){  _val=val;    }

...
```

A noter

- un constructeur ou un destructeur ne retourne rien (ne pas mettre void)
- un *constructeur* sert à ***initialiser un objet***
- un *modificateur* est une méthode qui *change* l'état de l'objet courant (pas constante)
- un *accesseur* est une méthode qui *consulte/donne* l'état de l'objet courant (méthode constante).

Utilisation du type `MaClasse`

```
#include "MaClasse.h"
#include<iostream>      // pour utiliser std::ostream (cout)
using namespace std;   // pour simplifier std::cout/std::endl

void main()
{
      MaClasse objet1;    // alloc auto et init. avec default constr.
                          // etat = (objet1._val==0, objet1._autre==2.1)

      MaClasse objet2(15); // alloc auto et init. Avec constr. à 1 argument int

      MaClasse objet3(objet1); // alloc auto et init. avec copy constr.

      cout << objet1.GetVal() << endl;

      objet1.SetVal(12);  // nouvel état : (objet1._val==12,objet1._autre==2.1)
}
      // Désallocation automatique de objet1,objet2,objet3
      // → destructeur appelé pour chaque objet juste avant sa désallocation
```

1.4. Les opérateurs

En C++, certains opérateurs peuvent être programmés. Ci-dessous l'exemple d'une classe de matrices 2×2. La somme (+) et le produit (*) de matrices sont programmés, ainsi que l'accès aux valeurs de la matrice avec l'opérateur fonction.

```
#pragma once        //HEADER Mat2x2.h
#include <iostream>

class Mat2x2
{ public:
    Mat2x2(double vInit=0);
    double operator()(int l,int c) const;
    double & operator()(int l,int c);
    Mat2x2 operator+(const Mat2x2 & M) const;
  private:
      double _mat[4]; // stocke les 4 valeurs
}; // < - - fin de classe
            /// fonctions globales [hors classe] ///
     Mat2x2 operator*(const Mat2x2 & Ml,const Mat2x2 & Mr);
     Mat2x2 operator*(double sc,const Mat2x2 & Mr);
     std::ostream & operator<<(std::ostream&,const Mat2x2 &);
```

```
#include "Mat2x2.h"        //SOURCE Mat2x2.cpp
using namespace std;

  Mat2x2::Mat2x2(double vI):_mat({vI,vI,vI,vI}){}

 //--- opérateurs méthodes de Mat2x2 -------------
 double Mat2x2::operator()(int l,int c) const {
    if(l<2 && c<2 && l>=0 && c>=0) return _mat[l*2+c];
    else throw "indices invalides";
  }

 double & Mat2x2::operator()(int l,int c){
    if(l<2 && c<2 && l>=0 && c>=0) return _mat[l*2+c];
    else throw "indices invalides";
  }

  Mat2x2 Mat2x2::operator+(const Mat2x2 & M) const{
    Mat2x2 resultat(*this);
    for(int i=0;i<4;i++) resultat._mat[i]+=M._mat[i];
    return resultat;
  }

 //--- opérateurs Hors classe (fonctions globales)-----
  Mat2x2 operator*(const Mat2x2 & Ml,const Mat2x2 & Mr){
    Mat2x2 resultat;
    resultat(0,0)=Ml(0,0)*Mr(0,0) + Ml(0,1)*Mr(1,0);
    resultat(0,1)=Ml(0,0)*Mr(0,1) + Ml(0,1)*Mr(1,1);
    resultat(1,0)=Ml(1,0)*Mr(0,0) + Ml(1,1)*Mr(1,0);
    resultat(1,1)=Ml(1,0)*Mr(0,1) + Ml(1,1)*Mr(1,1);
    return resultat;
  }

  Mat2x2 operator*(double sc,const Mat2x2 & Mr){
    Mat2x2 resultat(Mr);
    for(int i=0;i<4;i++) resultat(i/2,i%2)*=sc;
    return resultat;
  }

  ostream & operator<<(ostream & flot,const Mat2x2 & M){
    flot << "[" << M(0,0) <<","<<M(0,1) <<";";
    flot << M(1,0) <<","<<M(1,1) <<"]";
    return flot;
  }
```

```
... //utilisation de Mat2x2

int main(){
  Mat2x2 M1(3);
  cout << M1 << endl;
  M1(0,0)=1;
  M1(0,1)=2;
  M1(1,1)=4;
  cout << M1 << endl;
  cout << M1*M1 << endl;
  cout << M1+M1 << endl;
  cout << 3*M1 << endl;
  return 0;
}
```

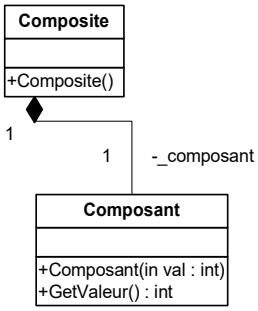
Sortie console:

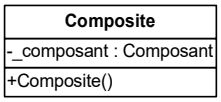
```
[3,3;3,3]
[1,2;3,4]
[7,10;15,22]
[2,4;6,8]
[3,6;9,12]
```

Méthode ou Hors classe ? Pour certains opérateurs, on peut choisir de programmer l'opérateur comme méthode (membre de la classe) ou comme fonction globale (hors classe). Dans l'exemple ci-dessus, l'opérateur + est membre de `Mat2x2` alors que les opérateurs * sont hors classe (remarquer la différence de signature). Certains opérateurs, comme `operator=` et `operator[]`, sont nécessairement membres de la classe.

1.5. La composition en C++ (1/2)

Composition: quand un objet est composé de 1 ou plusieurs objets d'autres classes. En UML, on représente cela avec un losange côté composite.





✍ ***Différence avec C# ou Java*** : dans l'exemple ci-dessous, un objet membre de type `Composant` est nécessairement créé à chaque fois qu'un objet `Composite` est créé, c'est automatique. C'est très différent de Java ou C#.

```
// composite.h
#pragma once

#include "composant.h"

class Composite{
public:
      Composite();
      Composant GetComposant() const;
private:
      Composant _composant; // objet membre
};
```

```
//composite.cpp
#include "composite.h"

            // liste d'initialisation
Composite::Composite():_composant(12){}

Composant Composite::GetComposant() const{
      return _composant;
}
```

✍ ***Liste d'initialisation*** **:** indique comment l'objet `_composant` doit être initialisé (quel constructeur choisir)

Autre exemple: un objet `Point` décrit des coordonnées qui sont stockées dans un objet `std::vector<int>`

```
#pragma once        // point.h
#include<vector>

class Point
{ public:
       Point(int x=0,int y=0);
       Point(const Point & p);
       int GetX() const;
       int GetY() const;
  private:
      std::vector<int> _coord;
};
```

```
#include "Point.h"        // point.cpp
using namespace std;

Point::Point(int X,int Y):_coord(2)
{
  //_coord initialisé avec taille=2
  _coord[0]=X;
  _coord[1]=Y;
}

Point::Point(const Point & p):_coord(p._coord){
 // _coord initialisé par copie de p._coord
 // That's all folks!
}

int Point::GetX() const{ return _coord[0];}
int Point::GetY() const{ return _coord[1];}
```

✍ ***Liste d'initialisation*** **:** un objet `Point` est constitué d'un objet `std::vector<int>`. Pour chaque constructeur de `Point`, la liste d'initialisation indique comment l'objet `_coord` doit être initialisé (voir fiche `vector<T>`). Ne pas utiliser la liste d'initialisation rend certains problèmes inextricables ou mal résolus.

Un bug ... 💎

```
// Point.cpp
#include "Point.h"

Point::Point(int X,int Y){
  _coord[0]=X;
  _coord[1]=Y;
}
/* sans indication, _coord est
initialisé de taille 0, donc
 _coord[0] n'existe pas ! */
```

... mal corrigé 💎

```
// Point.cpp
#include "Point.h"

Point::Point(int X,int Y){
  _coord.resize(2); //patch
  _coord[0]=X;
  _coord[1]=Y;
}
/* c'est plus performant avec la liste
d'initialisation    */
```

1.6. La composition en C++ (2/2)

Données en profondeur : quand un objet alloue dynamiquement de la mémoire pour stocker des données. La mémoire allouée dynamiquement doit faire l'objet de la plus grande attention. On peut par exemple reprendre l'exemple de la page précédente où le composant ***vecteur est alloué dynamiquement.***

Ressemblance avec C# ou Java : dans cet exemple, ça ressemble à ce que l'on fait en C# ou en Java : le constructeur d'un `Point` doit allouer (dynamiquement) un objet vecteur. Mais la ressemblance s'arrête là puisqu'en C++ cette mémoire ne sera pas gérée par un Garbage Collector : ***il faut prévoir sa désallocation***.

💣 cette technique, bien que formatrice, est néanmoins à décourager pour les néophytes. Il y a souvent des alternatives, par exemple l'implémentation de la section 1.5 est plus simple et plus robuste.

```
#pragma once        //HEADER Point.h
#include<vector>

class Point{
 public:
  Point(int x=0,int y=0);
  Point(const Point & p);
  Point & operator=(const Point & p);
  ~Point();

  int GetX() const;
  int GetY() const;
 private:
  std::vector<int> * _pCoord;
};
```

Remarque: en terme d'utilisation, rien ne distingue cette classe de celle de la section 1.5. Seul le choix d'implémentation diffère.

```
#include <iostream>
#include "Point.h"
using namespace std;

  // utilisation de la classe Point
int main()
{
    Point p1(10,10),p2(50,50);
    cout << p1.GetX() << endl;

    p1=p2;  // utilise op=
    cout << p1.GetX() << endl;

    Point p3(p2); // crée une copie
    cout << p3.GetY() << endl;
    return 0;
}
```

```
#include "Point.h"
using namespace std;

Point::Point(int X,int Y)
{
   _pCoord = new vector<int>(2);
   (*_pCoord)[0]=X;
   (*_pCoord)[1]=Y;
}

Point::Point(const Point & p)
{
  _pCoord = new vector<int>(*p._pCoord);
}

Point::~Point()
{
   delete _pCoord;  //désalloc.
}

Point & Point::operator=(const Point & p)
{
 (*_pCoord) = (*p._pCoord); // [OK]
            //_pCoord=p._pCoord; <-[KO]
  return *this;
}

int Point::GetX() const
{      return (*_pCoord)[0];      }

int Point::GetY() const
{      return (*_pCoord)[1];      }
```

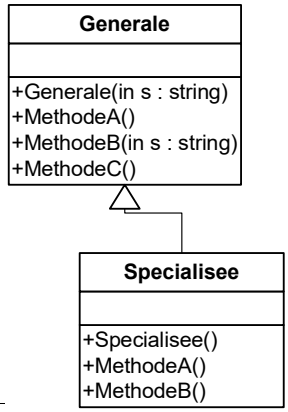
1.7. La dérivation en C++ (1/2)

La dérivation (extension) se représente en UML par une flèche. La flèche part de la sous-classe (ou classe spécialisée/étendue) et désigne la classe de base (la classe la plus générale).

```
#pragma once                // Spécialisee.h

#include "generale.h"

class Specialisee : public Generale
{
public:
      Specialisee();
      void MethodeA();
      void MethodeB();
};
```




```
// Specialisee.cpp
#include "specialisee.h"

                              // liste d'initialisation
Specialisee::Specialisee(): Generale(const std::string & s)
{
}

void Specialisee::MethodeA(){    // redéfinition
  // masque la méthode Generale::MethodeA()
  // ...
}
 ...
```

Liste d'initialisation : le constructeur de `Specialisee` indique comment initialiser la partie héritée: dans ce cas, à l'aide du constructeur à 1 argument de la classe de base `Generale`.

Toutes les méthode publiques de la classe de base sont utilisables sur un objet de la classe dérivée/étendue. Certains phénomènes de *masquage*, en cas de redéfinition, nécessitent néanmoins l'utilisation du nom de la classe de base et de l'opérateur de portée **::**

```
#include "specialisee.h"

using namespace std;

void main()
{
      Specialisee s1;

      s1.MethodeC();  // s1.Generale::MethodeC() (méthode héritée)

      s1.MethodeB();  // s1.Specialisee::MethodeB()

      s1.MethodeB("toto"); // s1.Generale::MethodeB("toto") (méthode héritée)

      s1.MethodeA();      // utilise la redéfinition
            // s1.Specialisee::MethodeA() masque s1.Generale::MethodeA()

      s1.Generale::MethodeA();  // on précise quelle méthode appeler

}
```

1.8. La dérivation en C++ (2/2) : exemple

Ci-dessous une classe `DeNFaces` dérivée/étendue/spécialisée en `De6` (dé à 6 faces).

```
#pragma once  // HEADER DeNFaces.h

class DeNFaces
{
public:
      DeNFaces(unsigned nbFaces);
      unsigned GetNombreDeFaces() const;
      void Lancer();
      unsigned GetValeur() const;
private:
      unsigned _valeur; //entre 1 et _nbFaces
      unsigned _nbFaces; //nb faces du dé
};
```

```
#pragma once // HEADER De6.h
#include "DeNFaces.h"
      //De6 étend DeNFaces
class De6 : public DeNFaces{
public:
    De6();  // default constructor
};
```

```
#include "De6.h"      // De6.cpp

      // liste d'initialisation
De6::De6():DeNFaces(6){ }
```

```
#include "De6.h"
#include<iostream>
using namespace std;
void main(void){
      DeNFaces de1(9);  // dé à 9 faces
      De6 de2;          // dé à 6 faces
      cout << de1.GetNombreDeFaces() << endl;
      cout << de2.GetNombreDeFaces() << endl;  // méthode héritée
      de1.Lancer();
      de2.Lancer();       // méthode héritée
      cout << d1.GetValeur() << endl;
      cout << d2.GetValeur() << endl;  // méthode héritée
}
```

On peut encore dériver/étendre la classe `De6`. On peut par exemple ajouter un attribut de couleur .

```
#pragma once     //De6Couleur.h
#include "De6.h"
      //De6Couleur étend De6
class De6Couleur:public De6{
public:
      De6Couleur(unsigned couleur);
      unsigned GetCouleur() const;
private:
      unsigned _couleur;
};
```

```
#include "De6Couleur.h"    // De6Couleur.cpp

                           //liste d'initial.
De6Couleur::De6Couleur(unsigned c):De6(),
                                   _couleur(c)
{ }

unsigned De6Couleur::GetCouleur() const{
      return _couleur;
}
```

Doit-on rappeler que ne pas utiliser la liste d'initialisation pose parfois problème ?
Le fichier ci-dessous ne compile pas.

```
#include "De6.h"
De6::De6(){ }
```

Pourquoi? Il n'y a pas de [default constructor] dans `DeNFaces`. Il est donc nécessaire de fournir un argument pour initialiser la partie héritée. CQFD

```
#include "De6Couleur.h"
#include<iostream>
using namespace std;

void main(void)
{
 De6Couleur dc(2); //dé 6 faces couleur=2
 cout << dc.GetNombreDeFaces() << endl;
 dc.Lancer();
 cout << dc.GetValeur() << endl;
 cout << dc.GetCouleur() << endl;
}
```

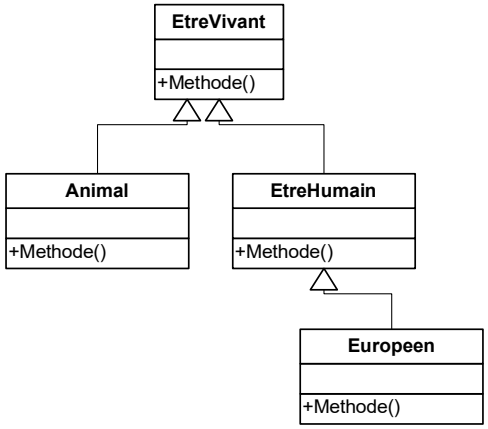
1.9. Le polymorphisme

Lorsqu'une hiérarchie de classes existe, certaines conversions de type sont possibles. D'un point de vue ensembliste, il est normal que les objets d'une sous-classe puissent être aussi considérés comme des objets de la classe de base. Par exemple, la classe des êtres humains étant un sous-ensemble de celle des êtres vivants, un être humain **EST UN** être vivant. La relation de dérivation peut-être assimilée à la relation **EST UN.**

```
class EtreVivant
{
public:
       void Methode();
};

class Animal: public EtreVivant
{
public:
       void Methode();
};

class Europeen: public EtreHumain
{
public:
       void Methode();
};
```



Par défaut, la ligature des méthodes est ***statique***. Cela signifie que via un pointeur de type `EtreHumain *`, on n'accède qu'aux méthodes de la classe `EtreHumain` ou de ses classes de base (par exemple `EtreVivant`).

Ligature statique (par défaut)

```
void main()    // CONVERSION  DERIVEE * -> BASE *
{
       Europeen E;
       Animal A;
       EtreHumain * ptrEH=&E;  // Europeen * -> EtreHumain *

       ptrEH->Methode();      // ptrEH->EtreHumain::Methode(), même si l'objet
                              // pointé est de type Europeen!

       ptrEH->EtreVivant::Methode();  // possible, of course!

       EtreVivant * ptrEV = ptrEH;   // EtreHumain * -> EtreVivant *

       ptrEV->Methode();      // ptrEH->EtreVivant::Methode() même si l'objet
                              // pointé est de type EtreHumain !

       EtreVivant * ptrEV2 = &A;     // Animal * -> EtreVivant *
}
```

Ligature dynamique : méthode virtuelle (mot clé virtual) dans la classe de base.

```
class EtreVivant
{
public:
       virtual void Methode();       // ligature dynamique
       virtual ~EtreVivant();        // le destructeur doit être virtuel
};
-------------------------------------------------------------------------------
void main()  // utilisation
{
       EtreVivant * tableau[3];
       tableau[0]=new EtreVivant;
       tableau[1]=new Animal;
       tableau[2]=new Europeen;

       tableau[0]->Methode();  // méthode appelée  EtreVivant::Methode()
       tableau[1]->Methode();  // méthode appelée  Animal::Methode()
       tableau[2]->Methode();  // méthode appelée  Europeen::Methode()

       for(unsigned i=0;i<3;i++) delete tableau[i];
        //     destructeurs appelés: EtreVivant::~EtreVivant(), Animal::~Animal(),
        //     Europeen::~Europeen(), parce que le destructeur est virtuel
}
```

1.10. Les classes abstraites

Il n'y a pas d'interface en C++. Les classes abstraites compensent l'absence d'interface. Une classe ***abstraite*** est une classe avec au moins ***une méthode virtuelle pure*** (**`virtual...=0`**).

```
#pragma once // Boisson.h
#include<string>
class Boisson  // classe abstraite
{public:
   Boisson(double prix);
   virtual std::string ToString() const;
   double GetPrix() const;

   virtual bool EstChaud() const =0;
   virtual bool EstSucre() const =0;
 private:
      double _prix;
};
```

```
#include "Boisson.h"   // Boisson.cpp
using namespace std;

Boisson::Boisson(double prix):_prix(prix)
{}

string Boisson::ToString() const{
    return "Boisson";}

double  Boisson::GetPrix() const{
    return _prix;}

// PAS DE CODE pour EstChaud()/EstSucre()
```

```
#pragma once    // ChocolatChaud.h
#include "Boisson.h"
       // classe CONCRETE
class ChocolatChaud:public Boisson
{  public:
       ChocolatChaud();
       std::string ToString() const;
       bool EstChaud() const;
       bool EstSucre() const;
};
```

```
#include "ChocolatChaud.h"
using namespace std;

ChocolatChaud::ChocolatChaud():Boisson(3.2){}
string ChocolatChaud::ToString() const{
    return "Chocolat Chaud"; }

// L'implémentation de EstChaud()/EstSucre()
bool ChocolatChaud::EstChaud() const
{       return true;         }
bool ChocolatChaud::EstSucre() const
{       return true;  }
```

```
#pragma once       //Soupe.h
#include "Boisson.h"
       // classe CONCRETE
class Soupe:public Boisson
{ public:
    Soupe(const std::string & n);
    std::string ToString() const;
    bool EstChaud() const;
    bool EstSucre() const;
 private:
   std::string _nom;
};
```

```
#include "Soupe.h"
using namespace std;
Soupe::Soupe(const string & nom):Boisson(4.5),
                                    _nom(nom){ }
string Soupe::ToString() const{
    return "Soupe:" + _nom;}

// L'implémentation de EstChaud()/EstSucre()
bool Soupe::EstChaud() const { return true;}
bool Soupe::EstSucre() const{ return false;}
```

```
  . . .              //UTILISATION
void main()
{   Boisson * pB =new Soupe("Potage de carottes");
    cout<< pB->ToString() << endl;
    cout<< "Prix=" <<pB->GetPrix() << endl;
    cout<< "Sucre="<< pB->EstSucre() << endl;
    delete pB;
    pB =new ChocolatChaud;
    cout<< pB->ToString() << endl;
    cout<< "Prix=" <<pB->GetPrix() << endl;
    cout<< "Sucre="<< pB->EstSucre() << endl;
    delete pB;
}
```

Résultats

```
Soupe:Potage de carottes
Prix=4.5
Chaud=1
Sucre=0
Chocolat Chaud
Prix=3.2
Chaud=1
Sucre=1
```

1.11. Les exceptions

Le C++ utilise les mots clé suivants :

`throw` : lancer une exception avec en paramètre une variable d'un type `Ty`

`try{ }`: bloc dans lequel des exceptions peuvent survenir

`catch(obj Ty){}` : interception des exceptions avec des variables de de type `Ty`

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

double Division(double n,double d){
    if(d!=0) return n/d;
    else{
        string s("Exc.:division par 0");
        throw s;
    }
}

void main(){
    try{
        cout << "1.2/4=" << Division(1.2,4) << endl;
        cout << "3.1/0=" << Division(3.1,0) << endl;
        cout << "-1.4/2.4=" << Division(-1.4,2.4) << endl;
    }
    catch(const string & s){    cout << s << endl; }
}
```

Sortie

```
1.2/4=0.3
Exc.:division par 0
```

Les classes STL lèvent des exceptions sur certaines opérations. La classe **`std::exception`** est classe de base pour les différents types d'exception.

```
#include <iostream>
#include<vector>
#include<exception>
#include <string>

using namespace std;

int main()
{
    vector<int> v={1,2,3};       // vecteur de taille 3
    string s="toto";      //chaine de taille 4
    try
    {
      // la méthode at(idx) vérifie l'indice idx
      // si idx invalide → exception
        cout << "v[2]=" << v.at(2) << endl;    // indice valide
        cout << "s[0]=" << s.at(0) << endl;  // indice valide

        v.at(5)=3;   // indice invalide → exception

        cout << "v[1]=" << v.at(1) << endl;  // pas exécuté
    }
    catch(const exception & e)
    {
        cout << e.what() << endl;

    }
    return 0;
}
```

Sortie

```
v[2]=3
s[0]=t
std::exception
```

1.12. Les fonctions paramétrées en type

Soit l'exemple d'une fonction de permutation des valeurs de deux variables entières.

```
#include <iostream>
using namespace std;
void permute(int & a,int & b)
{
 int c=a; a=b; b=c;
}

void main(){
       int x=2,y=3;
       permute(x,y);
       cout << x << y << endl; //x=3 y=2
}
```

Si l'on souhaite aussi permuter deux valeurs de type `double`, on peut surcharger permute.

```
void permute(int & a,int & b)
{ int c=a; a=b; b=c;       }

void permute(double & a, double & b)
{ double c=a; a=b; b=c; }

void main(){
       ...
}
```

Pourquoi écrire deux fois la même fonction ? La seule différence entre ces deux fonctions concerne les types utilisés. Si l'on peut paramétrer les types, on n'a qu'une seule définition à donner. C'est ce que permet la notation `template` (patron).

```
// patron de fonction paramétrée en type
template<class T> void permute(T & a,T & b)
{
  T c=a; a=b; b=c;
}
void main(){
       int x=2,y=3;
       double u=1.3,v=1.7;
       permute(x,y);// T=int
       permute(u,v);// T=double
}
```

La notation **`template<class T>`** indique au compilateur que **`T`** est un paramètre de type. La fonction permute peut alors être appelée avec deux arguments du même type (ça n'aurait pas de sens de permuter un caractère avec un flottant).

Dans cet exemple, le compilateur va créer deux fonctions différentes à partir du patron. La première fonction est créée pour `T=int` et la seconde pour `T=double`. Une fonction paramétrée en type définit donc une famille de fonctions ayant toutes le même algorithme mais pour des types différents.

1.13. Classes paramétrées en type

On peut également introduire des paramètres de type dans la définition d'une classe. D'ailleurs, les conteneurs STL sont des classes paramétrées en type. Prenons l'exemple d'une classe `Point` écrite de façon à pouvoir instancier des points dont les coordonnées sont d'un type choisi par l'utilisateur.

```
void main()
{
	Point<int> p1(12,16);		// point à coordonnées entières
	Point<double> p2(1.3,4.6);	// point à coordonnées rationnelles
}
```

Définition de la classe paramétrée **`Point<T>`.**

```
//fichier point.h
#pragma once

template<class T>   // T est un paramètre de type
class Point
{
private:
	T _XY[2];	// tableau de 2 éléments de type T
public:
	Point<T>(){ _XY[0]=0; _XY[1]=0;}

	Point<T>(const T & x,const T & y)
	{
		_XY[0]=x;
		_XY[1]=y;
	}

	Point<T>(const Point<T> & p)
	{
		_XY[0]=p._XY[0];
		_XY[1]=p._XY[1];
	}

	Point<T> & operator=(const Point<T> & p)
	{
		_XY[0]=p._XY[0];
		_XY[1]=p._XY[1];
		return *this;
	}

	void SetX(const T & x) { _XY[0]=x; }
	void SetY(const T & y){ _XY[1]=y; }

	const T GetX() const { return _XY[0]; }
	const T GetY() const { return _XY[1]; }

};
```

Remarque: pour les classes paramétrées en type il n'y a qu'un fichier header (extension .h) contenant toute l'implémentation des méthodes (comme ci-dessus).

1.14. Les namespaces

La notion de ***namespace*** est identique à celle de C# (notation un peu différente). Un namespace regroupe des types. **L'opérateur ::** permet d'indiquer l'appartenance à un namespace:

NA::MaFonction() <=> fonction MaFonction() dans le namespace NA
std::cout <=> objet cout du namespace std

```
#include <iostream>
namespace NA{
      void MaFonction(){ std::cout << "Fonction du namespace NA \n"; }
 }
namespace NB{
      void MaFonction(){ std::cout << "Fonction du namespace NB \n"; }
}
```

```
using namespace NA;  // utilise le namespace NA par défaut

void main(void){
      NA::MaFonction();   // fonction du namespace NA
      NB::MaFonction();   // fonction du namespace NB
      MaFonction();       // fonction du namespace par défaut, ici NA
}
```

Important : ne pas utiliser `using namespace` dans les fichiers header (.h)

Exemple de classe **`Rationnel`** définie dans un namespace **`sagi`**

```
#pragma once       //HEADER Rationnel.h
#include<iostream>

⚠ INTERDIT : << using namespace machin; >> dans FICHIER HEADER

namespace sagi{
      class Rationnel{
        public:
            Rationnel(int num=0, int den=1);
            ...
      };
      std::ostream & operator<<(std::ostream & flot, const Rationnel & r);
}
```

```
// main.cpp
#include "Rationnel.h"
using namespace std; //ici [OK]
using namespace sagi;

void main(){
      // sans using, on doit écrire
      // sagi::Rationnel r1(2,3);
      Rationnel r1(2,3),r2(4,5);

      // sans using, on doit écrire
      // std::cout << " r1=";
      cout << " r1=" << r1 << endl;
      cout << " r2=" << r2 << endl;
}
```

```
#include "Rationnel.h"  // Rationnel.cpp
using namespace std;  //ici [OK]

namespace sagi
{
 Rationnel::Rationnel(int num,int den)
 {
       ...
 }
  ...
}
```

1.15. Flot stringstream (type numérique -> string)

La classe `stringstream` est une classe de flots permettant de gérer des chaînes.

```
#include<sstream>  // header stringstream
#include<iostream>

using namespace std;

int main()
{
    stringstream ss;

    int var=128;
    double d=2.7;
    ss << "var=" <<var << " d=" << d;   // insère dans le flot ss

    string s = ss.str();  //le flot ss nous donne un objet string

    cout << s << endl;   // la chaîne s contient  "var=128 d=2.7"
}
```

1.16. Conversion string vers type numérique (parser)

La STL fournit un certain nombre de fonctions pour convertir une chaîne représentant une valeur numérique en valeur numérique. La question est ici de transformer la chaîne "128" en l'entier 128 ou la chaîne "2.47" en un nombre à virgule valant 2.47

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

using namespace std;

int main()
{

    string s1="128";
    string s2="2.47";

    int varI;
    double varD;

    varI=std::stoi(s1);
    varD=std::stod(s2);

}
```

1.17. Les fichiers

En C++, les fichiers peuvent être gérés par des flots de type `fstream` (file stream). La classe `fstream` dispose des méthodes suivantes (ou héritées de `ios`, `istream`, `ostream`)

```
fstream::fstream() : crée un objet non connecté à un fichier
fstream::fstream(char * nomFichier, ios::open_mode ) : ouvre un fichier
fstream::~fstream() : fermeture d'un fichier ouvert

fstream::open(char *, ios::open_mode) : ouvre un fichier
fstream::close() : ferme le fichier en cours
```

Quelques équivalences entre les modes d'ouverture des flots et les modes d'ouverture par fopen() en C.

ios::open_mode	mode pour la fonction fopen()
ios::out	w
ios::in	r
ios::out|ios::app	a
ios::in|ios::out	r+
ios::in|ios::binary	rb
ios::out|ios::binary	wb
ios::out|ios::app|ios::binary	ab

```
int main()
{
	char tampon[150];

	// création ou remplacement du fichier
	fstream f("toto.txt",ios::out);

	// écriture dans le fichier
	f << "test \t" << 1 << "\t" << 1.5 << endl;
	f.close();

	// réouverture du fichier en mode ajout
	f.open("toto.txt",ios::out|ios::app);
	f<<"ajout \t" << 2 << "\t"<< 2.5 << endl;	// écriture dans le fichier
	f.close();

	// ouverture en mode lecture
	f.open("toto.txt",ios::in);

	while(!f.eof()){
		f.getline(tampon,150);
		cout  << tampon << endl;
	}
	return 0;
}
```

toto.txt

```
test	1	1.5
ajout	2	2.5
```

Sortie console

```
test	1	1.5
ajout	2	2.5
```

1.18. L'accès direct dans les fichiers

La classe `fstream` hérite de méthodes de positionnement.

Pour la lecture (hérité de istream) :

```
fstream::seekg(déplacement, position) : positionne le pointeur
fstream::tellg() : retourne la position courante
```

Idem pour l'écriture (hérité de ostream) :

```
fstream::seekp(déplacement, position)
fstream::tellp()
```

La classe `ios` définit pour cela des valeurs `ios::beg` (début du flot), `ios::end` (fin du flot), `ios::cur` (position courante). Il est à noter que la lecture ou l'écriture font évoluer la position courante.

```
#include<fstream>
#include<iostream>
using namespace std;

int main()
{
	char tampon[150];

	// ouverture en écriture
	fstream fic1("fichier1.txt",ios::out);
	for(int i=0;i<3;i++) fic1<<"ligne numero "<< i << "\n";
	fic1.close();

	fstream fic2("fichier2.txt",ios::out);
	for(int i=0;i<3;i++) fic2<<"ligne numero "<< i << "\n";
	fic2.seekp(0,ios::beg);  //positionne en début
	fic2<<"xx\n";		// remplace 'l' 'i' 'g' 'n'
	fic2.close();		// 4 caractères 'x' 'x' + retour

	// lit et affiche chaque ligne
	fstream fic3("fichier2.txt",ios::in);
	while(!fic3.eof()){
		fic3.getline(tampon,150);
		cout << tampon <<endl;
	}
	return 0;
}
```

fichier1.txt

```
ligne numero 0
ligne numero 1
ligne numero 2
```

fichier2.txt

```
xx
e numero 0
ligne numero 1
ligne numero 2
```

Sortie console

```
xx
e numero 0
ligne numero 1
ligne numero 2
```

1.19. Gestion de projets en C++

Pour chaque classe, on a un fichier header (extension .h) et un fichier source (extension cpp). Le programme principal (fonction main()) est dans un dernier fichier (.cpp).

Il faut créer un ***projet*** incluant tous les fichiers sources (fichiers cpp). Les fichiers header doivent être visibles par les fichiers qui les utilisent. Le plus simple étant de tout mettre dans un même répertoire.

Fichier header : les directives de compilation conditionnelle sont souvent remplacées par `#pragma once`

```
// MaClasse.h (header)

#ifndef __MA_CLASSE__        // directives de compilation conditionnelle
#define __MA_CLASSE__        // rôle : éviter inclusions multiples

#include <iostream>       // pour les types utilisés ici

class MaClasse
{
      public:
            MaClasse();

            void Affiche(std::ostream & sortie) const;
};
#endif
```

Fichier source

```
// MaClasse.cpp (source)
#include "maclasse.h"  // fichier MaClasse.h doit être dans le même répertoire

MaClasse::MaClasse()
{
      ... //source du constructeur sans argument
}

//Noter que le type std::ostream est connu ici car inclus via MaClasse.h
void MaClasse::Affiche(std::ostream & sortie) const
{
      sortie << "Affichage sur sortie standard";
}
```

Fichier d'utilisation de la classe

```
// utilisation.cpp
#include "maclasse.h"   // parce que le type MaClasse est utilisé ici
                      // le type std::ostream est inclus indirectement
void main()
{
      MaClasse m;
      m.Affiche(std::cout);
}
```

Projet C++ : pour pouvoir compiler et éditer les liens (linker) et ainsi obtenir un programme exécutable (exe), les deux fichiers `maclasse.cpp` et `utilisation.cpp` doivent être dans un même projet.

Chaîne de compilation (ce que l'IDE va faire pour produire l'exécutable):

(compilation) [source texte] **`MaClasse.cpp`** → **`MaClasse.obj`** [code binaire]
(compilation) [source texte] **`Utilisation.cpp`** → **`Utilisation.obj`** [code binaire]
(linker) [binaire] **`MaClasse.obj + Utilisation.obj`** [+binaire librairies] → **`MonExe.exe`** **`[binaire]`**

2. Introduction aux classes STL d'usage fréquent

2.1. Introduction

Le C++ introduit la notion de fonctions et de classes *paramétrées en type*. Cette notion est déjà présente en C# (penser aux classes `List<T>` `LinkedList<T>` par exemple). Cela signifie qu'au sein d'une classe ou d'une fonction, certains types peuvent être passés en paramètre. Le mot clé *template* est introduit en C++ pour pouvoir paramétrer des types.

La bibliothèque **STL** (**Standard Template Library**) est fournie avec le compilateur C++. On y retrouve des classes *conteneur* paramétrées et des algorithmes paramétrés (recherche de valeurs dans un conteneur, tris de conteneur …)

Notion de conteneur Un conteneur est simplement un objet capable de stocker des données. En C++ :

- le conteneur `vector<Ty>`: tableau unidimensionnel d'éléments de type `Ty`
- le conteneur `list<Ty>`: liste doublement chaînée d'éléments de type `Ty`
- le conteneur `stack<Ty>`:pile (structure LIFO)
- le conteneur `queue<Ty>`:file (structure FIFO)
- le conteneur `deque<Ty>`: structure de données hybride
- le conteneur `set<Ty>`: ensemble
- le conteneur `map<key,Ty>`: tableau associatif associant à une clé de type `key` une valeur de type `Ty`
- classe `string` : classe de chaînes de caractères

Par exemple, `vector<int>` est une classe de tableaux d'entiers, `vector<Point>` est une classe de tableaux d'objets Point. A noter que `vector<int>` est une classe et `vector<Point>` en est une autre. Il n'y a pas de compatibilité entre ces types.

```
#include<iostream>   // header pour les classes istream et ostream
#include<vector>     // header pour le conteneur vector
using namespace std;  // la bibliothèque STL utilise le namespace std

void main(void){
      vector<int> tab1(4);
      vector<float> tab2(2);
      vector<vector<int>> tab3(2);
      tab1[1]=3;     // tab1 stocke des entiers
      tab2[0]=3.7;   // tab2 stocke des float
      tab3[0]=tab1;  // tab3 stocke des tableaux d'entiers (vector<int>)

      for(unsigned i=0;i<tab1.size();i++){ cout << tab1[i] << " ";}
}
```

Pour les vecteurs, la taille initiale est passée au constructeur de l'objet. L'objet `tab1` peut donc stocker 4 entiers, `tab2` peut stocker 2 float et `tab3` peut stocker 2 objets de la classe `vector<int>`. On peut donc copier `tab1` dans une case du tableau `tab3` (`tab3` est un tableau de tableaux).

Remarquer que l'accès à une case d'un objet `vector<T>` se fait simplement avec l'opérateur `[]`, comme sur un tableau classique en C.

L'objectif n'est pas de détailler ici toutes les possibilités de la bibliothèque STL, mais de présenter quelques caractéristiques des classes `string`, `vector`, et `list` qui sont des classes souvent utilisées.

2.2. Classe `string`

La classe string n'est pas une classe paramétrée en type. Elle sert uniquement à gérer des chaînes de caractères de façon un peu plus souple qu'en langage C. Voici un premier exemple.

```
#include<string>  // pour la classe string
#include<iostream>
using namespace std;

void main(void)
{
      string s1("Une chaine"),s2(" Toto");
      string s3(s1);
      string s4;

      s4=s1+s2;     // + : concaténation   = : affectation

      cout << s3 << endl;        // affichage avec cout
      cout << s3[0] << endl;     // utilisable comme tableau : s3[i] de type char

      if(s1==s3) cout << "Chaines identiques\n";

      if(s1<=s4) cout << "s1<=s4 selon l'ordre lexicographique\n";
}
```

Autre exemple simple.

```
#include<string>
#include<iostream>
using namespace std;

void main(void)
{
      string s1("abcdefg");

      cout << "Longueur = " << s1.length() << endl;
      cout << "Capacite = " << s1.capacity() << endl;
      cout << s1 << endl; // s1 = abcdefg

      s1.insert(1,"ABC"); // insère la chaîne "ABC" à l'indice 1
      cout << s1 << endl; // s1 = aABCbcdefg

      s1.insert(2,4,'a'); // insère 4 fois la lettre 'a' à l'indice 2
      cout << s1 << endl; // s1 = aAaaaaBCbcdefg

      s1.erase(0,1);             // supprime 1 caractère à l'indice 0
      cout << s1 << endl; // s1 = AaaaaBCbcdefg

      s1.erase(2,5);             // supprime 5 caractères à l'indice 2
      cout << s1 << endl; // s1 = Aabcdefg

      // possibilité d'obtenir un pointeur compatible const char *
      const char * ptr=s1.c_str();
}
```

Sortie

```
Longueur = 7
Capacite = 7
abcdefg
aABCbcdefg
aAaaaaBCbcdefg
AaaaaBCbcdefg
Aabcdefg
```

2.3. Fiche string

Par la suite, on considère les objets et les variables suivants :

```
string s, s1, s2;          int i, start, len, start1, len1, start2, len2, newSize;
char c;                    istringstream istr; // requires <sstream>
char* cs;                  ostringstream ostr; // requires <sstream>
bool b;
```

Méthodes et opérateurs de la classe string

Méthode	Description
Constructeurs	
`string s;`	chaîne vide
`string s(s1);`	s est une copie de s1
`string s(cs);`	s contient les caractères de la chaîne pointée par cs
Modificateurs	
`s1=s2;`	Assigne s2 à s1.
`s1=cs;`	Assigne la chaîne pointée par cs à s1.
`s1=c;`	Assigne une chaîne avec un seul caractère à s1.
`s[i]= c;`	Modifie le caractère d'indice i (i à partir de 0)
`s.at(i) = c;`	Modifie le caractère d'indice i. Exception si indice invalide
`s.append(s2);`	Concatène s2 à la fin de s. Idem s += s2;
`s.append(cs);`	Concatène la chaîne pointée par cs à la fin de s.
`s.assign(s2,start,len);`	Assigne la sous-chaîne s2[start..start+len-1] à s.
`s.clear();`	Retire tous les caractères de s
Accesseurs	
`cs =s.c_str();`	Retourne la chaîne C équivalente
`s1 =s.substr(start, len);`	Retourne la sous-chaîne s[start..start+len-1].
`c =s[i];`	Caractère d'indice i
`c =s.at(i);`	Caractère d'indice i. Exception si indice invalide
Taille	
`i =s.length();`	Retourne la taille de la chaîne
`i =s.size();`	Retourne la taille de la chaîne
`i =s.capacity();`	Nombre de caractères que s peut stocker avant réallocation
`b =s.empty();`	Indique si s est vide.
`i =s.resize(newSize,padChar);`	Change la taille de la chaîne et complète avec padChar
Recherche : retourne string::npos en cas d'échec	
`i =s.find(c);`	Position de l'occurence de c la plus à gauche dans s
`i =s.find(s1);`	Position de l'occurence de s1 la plus à gauche dans s
`i =s.rfind(s1);`	Idem find mais par la droite
`i =s.find_first_of(s1);`	Position du premier char de s également dans s1.
`i =s.find_first_not_of(s1);`	Position du premier char de s qui n'est pas dans s1.
`i =s.find_last_of(s1);`	Position du dernier char de s également dans s1.
`i =s.find_last_not_of(s1);`	Position du dernier char de s qui n'est pas dans s1.
Comparaison	
`i =s.compare(s1);`	<0 if s<s1, 0 if s==s1, or >0 if s>s1. (ordre lexicographique)
`i =s.compare(start1, len1, s1, start2, len2);`	Compare s[start1..start1+len1-1] à s1[start2..start2+len2-1]. Valeur de retour comme ci-dessus
`b =(s1==s2) [>,<, >=,<=,!=]`	Comparaisons selon l'ordre lexicographique
Entrée/Sortie	
`cin >> s;`	>> surchargé pour la saisie de chaînes
`getline(cin, s);`	Place une ligne de cin dans s.
`cout << s;`	<< surchargé pour l'affichage de chaînes
Concaténation	

Les opérateurs + et += réalisent la concaténation (l'ajout en fin)

```
s=s1+s2;
s+=s2;
```

2.4. vector<T>

La classe paramétrée `vector<T>` engendre des objets tableaux. On rappelle que `vector<int>` est une classe (le paramètre de type vaut ici `int`) et `vector<float>` en est une autre.

```
#include<iostream>
#include<vector>
using namespace std;

void main(void)
{
       vector<int> v={3,7};  //C#11          // v={3,7}
       vector<int> t1(4,-2),t2(5);           // t1={-2,-2,-2,-2}   t2={0,0,0,0,0}
       vector<float> t3(2),t4(4,1.2);        // t3={0,0}           t4={1.2,1.2,1.2,1.2}

       t1[0]=5;       //t1={5,-2,-2,-2}
       t4[2]=-2.3;    //t4={1.2,1.2,-2.3,1.2}

       t2=t1;  // t1 ppv t2 (OK même classe)
       t3=t4;  // t3 ppv t4 (OK même classe)

       //t1=t4; //KO (t1 et t4 de types différents)

       vector<int> t5(t1);   // constr. copie t5={5,-2,-2,-2}
       vector<float> t6(t3); // constr. Copie t6={1.2,1.2,-2.3}

       //vector<float> t7(t1); //KO car t1 n'est pas de la classe vector<float>
}
```

La classe dispose d'un constructeur à un argument (la taille initiale du vecteur), d'un constructeur à deux arguments (taille et valeurs initiales du vecteur). La classe dispose aussi d'un constructeur de copie.

La classe dispose d'un opérateur d'affectation. Enfin, puisque c'est une classe de tableaux, l'opérateur d'indexation [ ] donne accès aux éléments.

Les itérateurs

Dans le but d'homogénéiser la STL, les classes conteneur implémentent la notion *d'itérateur* qui généralise la notion de pointeur.

Un itérateur est un objet qui *pointe* un emplacement d'un conteneur (un peu comme un pointeur). Les opérateurs `++` et `--` sont surchargés sur les itérateurs pour passer à l'emplacement suivant ou précédent du conteneur. L'opérateur `*` permet l'indirection sur un itérateur (comme sur un pointeur).

Les classes conteneurs contiennent des classes d'itérateurs adaptés. Il y a deux classes d'itérateurs par conteneur: les **itérateurs constants** et **les itérateurs non constants**. Les premiers permettent de parcourir un conteneur sans pouvoir en modifier le contenu.

```
#include<vector>
#include<iostream>
using namespace std;
int main(void)
{
       vector<int> t1(4);     // t1 ={0,0,0,0}
       const vector<int> t2({1,2,3,4});  // t2 est un vecteur constant
       vector<int>::iterator it;     // it est un iterateur sur vector<int>
       vector<int>::const_iterator itConst; //itConst peut seulement lire

    // parcours « classique » du « tableau » t1
       for(unsigned i=0;i<t1.size();i++)
       {     t1[i]=2;     }
```

```
    // parcours de t1 avec itérateur it
    for(it=t1.begin();it!=t1.end();it++)   // écrit 3 dans toutes les cases de t1
    {
              (*it)=3;   // écrit 3 dans la case désignée par it
    }

    // parcours de t1 avec itérateur constant, cet itérateur ne peut pas modifier le contenu
    for(itConst=t1.begin();itConst!=t1.end();itConst++) // affiche toutes les cases de t1
    {
        cout << *itConst <<"|";    // affiche l'élément désigné par itConst
       // *itconst = value; // interdit parce que c'est un const_iterator
    }

    // t2 qui est constant, ne peut être parcouru que par itérateur constant.
    for(itConst=t2.begin();itConst!=t2.end();itConst++) // affiche toutes les cases de t2
    {
        cout << *itConst <<"|";   // affiche l'élément désigné par itConst
    }
}
```

Il faut avoir une connaissance minimale des itérateurs pour pouvoir exploiter la bibliothèque STL (certaines méthodes prennent des itérateurs en paramètre). Néanmoins, l'utilisation des itérateurs est la même pour les autres conteneurs.

```
#include<vector>
#include<iostream>
using namespace std;

  void affiche(const vector<int> & v)
  {
      cout << "|";
      for(int i=0;i<v.size();i++)    cout << v[i] << "|";
      cout << endl;
  }
void main(void)
{
    vector<int> t1(3,2); //t1={2,2,2}
    cout << "Taille = "<< t1.size() << endl; // Taille=3
    affiche(t1);
    t1.push_back(3); //t1={2,2,2,3}
    affiche(t1);
    t1.resize(6,-2);  //t1={2,2,2,3,-2,-2}

    affiche(t1);
    t1.insert(t1.begin(),2,-3); //t1={-3,-3,2,2,2,3,-2,-2}
    affiche(t1);
    t1.insert(t1.begin()+3,-1); //t1={-3,-3,2,-1,2,2,3,-2,-2}
    affiche(t1);
    t1.insert(t1.begin(),-6); //t1={-6,-3,-3,2,-1,2,2,3,-2,-2}
    affiche(t1);
    t1.erase(t1.begin()+2,t1.begin()+4); //t1={-6,-3,-1,2,2,3,-2,-2}
    affiche(t1);
    t1.erase(t1.begin()+3); //t1={-6,-3,-1,2,3,-2,-2}
    affiche(t1);
    t1.pop_back(); //t1={-6,-3,-1,2,3,-2}
    affiche(t1);
}
```

Résultats

```
Taille = 3
|2|2|2|
|2|2|2|3|
|2|2|2|3|-2|-2|
|-3|-3|2|2|2|3|-2|-2|
|-3|-3|2|-1|2|2|3|-2|-2|
|-6|-3|-3|2|-1|2|2|3|-2|-2|
|-6|-3|-1|2|2|3|-2|-2|
|-6|-3|-1|2|3|-2|-2|
|-6|-3|-1|2|3|-2|
```

2.5. Fiche `vector<T>`

Par la suite on considère les objets et variables suivants (T est un type)

```
T e;      // e variable de type T
vector<T> v, v1;
vector<T>::iterator iter, iter2, beg, end;
vector<T>::reverse_iterator riter;  /* beg, end could also be here */
int i, n;
bool b;
```

Intervalle d'itérateurs : la séquence d'éléments désignés par les itérateurs **beg** et **end** est

***beg, *(beg+1), …, *(end-1).**

Autrement dit, tous les éléments allant de l'itérateur **beg** jusqu'à l'itérateur **end-1** inclus.

Constructeurs/destructeur	
`vector<T> v;`	Crée un vecteur vide d'éléments de type T
`vector<T> v(n);`	Crée un vecteur de taille n avec valeur par défaut
`vector<T> v(n,e);`	Crée vecteur de taille n contenant e dans toutes les cases
`vector<T> v(beg,end);`	Crée un vecteur à partir des éléments *beg → *(end-1)
`v.~vector<T>();`	Libère la mémoire
Taille	
`i =v.size();`	Nombre d'éléments
`i =v.capacity();`	Nombre d'éléments avant réallocation
`i =v.max_size();`	Nombre maximum d'éléments
`b =v.empty();`	Teste la vacuité.
`v.reserve(n);`	Définit la capacité avant reallocation
Modificateurs	
`v =v1;`	Affectation
`v[i]=e;`	Modifie l'élément d'indice i
`v.at(i)=e;`	Comme operator[] avec exception si i invalide
`v.front()=e;`	v[0] = e.
`v.back()=e;`	v[v.size()-1] = e.
`v.push_back(e);`	Insère e en fin (réalloue si nécessaire)
`v.pop_back();`	Supprime le dernier
`v.clear();`	Supprime tous les éléments
`iter =v.assign(n,e);`	Remplace les éléments par n copies de e.
`iter =v.assign(beg,end);`	Affecte les éléments *beg, → *(end-1) à v
`iter2 =v.insert(iter,e);`	Insère une copie de e à la position iter et retourne sa position
`v.insert(iter,n,e);`	Insère n copies de e à partir de la position iter.
`v.insert(iter,beg,end);`	Insère à la position iter tous les éléments *beg, → *(end-1)
`iter2 =v.erase(iter);`	Supprime l'élément à la position iter et retourne la position de l'élément suivant.
`iter =v.erase(beg,end);`	Supprime les éléments compris entre beg et end-1 et retourne la position de l'élément suivant.
`iter =v.resize(n,val);`	Nouvelle taille n. Si la taille augmente, les nouvelles cases sont initialisées avec val
Accesseurs	
`e =v[i];`	Accès au i-ème élément. Pas de vérification de validité de i
`e =v.at(i);`	Accès au i-ème élément. Déclenche exception si i invalide
`e =v.front();`	Premier élément. Pas de vérification de validité
`e =v.back();`	Dernier élément. Pas de vérification de validité
Iterateurs	
`iter =v.begin();`	Retourne un itérateur désignant le premier élément.
`iter =v.end();`	Retourne un itérateur désignant juste après le dernier élément.
`riter =v.rbegin();`	Idem en ordre inverse
`riter =v.rend();`	Idem en ordre inverse
`++iter;`	Pre-incrément de l'itérateur
`--iter;`	Pre-decrément de l'itérateur
`iter2 =iter+ i;`	iter+i est un itérateur (comparable arithmétique des pointeurs)
`iter2 =iter- i;`	Iter -i est un itérateur
`e =*iter;`	De-réference pour obtenir la valeur désignée par l'itérateur

2.6. Conteneurs `list<T>`

Les autres conteneurs ont de nombreux points communs avec ce qu'on a déjà vu. C'est pourquoi un exemple devrait suffire à comprendre l'utilisation des listes chaînées.

```
#include<list>
#include<iostream>
using namespace std;

void affiche(const std::list<int> & l){
       list<int>::const_iterator itC;
       cout << "|";
       for(itC=l.begin();itC!=l.end();itC++)    cout << *itC << "|";
       cout << endl;
}

void affiche(const std::list<float> & l){
       list<float>::const_iterator itC;
       cout << "|";
       for(itC=l.begin();itC!=l.end();itC++)    cout << *itC << "|";
       cout << endl;
  }

void main(void)
{
       list<int> l1;                 //liste vide d'entiers
       list<float> l2,l4;            // liste vide de réels

       l1.push_front(12);    // ajout en tête
       l1.push_back(13);     // ajout en fin
       l1.push_front(-3);    // ajout en fin
       l1.push_front(7);     // l1=7,-3,12,13
       affiche(l1);

       l2.push_back(2.3);
       l2.push_back(2.7);
       l2.push_back(-1.2);   //l2=2.3,2.7,-1.2
       affiche(l2);

       list<int> l3(l1);     //l3=7,-3,12,13
       cout << "Taille = "<< l3.size() << endl;

       l4=l2; //l4 copie de l2 soit l4=2.3,2.7,-1.2
       affiche(l4);

       l1.insert(l1.begin(),2,-3); //l1=-3,-3,7,-3,12,13
       affiche(l1);
       cout << l1.front() <<" "<< l1.back() << endl; // -3 13

       list<int>::iterator it=l1.begin();
       while(it!=l1.end())
       {
              if((*it)==-3)  // suppression des noeuds valant -3
              {
                     it=l1.erase(it);
              }
              else it++;
       }
        affiche(l1);
}
```

Résultats

```
|7|-3|12|13|
|2.3|2.7|-1.2|
Taille = 4
|2.3|2.7|-1.2|
|-3|-3|7|-3|12|13|
-3 13
|7|12|13|
```

2.7. Fiche `list<T>`

Classe de listes doublement chaînées d'éléments de type T. Par la suite, on considère les objets et variables suivants (T est un type)

```
T e;
list<T> v, lst2;
list<T>::iterator iter, iter2, beg, end;
int i, , n, size;
```

Intervalle d'itérateurs : la séquence d'éléments désignés par les itérateurs **beg** et **end** est ***beg**, ***(beg+1)**, …, ***(end-1).** Autrement dit, tous les éléments allant de l'itérateur beg jusqu'à l'itérateur end-1 inclus.

Méthode	Description
Constructors and destructors	
`list<T> lst;`	Liste vide d'éléments de type T
`list<T> lst(n, e);`	Liste contenant n copies de e.
`list<T> lst(beg, end);`	Liste créée avec les éléments entre *beg → *(end-1).
`lst.~list<T>();`	Désalloue la mémoire
`i =lst.size();`	Nombre d'éléments
`b =lst.empty();`	Vrai si vide. A utiliser plutôt que lst.size()==0 qui peut faire un parcours de la liste
`lst =lst2;`	Affectation de liste
`lst.clear();`	Vide la liste.
`lst.assign(n, e);`	Affecte n copies de e à la liste courante (supprime les autres)
`lst.sort();`	Trie la liste en utilisant < sur les éléments
`lst.unique();`	Si lst est triée, supprime les doublons
`lst.reverse();`	Inverse lst.
`lst.merge(lst2);`	Fusion de deux listes triées
`e =lst.front();`	Premier element.
`e =lst.back();`	Dernier element.
`lst.push_front(e);`	Ajoute e en tête
`lst.pop_front();`	Supprime le premier
`lst.push_back(e);`	Ajoute e en fin
`lst.pop_back();`	Supprime le dernier
Iterateurs	
`iter2 =lst.assign(beg,end);`	Affecte les éléments compris entre [beg et [end
`iter2 =lst.insert(iter,e);`	Insére e à la position iter, retourne sa position.
`lst.insert(iter,n,e);`	Insére n copies de e à partir de la position iter
`lst.insert(iter,beg,end);`	Insére à la position iter, les éléments *beg → *(end-1)
`iter2 =lst.erase(iter);`	Supprime l'élément à la position iter, renvoie position du suivant
`iter2 =lst.erase(beg,end);`	Supprime les éléments entre beg et end-1 inclus, renvoie position du suivant
`lst.splice(iter,lst2);`	Insére une copie des éléments de lst2 avant iter.
`lst.splice(iter,lst2,beg);`	Insére avant iter une copie des éléments de lst2 commençant à beg.
`lst.splice(iter,lst2,beg,end);`	Insére, avant iter, les éléments *beg → *(end-1) de lst2.
`beg =lst.begin();`	Renvoie un itérateur sur le premier
`end =lst.end();`	Retourne un itérateur juste après le dernier
`beg =lst.rbegin();`	Retourne un itérateur sur le premier (en ordre inverse).
`end =lst.rend();`	Retourne un itérateur sur le dernier (en ordre inverse).
`lst.remove(e);`	Supprime tous les éléments valant e dans lst.