Chapitre 7 : Notions avancées

I52 Programmation Orientée Objet — C++

Valérie Gillot

Octobre 2020

Plan du chapitre

Section 1

Généricité

- Principe
- Modèles de fonctions
- Modèles de classes

Généricité : généralités

En programmation

La généricité (ou programmation générique) consiste à définir des algorithmes identiques opérant sur des données de types différents. L'algorithme doit être indépendant de la structure de données. La généricité est une forme de polymorphisme : polymorphisme de type

En C++

les méthodes virtuelles permettent un typage dynamique

Principe de la programmation générique

Elle consiste à abstraire un ensemble de concepts cohérents pour construire des algorithmes au-dessus indépendamment de leur implantation.

Généricité : généralités

En programmation

La généricité (ou programmation générique) consiste à définir des algorithmes identiques opérant sur des données de types différents. L'algorithme doit être indépendant de la structure de données. La généricité est une forme de polymorphisme : polymorphisme de type

En C++

les méthodes virtuelles permettent un typage dynamique

Principe de la programmation générique

Elle consiste à abstraire un ensemble de concepts cohérents pour construire des algorithmes au-dessus indépendamment de leur implantation.

Généricité : généralités

En programmation

La généricité (ou programmation générique) consiste à définir des algorithmes identiques opérant sur des données de types différents. L'algorithme doit être indépendant de la structure de données. La généricité est une forme de polymorphisme : polymorphisme de type

En C++

les méthodes virtuelles permettent un typage dynamique

Principe de la programmation générique

Elle consiste à abstraire un ensemble de concepts cohérents pour construire des algorithmes au-dessus indépendamment de leur implantation.

Généricité en C++

Modèle

Un modèle définit une famille de fonctions ou de classes paramétrée par une liste d'identificateurs qui sont

- soit des valeurs indiquées par leur type
- soit des types préfixés de `class` ou de `typename`

Instanciation

L'instanciation est la production effective d'un élément de la famille pour cela les paramètres doivent recevoir pour valeur

- une expression dont la valeur est connue lors de la compilation
- un type

Généricité en C++

Modèle

Un modèle définit une famille de fonctions ou de classes paramétrée par une liste d'identificateurs qui sont

- soit des valeurs indiquées par leur type
- soit des types préfixés de `class` ou de `typename`

Instanciation

L'instanciation est la production effective d'un élément de la famille pour cela les paramètres doivent recevoir pour valeur

- une expression dont la valeur est connue lors de la compilation
- un type

Modèles ou `template`

Les modèles : `template`

Les types génériques ou `template` (patrons ou modèles) sont utilisés pour les fonctions ou les classes :

1. les fonctions génériques (ou patrons/modèles de fonctions)
2. les classes génériques (ou patrons/modèles de classes)

Exemple

```
template<typename T, typename Q, int N>
//declaration ou definition d'une foncfion ou d'une classe
```

Ici `T` et `Q` sont des types (`class` avant C++ 17) et `N` une valeur de type `int`

Modèles ou `template`

Les modèles : `template`

Les types génériques ou `template` (patrons ou modèles) sont utilisés pour les fonctions ou les classes :

1. les fonctions génériques (ou patrons/modèles de fonctions)
2. les classes génériques (ou patrons/modèles de classes)

Exemple

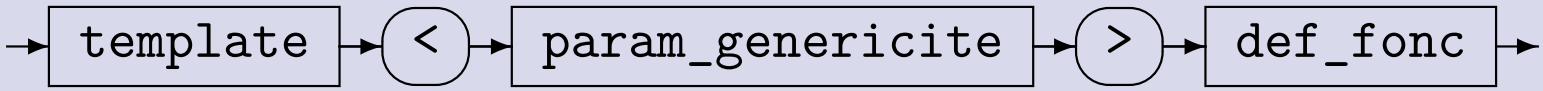
```
template<typename T, typename Q, int N>
//declaration ou definition d'une foncfion ou d'une classe
```

Ici `T` et `Q` sont des types (`class` avant C++ 17) et `N` une valeur de type `int`

Modèles de fonctions

Fonction générique

Un modèle de fonction (ou fonction générique) qui agit sur plusieurs types. Plus puissant que la sur-définition de fonction mais plus restrictif car un seul algorithme.



Syntaxe

- Définition :

```
template<par_1,...,par_k>
type nom (arg_for_1,...,arg_for_n) ...
```

  les paramètres `par_1,...,par_k` peuvent avoir des valeurs par défaut.

- Instanciation par appel :

```
nom (arg_eff_1,...,arg_eff_n)
```

Modèles de fonctions : exemple 1

Définition

```
template <typename T>
T mini(T a, T b)
{
if (a<b)
  return a;
else
  return b;
}
```

Instanciation

```
int main (){
int x=2, y=4;
float r=5.2, s=1.4;
cout << mini(x,y)<<endl<<mini(r,s)
    <<endl;
return 0;
}
```

Exécution

```
2
1.4
```

Modèles de fonctions : exemple 1

Définition

```
template <typename T>
T mini(T a, T b)
{
if (a<b)
  return a;
else
  return b;
}
```

Instanciation

```
int main (){
int x=2, y=4;
float r=5.2, s=1.4;
cout << mini(x,y)<<endl<<mini(r,s)
     <<endl;
return 0;
}
```

Exécution

```
2
1.4
```

Modèles de fonctions : exemple 1

Définition

```
template <typename T>
T mini(T a, T b)
{
if (a<b)
  return a;
else
  return b;
}
```

Instanciation

```
int main (){
int x=2, y=4;
float r=5.2, s=1.4;
cout << mini(x,y)<<endl<<mini(r,s)
     <<endl;
return 0;
}
```

Exécution

```
2
1.4
```

Modèles de fonctions : exemple 2

Définition et instanciations

```
template <typename T, typename U>
T f(T x, U y, T z)
{
return x+y+z;
}
int main (){
int n=1, p=2, q=3;
float r=2.5, s=5.0;
cout<<f(n,r,p);//OK T=int, U=float, (int) n+r+p cast de f
cout<<f(r,n,s);//OK T=float, U=int, (float) r+n+s
cout<<f(n,p,q);//OK T=U=int, (int) n+p+q
return 0;
}
```

Exécution

```
5
8.5
6
```

Modèles de fonctions : exemple 2

Définition et instanciations

```
template <typename T, typename U>
T f(T x, U y, T z)
{
return x+y+z;
}
int main (){
int n=1, p=2, q=3;
float r=2.5, s=5.0;
cout<<f(n,r,p);//OK T=int, U=float, (int) n+r+p cast de f
cout<<f(r,n,s);//OK T=float, U=int, (float) r+n+s
cout<<f(n,p,q);//OK T=U=int, (int) n+p+q
return 0;
}
```

Exécution

```
5
8.5
6
```

Modèles de fonctions : exemple 2

Définition et instanciations

```
template <typename T, typename U> T f(T x, U y, T z){
return x+y+z;
}
int main (){
int n=1, p=2, q=3;
float r=2.5, s=5.0;
cout<<f(n,p,r);
//KO types de n et r differents pas de correspondance

cout<<f<int,float>(n,p,r);
//OK T=int, U=float, (int) n+(float)p+(int)r

cout<<f<float>(n,p,r);
//OK T=float, U= type(p)=int, (float) (float n)+p+r
return 0;
}
```

Exécution

```
ERROR
5
5.5
```

Modèles de fonctions : exemple 2

Définition et instanciations

```
template <typename T, typename U> T f(T x, U y, T z){
return x+y+z;
}
int main (){
int n=1, p=2, q=3;
float r=2.5, s=5.0;
cout<<f(n,p,r);
//KO types de n et r differents pas de correspondance

cout<<f<int,float>(n,p,r);
//OK T=int, U=float, (int) n+(float)p+(int)r

cout<<f<float>(n,p,r);
//OK T=float, U= type(p)=int, (float) (float n)+p+r
return 0;
}
```

Exécution

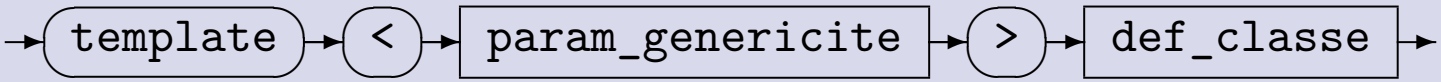
```
ERROR
5
5.5
```

Modèles de classes

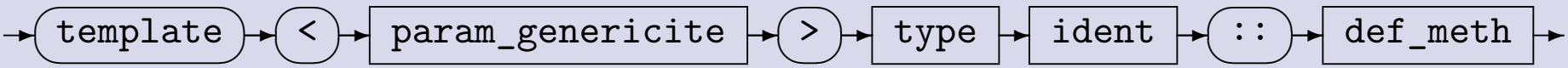
Classe générique

Un modèle de classe est un type paramétré par d'autres types et par des valeurs connues lors de la compilation

Déclaration de classe générique



Définition de méthode générique



Modèles de classes/methodes : instanciation

Syntaxe de l'instanciation d'un modèle de classe C

```
C <parametres effectifs> ident;
```

Instanciation d'un modèle de classe

Les paramètres effectifs pour instancier une classe sont obligatoires mais pas pour les méthodes.

Modèles de classe et méthode : exemple 1

Déclaration de classe *.h

```
template <typename T> class Point{
  T x, y;
public :
  Point(T abs = 0, T ord = 0)
    {x = abs; y = ord;}
  void Affic( );
};
```

Définition de méthode *.cc

```
template <typename T> void Point<T
    >::Affic( )
  {cout << "Coord : " << x << " "
       << y << endl;}
```

Utilisation

```
int main( ){
//instance avec T=int
Point<int> a1(3,5);
a1.Affic( );

//instance avec T=char
Point<char> a2('d','u');
a2.Affic( );
return 0;}
```

Exécution

```
Coord : 3 5
Coord : d u
```

Modèles de classe et méthode : exemple 1

Déclaration de classe *.h

```
template <typename T> class Point{
  T x, y;
public :
  Point(T abs = 0, T ord = 0)
    {x = abs; y = ord;}
  void Affic( );
};
```

Définition de méthode *.cc

```
template <typename T> void Point<T
    >::Affic( )
  {cout << "Coord : " << x << " "
      << y << endl;}
```

Utilisation

```
int main( ){
//instance avec T=int
Point<int> a1(3,5);
a1.Affic( );

//instance avec T=char
Point<char> a2('d','u');
a2.Affic( );
return 0;}
```

Exécution

```
Coord : 3 5
Coord : d u
```

Modèles de classe et méthode : exemple 1

Déclaration de classe *.h

```
template <typename T> class Point{
  T x, y;
public :
  Point(T abs = 0, T ord = 0)
    {x = abs; y = ord;}
  void Affic( );
};
```

Définition de méthode *.cc

```
template <typename T> void Point<T
    >::Affic( )
  {cout << "Coord : " << x << " "
      << y << endl;}
```

Utilisation

```
int main( ){
//instance avec T=int
Point<int> a1(3,5);
a1.Affic( );

//instance avec T=char
Point<char> a2('d','u');
a2.Affic( );
return 0;}
```

Exécution

```
Coord : 3 5
Coord : d u
```

Modèles de classe et méthode : exemple 1

Déclaration de classe *.h

```
template <typename T> class Point{
  T x, y;
public :
  Point(T abs = 0, T ord = 0)
    {x = abs; y = ord;}
  void Affic( );
};
```

Définition de méthode *.cc

```
template <typename T> void Point<T
    >::Affic( )
  {cout << "Coord : " << x << " "
      << y << endl;}
```

Utilisation

```
int main( ){
//instance avec T=int
Point<int> a1(3,5);
a1.Affic( );

//instance avec T=char
Point<char> a2('d','u');
a2.Affic( );
return 0;}
```

Exécution

```
Coord : 3 5
Coord : d u
```

Modèles de classe et méthode : Exemple 2

Tableau paramétré

```
//declaration de classe
template <typename T, int n = 0> class Tableau {
T tab[n]; //declaration d'un attribut tab avec une valeur n
public :...
};
//instance de la classe
int main( ){
Tableau<float,4> t;
//instanciation d'un objet t de la classe Tableau<float,4>
return 0;}
```

Modèles de classe et méthode : Exemple 3 [1]

modèle de classe

```
template<typename T,int maxsize>
class CDataArray{
  private:
  // attributs
    T array[maxsize];
    int length;

  public:
  //methodes
    CDataArray();
    int getSize();
    bool add(T element);
    //true si l'element a pu etre
        ajoute
};
```

Déclaration du modèle

le type des éléments et la taille du tableau sont des paramètres du modèle :

- le type des éléments du tableau T doit être instancié par un type
- la taille du tableau doit être instancié par une valeur (allocation statique déterminée à la compilation)

1. extrait du wikibooks

Modèles de classe et méthode : Exemple 3

Définition du constructeur

```
template<typename T,int maxsize>
CDataArray<T,maxsize>::CDataArray()
  : length(0)
  // Liste d'initialisation
      attributs
{}
```

Définition du constructeur

- La première ligne rappelle qu'il s'agit d'un modèle de classe.
- Le nom du constructeur est précédé du nom de la classe suivi des paramètres du modèle de classe.

Modèles de classe et méthode : Exemple 3

Définition des méthodes

```
template<typename T,int maxsize>
int CDataArray<T,maxsize>::getSize(){
return length;
}

template<typename T,int maxsize>
T CDataArray<T,maxsize>::get(int index){
 return array[index];
}

template<typename T,int maxsize>
bool CDataArray<T,maxsize>::add(T element){
 if (length>=maxsize) return false;
 array[length++]=element;
 return true;
}
```

Les méthodes sont :

- génériques
- préfixées des paramètres de généricité le la classe

Modèles de classe et méthode : Exemple 3

Utilisation

```
CDataArray<int,100> listeNumeros;

listeNumeros.add(10);
listeNumeros.add(15);

cout << listeNumeros.getSize() << endl;
```

Instanciation

Le compilateur génère une classe pour chaque ensemble de valeurs de paramètres d'instanciation différent.

`./templateCArray`

Modèles de classe et méthode : Exemple 3

Utilisation

```
CDataArray<int,100> listeNumeros;

listeNumeros.add(10);
listeNumeros.add(15);

cout << listeNumeros.getSize() << endl;
```

Instanciation

Le compilateur génère une classe pour chaque ensemble de valeurs de paramètres d'instanciation différent.

./templateCArray

Modèles de classe et méthode : Exemple 3

Utilisation

```
CDataArray<int,100> listeNumeros;

listeNumeros.add(10);
listeNumeros.add(15);

cout << listeNumeros.getSize() << endl;
```

Instanciation

Le compilateur génère une classe pour chaque ensemble de valeurs de paramètres d'instanciation différent.

`./templateCArray`

Modèles de classe et méthode : Amitiés

Classe amie de classe générique

- Une classe générique peut être amie d'une autre classe
- Toute les classes instances de la classe générique seront amies

Fonction amie d'une classe générique

- On peut définir une fonction amie dans un modèle de classe
- La fonction amie sera une amie de toutes les instances du modèle.

Exemple

```
template <typename T> class Essai{
  T x;
public :
  friend class A;//la classe A est amie de toute instance du modele
      Essai
  friend int f(float);//la fonction f est amie de toute instance du
      modele Essai...};
```

Modèles de classe et méthode : héritage

Classe mère générique

Une classe peut hériter d'une classe générique, il faut alors préciser les paramètres de généricité de la classe mère

```
class B : public A<int>
//la classe B derive de la classe A <int>
```

Classe fille générique

Une classe fille peut être générique indépendamment de la généricité de la classe mère

```
template <typename T> class B : public A
//la classe B<T> derive de la classe A
```

Section 2

Exceptions

Exceptions

Définition

Une exception est l'interruption de l'exécution du programme à la suite d'un événement particulier.

Objectif

Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause.
Ces traitements peuvent

- rétablir le programme dans son mode de fonctionnement normal et reprendre l'exécution
- terminer le programme si aucun traitement n'est approprié

Exceptions

Définition

Une exception est l'interruption de l'exécution du programme à la suite d'un événement particulier.

Objectif

Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause.
Ces traitements peuvent

- rétablir le programme dans son mode de fonctionnement normal et reprendre l'exécution
- terminer le programme si aucun traitement n'est approprié

Exceptions en C++

Gestionnaire d'exceptions

Le C++ permet la gestion des exceptions pour les erreurs survenues à l'exécution des programmes.
Le déroulement est le suivant :

1. une erreur à l'exécution du programme
2. lancement d'une exception par le programme
3. interruption de l'exécution normale
4. le gestionnaire d'exception s'exécute

Avantages

- une gestion des erreurs simplifiée par le report de traitement
- simplification du code : les cas particuliers traités par le gestionnaire d'exception

Exceptions en C++

Gestionnaire d'exceptions

Le C++ permet la gestion des exceptions pour les erreurs survenues à l'exécution des programmes.
Le déroulement est le suivant :

1. une erreur à l'exécution du programme
2. lancement d'une exception par le programme
3. interruption de l'exécution normale
4. le gestionnaire d'exception s'exécute

Avantages

- une gestion des erreurs simplifiée par le report de traitement
- simplification du code : les cas particuliers traités par le gestionnaire d'exception

Exceptions en C++

Gestionnaire d'exceptions

Le C++ permet la gestion des exceptions pour les erreurs survenues à l'exécution des programmes.
Le déroulement est le suivant :

1. une erreur à l'exécution du programme
2. lancement d'une exception par le programme
3. interruption de l'exécution normale
4. le gestionnaire d'exception s'exécute

Avantages

- une gestion des erreurs simplifiée par le report de traitement
- simplification du code : les cas particuliers traités par le gestionnaire d'exception

Exceptions en C++

Gestionnaire d'exceptions

Le C++ permet la gestion des exceptions pour les erreurs survenues à l'exécution des programmes.
Le déroulement est le suivant :

1. une erreur à l'exécution du programme
2. lancement d'une exception par le programme
3. interruption de l'exécution normale
4. le gestionnaire d'exception s'exécute

Avantages

- une gestion des erreurs simplifiée par le report de traitement
- simplification du code : les cas particuliers traités par le gestionnaire d'exception

Exceptions en C++

Gestionnaire d'exceptions

Le C++ permet la gestion des exceptions pour les erreurs survenues à l'exécution des programmes.
Le déroulement est le suivant :

1. une erreur à l'exécution du programme
2. lancement d'une exception par le programme
3. interruption de l'exécution normale
4. le gestionnaire d'exception s'exécute

Avantages

- une gestion des erreurs simplifiée par le report de traitement
- simplification du code : les cas particuliers traités par le gestionnaire d'exception

Exceptions en C++

Gestionnaire d'exceptions

Les actions possibles sont :

1. Lancer une exception
2. Attraper une exception
3. Attraper toutes les exceptions
4. la déclaration d'exceptions lancées

Exceptions en C++

Gestionnaire d'exceptions

Les actions possibles sont :

1. Lancer une exception
2. Attraper une exception
3. Attraper toutes les exceptions
4. la déclaration d'exceptions lancées

Exceptions en C++

Gestionnaire d'exceptions

Les actions possibles sont :

1. Lancer une exception
2. Attraper une exception
3. Attraper toutes les exceptions
4. la déclaration d'exceptions lancées

Exceptions en C++

Gestionnaire d'exceptions

Les actions possibles sont :

1. Lancer une exception
2. Attraper une exception
3. Attraper toutes les exceptions
4. la déclaration d'exceptions lancées

Exceptions en C++

Déroulement

1. Construction d'une exception par la fonction qui détecte une anomalie
2. Lancement (`throw`) de l'exception vers la fonction appelante
3. L'exception traverse toutes les fonctions appelantes jusqu'à se faire attraper
   - les fonctions traversées sont immédiatement terminées
   - abandon des instructions restantes
   - destruction des objets locaux
4. Si une exception n'est pas attrapée, il y a terminaison du programme

Exceptions en C++

Déroulement

1. Construction d'une exception par la fonction qui détecte une anomalie
2. Lancement (`throw`) de l'exception vers la fonction appelante
3. L'exception traverse toutes les fonctions appelantes jusqu'à se faire attraper
   - les fonctions traversées sont immédiatement terminées
   - abandon des instructions restantes
   - destruction des objets locaux
4. Si une exception n'est pas attrapée, il y a terminaison du programme

Exceptions en C++

Déroulement

1. Construction d'une exception par la fonction qui détecte une anomalie
2. Lancement (`throw`) de l'exception vers la fonction appelante
3. L'exception traverse toutes les fonctions appelantes jusqu'à se faire attraper
   - les fonctions traversées sont immédiatement terminées
   - abandon des instructions restantes
   - destruction des objets locaux
4. Si une exception n'est pas attrapée, il y a terminaison du programme

Exceptions en C++

Déroulement

1. Construction d'une exception par la fonction qui détecte une anomalie
2. Lancement (`throw`) de l'exception vers la fonction appelante
3. L'exception traverse toutes les fonctions appelantes jusqu'à se faire attraper
   - les fonctions traversées sont immédiatement terminées
   - abandon des instructions restantes
   - destruction des objets locaux
4. Si une exception n'est pas attrapée, il y a terminaison du programme

Exceptions en C++

Déroulement

1. Construction d'une exception par la fonction qui détecte une anomalie
2. Lancement (`throw`) de l'exception vers la fonction appelante
3. L'exception traverse toutes les fonctions appelantes jusqu'à se faire attraper
   - les fonctions traversées sont immédiatement terminées
   - abandon des instructions restantes
   - destruction des objets locaux
4. Si une exception n'est pas attrapée, il y a terminaison du programme

Exceptions en C++

Déroulement

1. Construction d'une exception par la fonction qui détecte une anomalie
2. Lancement (`throw`) de l'exception vers la fonction appelante
3. L'exception traverse toutes les fonctions appelantes jusqu'à se faire attraper
   - les fonctions traversées sont immédiatement terminées
   - abandon des instructions restantes
   - destruction des objets locaux
4. Si une exception n'est pas attrapée, il y a terminaison du programme

Exceptions en C++

Déroulement

1. Construction d'une exception par la fonction qui détecte une anomalie
2. Lancement (`throw`) de l'exception vers la fonction appelante
3. L'exception traverse toutes les fonctions appelantes jusqu'à se faire attraper
   - les fonctions traversées sont immédiatement terminées
   - abandon des instructions restantes
   - destruction des objets locaux
4. Si une exception n'est pas attrapée, il y a terminaison du programme

Lancement d'une exception

L'instruction `throw`

- Syntaxe :

```
throw exp;
```

- Sémantique : lancement de l'exception
  - `exp` est une expression qui désigne l'exception (message, code, objet exception)
  - le type de `exp` peut être quelconque (int, char*, MyExceptionClass, ...)
  - la valeur de `exp` décrit l'exception produite

Attraper et gérer une exception

Syntaxe : les instructions try et catch

```
try {//instructions suceptibles de provoquer le lancement d'une
    exception (directement ou dans les fonctions appelees)
  }
catch (declarationParam1){
  //instructions pour traiter les exceptions correspondant au type de
      Param1
  }
catch (declarationParam2){
  //instructions pour traiter les exceptions, non attrapees par le
      gestionnaire precedent et correspondant au type de Param2
  }
...
catch(...){
  //instructions pour traiter les exceptions non encore attrapees par
      les gestionnaires precedents
  }
```

Attraper et gérer une exception

Principe

Pour attraper une exception, il faut qu'un bloc encadre l'instruction directement, ou indirectement, dans la fonction même ou dans la fonction appelante, ou à un niveau supérieur. Dans le cas contraire, le système récupère l'exception et met fin au programme.

Règles

1. un bloc `try` doit être suivi par au moins un gestionnaire d'exception (`catch`)
2. un bloc `catch` suit un bloc `try` ou un bloc `catch`
3. `catch(...)` est optionnel, s'il existe c'est le dernier

Lancer et Attraper une exception : exemple 1

La division par 0 !

```
int division(int a, int b)
{
    if (b == 0)
    {
   // division par zero
        throw 0;
    }
    else return a / b;
}
```

```
int main(){
    try {
        cout << "1/0 = " <<
            division(1, 0) << endl;
    }
    catch (int code)
    {
        cerr << "Exception " <<
            code << endl;
    }
    return 0;
}
```

./division0

Lancer d'exceptions : exemple 2

Lancement

```
enum VectEr {ErCreation,ErLimite};

class Vect{
private :
  int nb;
  int* ad;
public :
  Vect(int) throw(VectEr);
  ~Vect();
  int& operator[](int) throw(VectEr
      );
  void Affic(){
    int i;
    for(i=0;i<nb;i++) cout << ad[i]
        << " ";
    cout << endl;}
};
```

Définition

```
Vect::Vect(int n) throw (VectEr){
  int i;
  if (n<0) throw ErCreation;
  else {
    ad = new int[nb=n];
    for(i=0;i<nb;i++) ad[i] = 0;}
    }
Vect::~Vect(){ delete [] ad;}
int& Vect::operator[](int i) throw
    (VectEr) {
  if ((i<0) || (i>nb)) throw
      ErLimite;
  else
    return ad[i];
    }
```

Attraper et gérer une exception : exemple 2

Gestion et capture

```
int main(){
try {
  Vect v1(4);
  v1.Affic();
  v1[11]=5;
  }
catch (VectEr vr){
  if (vr == ErCreation)
    cerr << "Mauvais nombre
        elements";
  else
    cerr << "Mauvais indice";
    }
return 0;
}
```

./exceptionVect

Exécution

```
0000
Mauvais indice
```

Section 3

Héritage multiple

Héritage multiple

C++

Le C++ autorise l'héritage multiple

Syntaxe

```
class A1{};// classe mere 1
class A2{};// classe mere 2
...
class An{};// classe mere n
class B : <type heritage> : A1, <type heritage> : A2, ..., <type
    heritage> : An
{
//declaration de la classe fille B
};
```

Héritage multiple

C++

Le C++ autorise l'héritage multiple

Syntaxe

```
class A1{};// classe mere 1
class A2{};// classe mere 2
...
class An{};// classe mere n
class B : <type heritage> : A1, <type heritage> : A2, ..., <type
    heritage> : An
{
//declaration de la classe fille B
};
```

Héritage multiple

Sémantique

- Un objet `B` est aussi un objet `A1`, un objet `A2`, ..., un objet `An`
- Il possède tous les attributs des classes `A1`,`A2`, ...,`An`, plus ses attributs spécifiques

Construction

Lors de la construction d'une instance de la classe `B`, les constructeurs appelés sont : celui de `A1`, puis celui de `A2`, ... et enfin celui de `An`.

Destruction

Lors de la destruction d'une instance de la classe `B`, les destructeurs sont appelés dans l'ordre inverse de la construction : celui de `An`, ...,puis celui de `A2` et enfin celui de `A1`.

Héritage multiple

Sémantique

- Un objet `B` est aussi un objet `A1`, un objet `A2`, ..., un objet `An`
- Il possède tous les attributs des classes `A1`,`A2`, ...,`An`, plus ses attributs spécifiques

Construction

Lors de la construction d'une instance de la classe `B`, les constructeurs appelés sont : celui de `A1`, puis celui de `A2`, ... et enfin celui de `An`.

Destruction

Lors de la destruction d'une instance de la classe `B`, les destructeurs sont appelés dans l'ordre inverse de la construction : celui de `An`, ...,puis celui de `A2` et enfin celui de `A1`.

Héritage multiple

Sémantique

- Un objet `B` est aussi un objet `A1`, un objet `A2`, ..., un objet `An`
- Il possède tous les attributs des classes `A1`,`A2`, ...,`An`, plus ses attributs spécifiques

Construction

Lors de la construction d'une instance de la classe `B`, les constructeurs appelés sont : celui de `A1`, puis celui de `A2`, ... et enfin celui de `An`.

Destruction

Lors de la destruction d'une instance de la classe `B`, les destructeurs sont appelés dans l'ordre inverse de la construction : celui de `An`, ...,puis celui de `A2` et enfin celui de `A1`.

Héritage multiple : exemple

Classe mère #1

```
class Point{
private:
  int x, y;
public :
  Point(int abs,int ord)
    {x = abs; y = ord; cout << "
        ConsPoint : "<<endl;}
  ~Point( )
    {cout << "DesPoint : \n";}
  void Affic( )
    {cout << "CoordPoint : " << x
        << " " << y << endl;}
    };
```

Classe mère #2

```
class Coul{
private :
  short couleur;
public :
  Coul(short cl)
    {couleur = cl; cout << "
        ConsCoul :"<<endl;}
  ~Coul( )
    {cout << "DesCoul :"<< endl;}
  void Affic( )
    {cout << "Couleur : " <<
        couleur <<endl;}
    };
```

Héritage multiple : exemple

Classe fille

```
class PointCoul : public Point, public Coul {
// pas d'attribut pecifique
public :
  PointCoul(int,int,short);
  ~PointCoul(){cout << "DesPointCoul :" << endl;}
  void Affic(){Point::Affic(); Coul::Affic();}
};

PointCoul::PointCoul(int abs,int ord,short cl): Point(abs,ord), Coul(cl)
{
cout << "ConsPointCoul :"<<endl;
}
```

Héritage multiple : exemple

Création d'une instance de la classe PointCoul

```
int main(){
  PointCoul p(3,9,2);//Appel const : ConsPoint, ConsCoul puis
      ConsPointCoul
  p.Affic( );//CoordPoint 3 9 puis Couleur 2 sur la ligne suivante
  p.Point::Affic( );//CoordPoint 3 9
  p.Coul::Affic( );//Couleur 2
  return 0;} //DesPointCoul puis DesCoul puis DesPoint
```

./multiHeritagePoint

Exécution

```
ConsPoint : ConsCoul :ConsPointCoul :
CoordPoint : 3 9
Couleur : 2
CoordPoint : 3 9
Couleur : 2
DesPointCoul :DesCoul : DesPoint :
```

Héritage multiple : exemple

Création d'une instance de la classe PointCoul

```
int main(){
  PointCoul p(3,9,2);//Appel const : ConsPoint, ConsCoul puis
      ConsPointCoul
  p.Affic( );//CoordPoint 3 9 puis Couleur 2 sur la ligne suivante
  p.Point::Affic( );//CoordPoint 3 9
  p.Coul::Affic( );//Couleur 2
  return 0;} //DesPointCoul puis DesCoul puis DesPoint
```

./multiHeritagePoint

Exécution

```
ConsPoint : ConsCoul :ConsPointCoul :
CoordPoint : 3 9
Couleur : 2
CoordPoint : 3 9
Couleur : 2
DesPointCoul :DesCoul : DesPoint :
```

Héritage multiple : exemple

Création d'une instance de la classe PointCoul

```
int main(){
  PointCoul p(3,9,2);//Appel const : ConsPoint, ConsCoul puis
      ConsPointCoul
  p.Affic( );//CoordPoint 3 9 puis Couleur 2 sur la ligne suivante
  p.Point::Affic( );//CoordPoint 3 9
  p.Coul::Affic( );//Couleur 2
  return 0;} //DesPointCoul puis DesCoul puis DesPoint
```

./multiHeritagePoint

Exécution

```
ConsPoint : ConsCoul :ConsPointCoul :
CoordPoint : 3 9
Couleur : 2
CoordPoint : 3 9
Couleur : 2
DesPointCoul :DesCoul : DesPoint :
```

Héritage multiple : conflits

Héritage en chaîne

```
class Base{
//classe grand-mere
int x,y;
};
```

```
class H1: public Base{
//classe fille 1
};
```

```
class H2: public Base{
//classe fille 2
};
```

```
class Finale: public H1, public H2 {
//classe petite fille
};
```

Héritage multiple : conflit

Conflit

Un objet de la classe `Finale` hérite deux fois de la classe `Base`

- les instanciations des objets font faire des appels successifs aux contsructeurs :
  1. le constructeur `Finale::Finale()` appelle les constructeurs `H1::H1()` et `H2::H2()`
  2. le constructeur `H1::H1()` appelle le constructeur `Base::Base()`
  3. le constructeur `H2::H2()` appelle le constructeur `Base::Base()`
- les attributs `x,y` sont dupliqués par héritage de la classe `H1` et `H2`, accès différencié par les espaces de noms `Base::H1::x` et `Base::H2::x`
- le problème de conflit pour les constructeurs n'existe pas pour les autres méthodes

Héritage multiple : conflit

Conflit

Un objet de la classe `Finale` hérite deux fois de la classe `Base`

- les instanciations des objets font faire des appels successifs aux contsructeurs :
  1. le constructeur `Finale::Finale()` appelle les constructeurs `H1::H1()` et `H2::H2()`
  2. le constructeur `H1::H1()` appelle le constructeur `Base::Base()`
  3. le constructeur `H2::H2()` appelle le constructeur `Base::Base()`
- les attributs `x,y` sont dupliqués par héritage de la classe `H1` et `H2`, accès différencié par les espaces de noms `Base::H1::x` et `Base::H2::x`
- le problème de conflit pour les constructeurs n'existe pas pour les autres méthodes

Héritage multiple : conflit

Conflit

Un objet de la classe `Finale` hérite deux fois de la classe `Base`

- les instanciations des objets font faire des appels successifs aux contsructeurs :
  1. le constructeur `Finale::Finale()` appelle les constructeurs `H1::H1()` et `H2::H2()`
  2. le constructeur `H1::H1()` appelle le constructeur `Base::Base()`
  3. le constructeur `H2::H2()` appelle le constructeur `Base::Base()`
- les attributs `x,y` sont dupliqués par héritage de la classe `H1` et `H2`, accès différencié par les espaces de noms `Base::H1::x` et `Base::H2::x`
- le problème de conflit pour les constructeurs n'existe pas pour les autres méthodes

Héritage multiple : conflit

Conflit

Un objet de la classe `Finale` hérite deux fois de la classe `Base`

- les instanciations des objets font faire des appels successifs aux contsructeurs :
  1. le constructeur `Finale::Finale()` appelle les constructeurs `H1::H1()` et `H2::H2()`
  2. le constructeur `H1::H1()` appelle le constructeur `Base::Base()`
  3. le constructeur `H2::H2()` appelle le constructeur `Base::Base()`
- les attributs `x,y` sont dupliqués par héritage de la classe `H1` et `H2`, accès différencié par les espaces de noms `Base::H1::x` et `Base::H2::x`
- le problème de conflit pour les constructeurs n'existe pas pour les autres méthodes

Héritage multiple : conflit

Conflit

Un objet de la classe `Finale` hérite deux fois de la classe `Base`

- les instanciations des objets font faire des appels successifs aux contsructeurs :
  1. le constructeur `Finale::Finale()` appelle les constructeurs `H1::H1()` et `H2::H2()`
  2. le constructeur `H1::H1()` appelle le constructeur `Base::Base()`
  3. le constructeur `H2::H2()` appelle le constructeur `Base::Base()`
- les attributs `x,y` sont dupliqués par héritage de la classe `H1` et `H2`, accès différencié par les espaces de noms `Base::H1::x` et `Base::H2::x`
- le problème de conflit pour les constructeurs n'existe pas pour les autres méthodes

Héritage multiple : conflit

Conflit

Un objet de la classe `Finale` hérite deux fois de la classe `Base`

- les instanciations des objets font faire des appels successifs aux contsructeurs :
  1. le constructeur `Finale::Finale()` appelle les constructeurs `H1::H1()` et `H2::H2()`
  2. le constructeur `H1::H1()` appelle le constructeur `Base::Base()`
  3. le constructeur `H2::H2()` appelle le constructeur `Base::Base()`
- les attributs `x,y` sont dupliqués par héritage de la classe `H1` et `H2`, accès différencié par les espaces de noms `Base::H1::x` et `Base::H2::x`
- le problème de conflit pour les constructeurs n'existe pas pour les autres méthodes

Héritage : conflit

Instance de la classe finale

```
int main( )
{
Finale F;
return 0;
}
```

./HeritageConflit

Exécution

```
+++ Construction de Base
+++ Construction de H1
+++ Construction de Base
+++ Construction de H2
+++ Construction de Finale
--- Destruction de Finale
--- Destruction de H2
--- Destruction de Base
--- Destruction de H1
--- Destruction de Base
```

Héritage virtuel

L'héritage virtuel permet d'éviter les conflits (attributs dupliqués et appel des constructeurs successifs).
Les constructeurs multi-hérités ne sont appelés qu'une seule fois.

Héritage : conflit

Instance de la classe finale

```
int main( )
{
Finale F;
return 0;
}
```

./HeritageConflit

Exécution

```
+++ Construction de Base
+++ Construction de H1
+++ Construction de Base
+++ Construction de H2
+++ Construction de Finale
--- Destruction de Finale
--- Destruction de H2
--- Destruction de Base
--- Destruction de H1
--- Destruction de Base
```

Héritage virtuel

L'héritage virtuel permet d'éviter les conflits (attributs dupliqués et appel des constructeurs successifs).
Les constructeurs multi-hérités ne sont appelés qu'une seule fois.

Héritage : conflit

Instance de la classe finale

```
int main( )
{
Finale F;
return 0;
}
```

./HeritageConflit

Exécution

```
+++ Construction de Base
+++ Construction de H1
+++ Construction de Base
+++ Construction de H2
+++ Construction de Finale
--- Destruction de Finale
--- Destruction de H2
--- Destruction de Base
--- Destruction de H1
--- Destruction de Base
```

Héritage virtuel

L'héritage virtuel permet d'éviter les conflits (attributs dupliqués et appel des constructeurs successifs).
Les constructeurs multi-hérités ne sont appelés qu'une seule fois.

Héritage virtuel : exemple

```
class Base
{
  Base() {cout << "Construction de Base" << endl;}
  virtual ~Base() {cout << "Destruction de Base" << endl;}
};
```

```
class H1 : virtual public Base {
  H1():Base()
    {cout << "Construction de H1";}
  virtual ~H1()
    { cout << "Destruction de H1";}
};
```

```
class H2 : virtual public Base{
  H2():Base()
    {cout << "Construction de H2";}
  virtual ~H2()
    {cout << "Destruction de H2";}
};
```

```
class Finale : virtual public H1, virtual public H2
{
  Finale() : H1(), H2() {cout << "Construction de Finale" << endl;}
  ~Finale() {cout << "Destruction de Finale" << endl;}
};
```

Héritage virtuel : exemple

Instance de la classe finale

```
int main( )
{
Finale F;
return 0;
}
```

./HeritageVirtuel

Exécution

```
+++ Construction de Base
+++ Construction de H1
+++ Construction de H2
+++ Construction de Finale
--- Destruction de Finale
--- Destruction de H2
--- Destruction de H1
--- Destruction de Base
```

Héritage virtuel : exemple

Instance de la classe finale

```
int main( )
{
Finale F;
return 0;
}
```

./HeritageVirtuel

Exécution

```
+++ Construction de Base
+++ Construction de H1
+++ Construction de H2
+++ Construction de Finale
--- Destruction de Finale
--- Destruction de H2
--- Destruction de H1
--- Destruction de Base
```

Section 4

Bibliothèque STL

La bibliothèque standard de C++

la bibliothèque standard de C++

Cette bibliothèque incluse en totalité dans l'espace de nom `std` comprend

- la bibliothèque standard du C
- la bibliothèque des flux d'entrée-sortie : IO Stream Library
- la bibliothèque STL

La bibliothèque standard de C++

la bibliothèque standard de C++

Cette bibliothèque incluse en totalité dans l'espace de nom `std` comprend

- la bibliothèque standard du C
- la bibliothèque des flux d'entrée-sortie : IO Stream Library
- la bibliothèque STL

La bibliothèque standard de C++

la bibliothèque standard de C++

Cette bibliothèque incluse en totalité dans l'espace de nom `std` comprend

- la bibliothèque standard du C
- la bibliothèque des flux d'entrée-sortie : IO Stream Library
- la bibliothèque STL

Bibliothèque STL

Standard Template Library (STL)

La bibliothèque STL est une est une bibliothèque C++, normalisée par l'ISO (document ISO/CEI 14882) et mise en œuvre à l'aide des templates.

la bibliothèque STL

fournit :

- un ensemble de classes conteneurs
- des itérateurs
- des algorithmes génériques
- une classe string

Bibliothèque STL

Standard Template Library (STL)

La bibliothèque STL est une est une bibliothèque C++, normalisée par l'ISO (document ISO/CEI 14882) et mise en œuvre à l'aide des templates.

la bibliothèque STL

fournit :

- un ensemble de classes conteneurs
- des itérateurs
- des algorithmes génériques
- une classe `string`

Bibliothèque STL

Standard Template Library (STL)

La bibliothèque STL est une est une bibliothèque C++, normalisée par l'ISO (document ISO/CEI 14882) et mise en œuvre à l'aide des templates.

la bibliothèque STL

fournit :

- un ensemble de classes conteneurs
- des itérateurs
- des algorithmes génériques
- une classe `string`

Bibliothèque STL

Standard Template Library (STL)

La bibliothèque STL est une est une bibliothèque C++, normalisée par l'ISO (document ISO/CEI 14882) et mise en œuvre à l'aide des templates.

la bibliothèque STL

fournit :

- un ensemble de classes conteneurs
- des itérateurs
- des algorithmes génériques
- une classe `string`

Les conteneurs

Définition

Les conteneurs sont des objets qui représentent des collections d'objets

Propriétés

Les conteneurs sont des classes qui

- permettent de représenter les structures de données (listes, tableaux, ensembles,...)
- sont dotées de méthodes permettant de
  - créer, copier et détruire ces conteneurs
  - d'insérer, de rechercher ou de supprimer des éléments dans ces conteneurs

Type de données dans les conteneurs

les conteneurs peuvent s'appliquer à tout type de données qui supporte la copie et l'affectation

Les conteneurs

Définition

Les conteneurs sont des objets qui représentent des collections d'objets

Propriétés

Les conteneurs sont des classes qui

- permettent de représenter les structures de données (listes, tableaux, ensembles,...)
- sont dotées de méthodes permettant de
  - créer, copier et détruire ces conteneurs
  - d'insérer, de rechercher ou de supprimer des éléments dans ces conteneurs

Type de données dans les conteneurs

les conteneurs peuvent s'appliquer à tout type de données qui supporte la copie et l'affectation

Les conteneurs

Définition

Les conteneurs sont des objets qui représentent des collections d'objets

Propriétés

Les conteneurs sont des classes qui

- permettent de représenter les structures de données (listes, tableaux, ensembles,...)
- sont dotées de méthodes permettant de
  - créer, copier et détruire ces conteneurs
  - d'insérer, de rechercher ou de supprimer des éléments dans ces conteneurs

Type de données dans les conteneurs

les conteneurs peuvent s'appliquer à tout type de données qui supporte la copie et l'affectation

Les conteneurs

Trois familles de conteneurs

On distingue 3 familles de conteneurs

- les séquences
- les conteneurs associatifs
- les collections de bits

Les conteneurs : les séquences

Les séquences

Ces conteneurs sont des ensembles finis d'objets avec une organisation linéaire :

- les vecteurs `vector` : une classe modèle `std::vector` est un tableau dynamique.
- les tableaux `array` : une classe modèle `std::array` représentant un tableau de taille fixe.
- les listes `list` : une classe modèle `std::list` représentant une liste doublement chaînée.
- les files à double entrée `deque` : une classe modèle `std::deque` est une file sur laquelle on peut retirer et ajouter des éléments sur les deux extrémités

Les conteneurs : les séquences

Les séquences

Ces conteneurs sont des ensembles finis d'objets avec une organisation linéaire :

- les vecteurs `vector` : une classe modèle `std::vector` est un tableau dynamique.
- les tableaux `array` : une classe modèle `std::array` représentant un tableau de taille fixe.
- les listes `list` : une classe modèle `std::list` représentant une liste doublement chaînée.
- les files à double entrée `deque` : une classe modèle `std::deque` est une file sur laquelle on peut retirer et ajouter des éléments sur les deux extrémités

Les conteneurs : les séquences

Les séquences

Ces conteneurs sont des ensembles finis d'objets avec une organisation linéaire :

- les vecteurs `vector` : une classe modèle `std::vector` est un tableau dynamique.
- les tableaux `array` : une classe modèle `std::array` représentant un tableau de taille fixe.
- les listes `list` : une classe modèle `std::list` représentant une liste doublement chaînée.
- les files à double entrée `deque` : une classe modèle `std::deque` est une file sur laquelle on peut retirer et ajouter des éléments sur les deux extrémités

Les conteneurs : les séquences

Les séquences

Ces conteneurs sont des ensembles finis d'objets avec une organisation linéaire :

- les vecteurs `vector` : une classe modèle `std::vector` est un tableau dynamique.
- les tableaux `array` : une classe modèle `std::array` représentant un tableau de taille fixe.
- les listes `list` : une classe modèle `std::list` représentant une liste doublement chaînée.
- les files à double entrée `deque` : une classe modèle `std::deque` est une file sur laquelle on peut retirer et ajouter des éléments sur les deux extrémités

Les conteneurs : les séquences

Les adaptateurs

Ces conteneurs sont des interfaces permettant d'utiliser les séquences

- les piles `stack`
- les files `queue`
- les files de priorité `priority_queue`

Les conteneurs associatifs

Conteneurs associatifs avec clé

Ces conteneurs sont des collections de valeurs dont le moyen d'accès est une clé.

- les tables associatives `map` : la classes modèle `std::map` représentant un tableau associatif
- les tables associatives multiples `multimap` : la classe modèle `std::multimap`

Conteneurs associatifs sans clé

Ces conteneurs sont des collections où valeur et clé sont confondues

- les ensembles `set`
- les ensembles multiples `multiset`

Les conteneurs associatifs

Conteneurs associatifs avec clé

Ces conteneurs sont des collections de valeurs dont le moyen d'accès est une clé.

- les tables associatives `map` : la classes modèle `std::map` représentant un tableau associatif
- les tables associatives multiples `multimap` : la classe modèle `std::multimap`

Conteneurs associatifs sans clé

Ces conteneurs sont des collections où valeur et clé sont confondues

- les ensembles `set`
- les ensembles multiples `multiset`

Les collections de bits

Collections de bits

- les vecteurs de booléen `vector<bool>`
- les ensembles de bits `bitset` : une classe conteneur spécialisée `std::bitset` représentant un tableau de bits.

Les conteneurs : méthodes communes

Exemples de méthodes

```
bool empty() const;
// le conteneur est-il vide ?
size_t size() const;
//nombre d'element du conteneur
size_t max_size() const;
//nombre d'element max du
    conteneur
T& operator[](cle);
const T & operator[](cle) const;
//acces indexe au element du
    conteneur
iterateur insert (iterateur, const
     T & x);
void insert (iterateur, size_t n
    const T & x);
//insertion de x [n copies de x]
```

```
iterateur insert (iterateur, const
     T & x);
void insert (iterateur, size_t n
    const T & x);
//insertion de x [n copies de x]
iterateur erase(iterateur pos);
//suppression de la valeur pointee
     par pos
iterateur begin() const;
const iterateur begin() const;
//construction d'un iterateur
    positionne au debut
iterateur end() const;
const iterateur end() const;
//construction d'un iterateur
    positionne a la fin
```

Les conteneurs : Diagramme [2]

Diagramme des conteneurs

2. wikipedia

Les itérateurs

En génie logiciel

Un itérateur est un objet qui permet de parcourir tous les éléments contenus dans un autre objet, le plus souvent un conteneur (liste, arbre, etc).

En C++

Les itérateurs sont une généralisation des pointeurs. Ils permettent de manipuler les conteneurs de façon uniforme en proposant des méthodes génériques.
Les itérateurs fournissent un moyen simple et élégant de parcourir des séquences d'objets et permettent la description d'algorithmes indépendamment de toute structure de données.
Fichier en-tête `<iterator>`

Les itérateurs : propriétés

Propriétés

- un itérateur (valide), renvoie à un élément courant de la collection atteignable par les opérateurs d'indirection * et ->
- les opérateurs ++ modifie l'itérateur pour renvoyer l'élément suivant
- la relation d'égalité entre deux itérateurs ==

Les itérateurs

5 catégories d'itérateurs

- les itérateurs de sortie (`output iterator`) qui supportent les "écritures" et l'avancement (++)
- les itérateurs d'entrée (`input iterator`) qui supportent la "lecture", l'avancement et l'égalité (== et !=)
- les itérateurs en avant (`forward iterator`) qui supportent la lecture, l'écriture, l'avancement et l'égalité
- les itérateurs bidirectionnels (`bidirectional iterator`) comme les précédents avec en plus le recul (-)
- les itérateurs d'accès aléatoire (`random access iterator`) qui possèdent les propriétés des itérateurs bidirectionnels avec en plus l'accès avec l'opérateur d'indexation ([]), l'addition et soustraction d'un entier, la soustraction de deux itérateurs et une relation d'ordre (<,>,<=,>=)

Les itérateurs

5 catégories d'itérateurs

- les itérateurs de sortie (`output iterator`) qui supportent les "écritures" et l'avancement (++)
- les itérateurs d'entrée (`input iterator`) qui supportent la "lecture", l'avancement et l'égalité (== et !=)
- les itérateurs en avant (`forward iterator`) qui supportent la lecture, l'écriture, l'avancement et l'égalité
- les itérateurs bidirectionnels (`bidirectional iterator`) comme les précédents avec en plus le recul (-)
- les itérateurs d'accès aléatoire (`random access iterator`) qui possèdent les propriétés des itérateurs bidirectionnels avec en plus l'accès avec l'opérateur d'indexation ([]), l'addition et soustraction d'un entier, la soustraction de deux itérateurs et une relation d'ordre (<,>,<=,>=)

Les itérateurs

5 catégories d'itérateurs

- les itérateurs de sortie (`output iterator`) qui supportent les "écritures" et l'avancement (++)
- les itérateurs d'entrée (`input iterator`) qui supportent la "lecture", l'avancement et l'égalité (== et !=)
- les itérateurs en avant (`forward iterator`) qui supportent la lecture, l'écriture, l'avancement et l'égalité
- les itérateurs bidirectionnels (`bidirectional iterator`) comme les précédents avec en plus le recul (-)
- les itérateurs d'accès aléatoire (`random access iterator`) qui possèdent les propriétés des itérateurs bidirectionnels avec en plus l'accès avec l'opérateur d'indexation ([]), l'addition et soustraction d'un entier, la soustraction de deux itérateurs et une relation d'ordre (<,>,<=,>=)

Les itérateurs

5 catégories d'itérateurs

- les itérateurs de sortie (`output iterator`) qui supportent les "écritures" et l'avancement (++)
- les itérateurs d'entrée (`input iterator`) qui supportent la "lecture", l'avancement et l'égalité (== et !=)
- les itérateurs en avant (`forward iterator`) qui supportent la lecture, l'écriture, l'avancement et l'égalité
- les itérateurs bidirectionnels (`bidirectional iterator`) comme les précédents avec en plus le recul (-)
- les itérateurs d'accès aléatoire (`random access iterator`) qui possèdent les propriétés des itérateurs bidirectionnels avec en plus l'accès avec l'opérateur d'indexation ([]), l'addition et soustraction d'un entier, la soustraction de deux itérateurs et une relation d'ordre (<,>,<=,>=)

Les itérateurs

5 catégories d'itérateurs

- les itérateurs de sortie (`output iterator`) qui supportent les "écritures" et l'avancement (++)
- les itérateurs d'entrée (`input iterator`) qui supportent la "lecture", l'avancement et l'égalité (== et !=)
- les itérateurs en avant (`forward iterator`) qui supportent la lecture, l'écriture, l'avancement et l'égalité
- les itérateurs bidirectionnels (`bidirectional iterator`) comme les précédents avec en plus le recul (-)
- les itérateurs d'accès aléatoire (`random access iterator`) qui possèdent les propriétés des itérateurs bidirectionnels avec en plus l'accès avec l'opérateur d'indexation (`[]`), l'addition et soustraction d'un entier, la soustraction de deux itérateurs et une relation d'ordre (<,>,<=,>=)

Les algorithmes génériques

Algorithmes dans STL

La bibliothèque STL fournit des algorithmes

- d'insertion/suppression,
- de recherche
- de tri

Fichier en-tête : <algorithm>

Propriétés

Les algorithmes sont indépendants des conteneurs et les manipulent grâce aux itérateurs

Algorithmes

Exemples d'algorithmes

- for_each : appel de la fonction `f` sur chaque élément de la séquence

```
template<class IterIn, Class Fonc>
Fonc for_each(IterIn debut, IterIn fin, Fonc f)
```

- find : recherche du premier élément de la séquence égal à la `valeur` indiquée

```
template<class IterIn, Class T>
IterIn find(IterIn debut, IterIn fin, const T & valeur)
```

- fill : tous les éléments de la séquences sont affectés de la valeur indiquée (`valeur`), le type `T` doit supporter affectation

```
template<class IterAv, Class T>
void fill(IterAv debut, IterAv fin, const T & valeur)
```

Exemple [3]

Conteneur, itérateur, algorithme

Recherche de la première occurrence de la chaîne "C++" dans une liste de chaînes.

```
list<string> l; //conteneur liste de chaine
list<string>::iterator p;// iterator
l.push_back("Le dernier");//ajour d'element en queue de conteneur
l.push_back("cours de");
l.push_back("C++");
for(p=l.begin();p!=l.end();p++) //affichage de la liste
   cout<<*p<<endl;
p=find(l.begin(), l.end(), (string)"C++");// algorithme de recherche
    find
if (p==l.end())
  cout<<"Element absent"<<endl;
else
  cout<<"Element trouve"<<endl;
return 0;
```

3. Le langage et la bibliothèque C++, Henri Garreta

Exemple

`./IteratorChaine`


End of Story