C++11, Template et STL

Quelques rappels

Pierre Ramet

pierre.ramet@inria.fr

Stage Programmation Licence, 2019

IUT Bordeaux

Plan de l'exposé

Plan de l'exposé

Classes génériques

Les classes *templates* permettent de paramétrer la définition d'une classe avec des types de données.

Exemples en C++

```
{ ...
  Pile<float> ma_pile;
  Liste<Piece> ma_liste;
}
```

C'est à l'**instanciation** et non à la définition qu'une classe générique connait les types qui la paramètrent.

Prix

Le prix à payer est la syntaxe assez lourde (parfois illisible) des classes *templates*.

Classes génériques

Les classes *templates* permettent de paramétrer la définition d'une classe avec des types de données.

Exemples en C++

```
{ ...
  Pile<float> ma_pile;
  Liste<Piece> ma_liste;
}
```

C'est à l'**instanciation** et non à la définition qu'une classe générique connait les types qui la paramètrent.

Prix

Le prix à payer est la syntaxe assez lourde (parfois illisible) des classes *templates*.

Classes génériques

Les classes *templates* permettent de paramétrer la définition d'une classe avec des types de données.

Exemples en C++

```
{ ...
  Pile<float> ma_pile;
  Liste<Piece> ma_liste;
}
```

C'est à l'**instanciation** et non à la définition qu'une classe générique connait les types qui la paramètrent.

Prix

Le prix à payer est la syntaxe assez lourde (parfois illisible) des classes *templates*.

Classes génériques

Les classes *templates* permettent de paramétrer la définition d'une classe avec des types de données.

Exemples en C++

```
{ ...
  Pile<float> ma_pile;
  Liste<Piece> ma_liste;
}
```

C'est à l'**instanciation** et non à la définition qu'une classe générique connait les types qui la paramètrent.

Prix

Le prix à payer est la syntaxe assez lourde (parfois illisible) des classes *templates*.

Déclaration

- On fait précéder la déclaration de la classe de `template<class X>` et ensuite on utilisera `X` comme un identificateur de type quelconque.
- On énumère les différentes instanciations prévisibles. Cette méthode présente l'inconvénient de forcer à prévoir ces instanciations.

méthodes `inline`

Si on veut éviter cet inconvénient, on peut définir directement les méthodes dans le fichier d'en-tête.

Déclaration

- On fait précéder la déclaration de la classe de `template<class X>` et ensuite on utilisera `X` comme un identificateur de type quelconque.
- On énumère les différentes instanciations prévisibles. Cette méthode présente l'inconvénient de forcer à prévoir ces instanciations.

méthodes `inline`

Si on veut éviter cet inconvénient, on peut définir directement les méthodes dans le fichier d'en-tête.

Déclaration

- On fait précéder la déclaration de la classe de `template<class X>` et ensuite on utilisera `X` comme un identificateur de type quelconque.
- On énumère les différentes instanciations prévisibles. Cette méthode présente l'inconvénient de forcer à prévoir ces instanciations.

méthodes `inline`

Si on veut éviter cet inconvénient, on peut définir directement les méthodes dans le fichier d'en-tête.

Exercice

Exercice 0.

En partant de la classe `TabEntier`, écrivez une version générique. Ne pas oublier de prévoir les instanciations :

```
template class TabGeneric<int>;
template class TabGeneric<double>;
template class TabGeneric<char>;
```

Exercice 1.

Etendre la classe générique `TabGeneric` avec un second paramètre pour préciser le type de l'indice. Par exemple :

```
TabGeneric<long,float> tableau(nbelem);
```

Plan de l'exposé

Objectif de la STL

- Offrir les structures de données et opérations de bases pour que des bibliothèques différentes puissent communiquer ou échanger des données entre elles.
- Propose une implémentation générique et efficace des types abstraits de données classiques (piles, files, listes, vecteurs, etc.).
- Fournir un ensemble d'algorithmes pour effectuer des opérations fréquentes sur ces types de données (fusion, tri, parcours, etc.).

Objectif de la STL

- Offrir les structures de données et opérations de bases pour que des bibliothèques différentes puissent communiquer ou échanger des données entre elles.
- Propose une implémentation générique et efficace des types abstraits de données classiques (piles, files, listes, vecteurs, etc.).
- Fournir un ensemble d'algorithmes pour effectuer des opérations fréquentes sur ces types de données (fusion, tri, parcours, etc.).

Objectif de la STL

- Offrir les structures de données et opérations de bases pour que des bibliothèques différentes puissent communiquer ou échanger des données entre elles.
- Propose une implémentation générique et efficace des types abstraits de données classiques (piles, files, listes, vecteurs, etc.).
- Fournir un ensemble d'algorithmes pour effectuer des opérations fréquentes sur ces types de données (fusion, tri, parcours, etc.).

Les conteneurs

Définition

- Un conteneur est un objet qui en contient d'autres.
- En général, il est possible d'ajouter ou de supprimer des objets d'un conteneur.

Principe

- Stocker des objets d'une certaine manière.
- Proposer des moyens d'accès spécifiques à l'utilisateur pour accéder/ajouter/mettre à jour ces données.

Ainsi, on se sert de la même manière d'une liste d'entiers ou d'une liste de chaînes de caractères.

Les conteneurs

Définition

- Un conteneur est un objet qui en contient d'autres.
- En général, il est possible d'ajouter ou de supprimer des objets d'un conteneur.

Principe

- Stocker des objets d'une certaine manière.
- Proposer des moyens d'accès spécifiques à l'utilisateur pour accéder/ajouter/mettre à jour ces données.

Ainsi, on se sert de la même manière d'une liste d'entiers ou d'une liste de chaînes de caractères.

Mise en œuvre : les classes génériques

Les conteneurs de la STL sont des classes templates.

Conteneurs de la STL (1)

Chaque conteneur prend le type **T** en paramètre à l'instanciation.

Conteneurs	descriptif	entête
`vector`	Tableau de **T** à une dimension	`#include <vector>`
`list`	Liste doublement chaînée de **T**	`#include <list>`
`deque`	File d'attente à double accès de **T**	`#include <deque>`
`queue`	File d'attente de **T**	`#include <queue>`
`set`	Ensemble de **T**	`#include <set>`
`hash_set`	Ensemble de **T** (autre implémentation)	`#include <hash_set>`

Conteneurs de la STL (2)

Il existe d'autres conteneurs qui prennent deux types **T** et **U** en paramètres :

Conteneurs	descriptif	entête
`map`	Tableau associatif entre un **T** et un **U**	`#include <map>`
`hash_map`	Tableau associatif entre un **T** et un **U** (autre implémentation)	`#include <hash_map>`
`multimap`	Tableau associatif entre un **T** et plusieurs **U**	`#include <map>`

Conteneurs de la STL (3)

Il existe des presque-conteneurs qui ne prennent pas de paramètres :

Conteneurs	descriptif	entête
`bitset`	Tableau de booléens	`#include <bitset>`
`string`	Chaînes de caractères	`#include <string>`

Règles sur les objets mis dans les conteneurs

Un conteneur ne peut pas contenir des types complètement arbitraires.

Le type T offre les opérations suivantes :

- constructeur par défaut,
- constructeur par copie,
- opérateur d'affectation.

Pour certains conteneurs (ex. `map`) ou algorithmes (ex. tris), les opérateurs de comparaison == et < doivent aussi être fournis.

Règles sur les objets mis dans les conteneurs

Un conteneur ne peut pas contenir des types complètement arbitraires.

Le type T offre les opérations suivantes :

- constructeur par défaut,
- constructeur par copie,
- opérateur d'affectation.

Pour certains conteneurs (ex. `map`) ou algorithmes (ex. tris), les opérateurs de comparaison == et < doivent aussi être fournis.

Les itérateurs

- Les itérateurs forment un mécanisme pour parcourir les conteneurs.
- Les itérateurs sont génériques et communs à tous les conteneurs.

Différents types de parcours :

- parcours dans l'ordre naturel,
- parcours en sens inverse,
- parcours dans les deux sens,
- accès direct ou aléatoire (similaire à l'indice d'un tableau),
- les itérateurs en lecture seule.

Les algorithmes

La STL vous fournit un certain nombre d'algorithmes pour manipuler des conteneurs. Ces algorithmes fonctionnent en utilisant des itérateurs définis sur ces conteneurs.
Il faut inclure l'entête `<algorithm>` pour se servir de la majorité des algorithmes.

Plan de l'exposé

La classe générique `stack`

Modélise une pile d'éléments.

ex_stack.cxx

```
#include <stack>
#include <iostream.h>
int main() {
  // Instanciation d'une pile d'entiers
  stack<int> pile;
  // Pile remplie avec les puissances de 2
  for ( int i = 1; i < 1000; i = i * 2 )
    // Empile la valeur de 'i'
    pile.push( i );
  // On l'affiche en la vidant
  while ( ! pile.empty() ) {
    // Affiche le sommet de pile
    cout << " " << pile.top();
    // depile
    pile.pop();
  }
  cout << endl;
}
```

ex_stack2.cxx

```
#include <stack>
#include <string>
#include <iostream.h>
int main() {
  // Instanciation d'une pile de chaines
  stack<string> pile;
  string s( "bac" );
  // On remplit la pile
  for ( int i = 1; i < 4; i++ ) {
    // Empile la valeur de 's'
    pile.push( s );
    // Modifie 's'
    s = s + s;
  }
  // On l'affiche en la vidant
  while ( ! pile.empty() ) {
    // Affiche le sommet de pile
    string t = pile.top();
    cout << " " << t;
    // Depile
    pile.pop();
  }
}
```

Exercices sur les `stack`(1)

Exercice 2.

Donnez l'affichage produit par ces programmes.

Solution.

Le premier donne à l'exécution :

512 256 128 64 32 16 8 4 2 1

et l'autre :

bacbacbacbac bacbac bac

Exercices sur les `stack`(1)

Exercice 2.

Donnez l'affichage produit par ces programmes.

Solution.

Le premier donne à l'exécution :

512 256 128 64 32 16 8 4 2 1

et l'autre :

bacbacbacbac bacbac bac

Exercices sur les `stack`(2)

Exercice 3.

Rajoutez une fonction qui affiche les éléments d'une pile d'entiers passée en paramètre dans l'ordre inverse. Le programme appelle cette fonction et doit alors afficher :

```
1   2   4   8   16   32   64   128   256   512
```

Solution.

```
void reverseDisplay(stack<int> p) {
  stack<int> tmp;
  while(!p.empty()) {
    tmp.push(p.top()); p.pop();
  }
  while(!tmp.empty()) {
    cout << tmp.top() << " "; tmp.pop();
  }
}
```

Exercices sur les `stack`(2)

Exercice 3.

Rajoutez une fonction qui affiche les éléments d'une pile d'entiers passée en paramètre dans l'ordre inverse. Le programme appelle cette fonction et doit alors afficher :

```
1   2   4   8   16   32   64   128   256   512
```

Solution.

```
void reverseDisplay(stack<int> p) {
  stack<int> tmp;
  while(!p.empty()) {
    tmp.push(p.top()); p.pop();
  }
  while(!tmp.empty()) {
    cout << tmp.top() << " "; tmp.pop();
  }
}
```

Exercices sur les `stack`(4)

Exercice 4.

Modifiez légèrement cette fonction pour qu'elle puisse accepter une pile de n'importe quel type.

Solution.

```
template <class T>
void reverseDisplay(stack<T> p) {
  ...
}
```

Exercices sur les `stack`(4)

Exercice 4.

Modifiez légèrement cette fonction pour qu'elle puisse accepter une pile de n'importe quel type.

Solution.

```
template <class T>
void reverseDisplay(stack<T> p) {
  ...
}
```

Plan de l'exposé

La classe générique `vector`

- Modélise les tableaux d'objets à une dimension.
- Gére naturellement des tableaux de taille variable de n'importe quel type qui dispose :
  - d'un constructeur par défaut,
  - d'un constructeur par copie,
  - et d'un opérateur d'affectation.

Instanciation

Un vecteur d'entiers s'instancie de la façon suivante :

type d'instanciation	statique	dynamique
tableau de 0 éléments	`vector<int> v;`	`vector<int>* ptr_v = new vector<int>;`
tableau de *n* éléments	`vector<int> v( n );`	`vector<int>* ptr_v = new vector<int>( n );`

Opérations de base

Opérations sur les vecteurs.

- Un `vector` donne le nombre de ses éléments par la méthode `size()`.
- Un `vector` fournit un accès direct en lecture et écriture pour un indice d'élément entre `0` et `size()-1`.
- On peut aussi accéder au premier et dernier élément d'un `vector` avec les méthodes `front()` et `back()`.

front() et back()

```
vector<int> v( 10 );
for ( int i = 0; i < 10; i++ )
  v[ i ] = i * i;
cout << v[ 0 ] << endl;              // 0 = 0 * 0
cout << v.front() << endl;           // idem
cout << v[ v.size() - 1 ] << endl; // 81 = 9 * 9
cout << v.back() << endl;            // idem
v.back() = 13;
cout << v[ v.size() - 1 ] << endl; // 13
```

push_back() et pop_back()

```
v.push_back( 10 * 10 );
cout << v.back() << endl;        // 100 = 10 * 10
v.push_back( 11 * 11 );
cout << v.back() << endl;        // 121 = 11 * 11
// v contient desormais 12 elements
```

```
v.pop_back();
v.pop_back();
v.pop_back();
// v contient desormais 9 elements
cout << v.size() << endl;        // 9 elements
```

Exercice sur les vecteurs.

Exercice 5.

Ecrivez une fonction `fibo` prenant un `vector<int>` en argument et lui rajoutant comme nouvel élément la somme de ses deux derniers éléments. Le vecteur contient au moins deux éléments. Écrivez une fonction `main` qui affiche les 12 premiers termes de la suite de Fibonacci.

Solution exercice 5.

```
void fibo( vector<int> & v )
{
  v. push_back( v.back() + v[ v.size() - 2 ] );
}

int main() {
  vector<int> v( 2 );
  v[ 0 ] = 0;
  v[ 1 ] = 1;
  for ( int i = 0; i < 10; i++ )
    fibo( v );
  for ( int i = 0; i < v.size(); i++ )
    cout << " " << v[ i ];
  cout << endl;
}
```

Exercice sur les vecteurs.

Solution exercice 5.

Doit afficher la série de Fibonacci :

```
0 1 1 2 3 5 8 13 21 34 55 89
```

Itérateurs sur un `vector`

Iterator

- Un itérateur d'un `vector<`**T**`>` dans le sens naturel est de type `vector<`**T**`>::iterator`.
- On se sert des méthodes `begin()` et `end()` d'une instance d'un vecteur pour obtenir respectivement l'itérateur sur le premier élément et l'itérateur suivant le dernier élément (!).
- Pour avancer, on utilise l'opérateur `++`.
- Pour connaître l'élément pointé par l'itérateur, on utilise l'opérateur d'indirection `*`.

Exemple

```
#include <iostream.h>
#include <vector>
int main() {
  vector<int> v( 10 );
  for ( int i = 0; i < 10; i++ )
    v[ i ] = 100 - i * i;
  // 'p' pointe sur le premier element
  vector<int>::iterator p = v.begin();
  // Tant que 'p' ne pointe pas sur l'element apres le dernier.
  while ( p != v.end() )
    {
      // On affiche l'element pointe par 'p'.
      cout << " " << *p;
      // On avance au suivant.
      p++;
    }
```

Exemple(suite)

```
  cout << endl;
  // Avec les vecteurs, on peut deplacer ou
  //on veut les iterateurs
  p = v.begin() + 5; // 6e element
  while ( p != v.end() - 1 ) // 9e element
    {
      // On affiche l'element pointe par 'p'.
      cout << " " << *p;
      // On avance de deux coups.
      p += 2;
    }
  cout << endl;
}
```

Exercices sur les itérateurs.

Exercice 6.

Quel affichage produit ce programme?

Solution.

100 99 96 91 84 75 64 51 36 19 75 51

Exercices sur les itérateurs.

Exercice 6.

Quel affichage produit ce programme?

Solution.

100 99 96 91 84 75 64 51 36 19 75 51

Exercice sur les itérateurs.

Exercice 7.

Modifiez ce programme pour qu'il affiche le 2e, le 5e, et le 8e élément.

Solution.

```
p = v.begin() + 1;
...
p += 3;
```

Exercice sur les itérateurs.

Exercice 7.

Modifiez ce programme pour qu'il affiche le 2e, le 5e, et le 8e élément.

Solution.

```
p = v.begin() + 1;
...
p += 3;
```

Algorithmes simples: fill

fill

```
vector <int> v( 10 );
// Remplit 'v' avec des 7
fill( v.begin(), v.end(), 7 );
affiche( v );
```

Algorithmes simples: sort

sort

```
// Trie 'v' entre le 3e et le 8e element inclus
vector <int>::iterator p = v.begin();
sort( p + 3, p + 9 );
affiche( v );
// Trie tous les elements de 'v'
sort( v.begin(), v.end() );
affiche( v );
```

Algorithmes simples: reverse

reverse

```
// Inverse l'ordre du 1e au 5e element
reverse( v.begin(), v.begin() + 5 );
affiche( v );
```

Algorithmes simples:random_shuffle

random_shuffle

```
// Melange le vecteur a partir du 5e element
random_shuffle( v.begin() + 5, v.end() );
affiche( v );
```

Algorithmes simples:find

find

```
// Renvoie l'iterateur sur l'element 15
vector<int>::iterator p =
  find( v.begin(), v.end(), 15 );
cout << "Valeur recherchee: " << *p << endl;
// S'il ne le trouve pas, renvoie "v.end()"
p = find( v.begin(), v.end(), 20 );
if ( p == v.end() )
  cout << "non trouve" << endl;
```

Algorithmes simples: min_element et max_element

min_element

```
// Renvoie l'iterateur sur l'element minimum
p = min_element( v.begin(), v.end() );
cout << "Valeur min: " << *p << endl;
```

max_element

```
// Renvoie l'iterateur sur l'element maximum
// parmi les 6 derniers elements
p = max_element( v.end() - 6, v.end() );
cout << "Valeur max: " << *p << endl;
```

Algorithmes simples: min_element et max_element

min_element

```
// Renvoie l'iterateur sur l'element minimum
p = min_element( v.begin(), v.end() );
cout << "Valeur min: " << *p << endl;
```

max_element

```
// Renvoie l'iterateur sur l'element maximum
// parmi les 6 derniers elements
p = max_element( v.end() - 6, v.end() );
cout << "Valeur max: " << *p << endl;
```

Plan de l'exposé

Le container `map`

- Le container `map` est un tableau associatif.
- Pour instancier une `map` :

```
map<type_clé, type_valeur_associée>
```

- L'implémentation est comparable à celle des ensembles, les clés doivent donc être comparables.

Exemple avec `map`

Exemple

```
int main(){
  map<string, int> m;
  string s;
  cin>>s;
  while(s!="stop"){
    m[s]++;
    cin >> s;
  }
  map<string, int>::iterator i;
  for(i=m.begin();i!= m.end();i++)
    cout << i->first <<" " << i->second << endl;
  cout << endl << m.size() << endl;
}
```

Exemple

Dans le cas de la saisie suivante :

```
toto
bonjour
attention attention attention
bonjour bonjour
toto
mais attention
stop
```

Affiche :

```
attention 4
bonjour 3
mais 1
toto 2

4
```

Exemple

Dans le cas de la saisie suivante :

```
toto
bonjour
attention attention attention
bonjour bonjour
toto
mais attention
stop
```

Affiche :

```
attention 4
bonjour 3
mais 1
toto 2

4
```

Exercice 8.

En utilisant un vecteur de `Piece`, remplacer la définition d'un joueur pour permettre la gestion des différents types de piece. Modifier le constructeur et le destructeur de la classe `Joueur`.

Solution 1

```
vector<Piece> m_pieces;
```

Solution 2

```
vector<Piece*> m_pieces;
```

Plan de l'exposé

Inférence de type (1)

On peut écrire :

```
auto nombre = 42; // variable de type entier
```

Mais risque de devenir rapidement illisible !

- `auto` n'est pas comme un type qui change en cours d'exécution (cette variable sera et restera un entier).
- vous devez obligatoirement initialiser une variable déclarée avec `auto` à la déclaration.

Utilisation de `decltype` :

```
auto variable = 42;
decltype(variable) autreVariable;
```

Inférence de type (2)

A la place de :

```
for(vector<int>::iterator i(nombres.begin()) ;
    i != nombres.end() ; ++i) {
        cout << *i << endl;
}
```

On peut utiliser `auto` :

```
for(auto i(nombres.begin()) ; i != nombres.end() ; ++i) {
    cout << *i << endl;
}
```

Mais on peut préférer `typedef` :

```
typedef vector<int>::iterator iter_entier;
for(iter_entier i(nombres.begin()) ; i != nombres.end() ; ++i) {
    cout << *i << endl;
}
```

Inférence de type (3)

Une fonction surchargée pour les entiers et les réels :

```
int ajouterDixPourcents(int nombre) {
    if(nombre + 10 <= 100)
        return nombre + 10;
    else
        return 100;
}

float ajouterDixPourcents(float nombre) {
    if(nombre + 0.1 <= 1)
        return nombre + 0.1;
    else
        return 1;
}
```

Inférence de type (4)

Une fonction générique aura le prototype suivant :

```
template <typename T> auto ajouter(T nombre) ->
    decltype(ajouterDixPourcents(nombre)) {
        return ajouterDixPourcents(nombre);
}
```

Example d'utilisation :

```
int main() {
    int pcEntier(42);
    float pcFlottant = pcEntier / 100.f;
    cout << pcFlottant << " = " << pcEntier << "%" << endl;

    pcEntier = ajouter(pcEntier);
    pcFlottant = ajouter(pcFlottant);
    cout << pcFlottant << " = " << pcEntier << "%" << endl;
}
```

Boucle basée sur un intervalle (1)

Version `for_each` :

```
#include <algorithm>
#include <iostream>
#include <vector>

void afficherElement(int element) {
    cout << element << endl;
}

int main() {
    vector<int> nombres = { 1, 2, 3, 4, 5 };
    for_each(nombres.begin(), nombres.end(), afficherElement);
}
```

Boucle basée sur un intervalle (2)

Version `for` :

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    vector<int> nombres = { 1, 2, 3, 4, 5 };
    for(int &element : nombres) {
        cout << element << endl;
    }
}
```

Boucle basée sur un intervalle (3)

Idem pour un tableau associatif :

```
map<int, string> nombres = {{1,"un"}, {2,"deux"}, {3,"trois"}};
for(auto i : nombres) {
    cout << i.first << " => " << i.second << endl;
}
```

Mettre à jour tous les itérateurs des classes `Piece`, `Joueur` et `Echiquier`.

Enumération (1)

Enumération standard

```
enum TypePiece { PION, ROI, REINE, TOUR, CAVALIER, FOU };
TypePiece montype = TOUR;
cout << montype << endl;
cout << static_cast<int>(montype) << endl; // idem
```

Enumération fortement typée : `class` après `enum`

```
enum class TypePiece { Pion, Roi, Reine, Tour, Cavalier, Fou };
TypePiece montype = TypePiece::Tour;
cout << montype << endl; // ne compile pas !
cout << static_cast<int>(montype) << endl; // ok
```

Assurer une certaine sécurité

```
int nombre = 42 + TypePiece::Tour; // ne compile pas !
```

Enumération (2)

Définition d'une couleur blanche ou noire

```
enum class Color {
    colorNone,  // Represente l'absence de couleur
    colorWhite, // Represente la couleur blanche
    colorBlack, // Represente la couleur noire
    colorMax    // Valeur maximale a ne jamais atteindre
};
```

Initialisation

Comme pour un tableau statique

```
int nombres[] = { 1, 2, 3, 4, 5 };
```

Utiliser une liste d'initialisateurs

```
#include <iostream>
#include <map>
#include <ostream>
#include <string>

int main() {
    map<string, int> nombres = { { "roi", 1 },
        { "fou", 2 }, { "pion", 8 } };
    cout << nombres["fou"] << endl;
}
```

Initialisation d'un pointeur

Block

```
int *pointeur = NULL; // comme en C
int *pointeur(0); // en C++
int *pointeur(nullptr); // en C++ 11
```

Impossible d'utiliser `nullptr` comme un entier

```
int nul = NULL; // compile
int nul = nullptr; // ne compile pas
```

Mettre à jour l'initialisation de la classe `Echiquier`.

Tableau de taille fixe (1)

Il y a un conteneur de la STL pour ce genre de tableau.

std::array<TYPE, TAILLE> NOM;

```
#include <array>
std::array<int, 5> tableauFixe; // idem int tableauFixe[5];
std::array<int, 5> tableauFixeInit = { 1, 2, 3, 4, 5 };
```

La taille ne peut pas être une variable, mais peut être une constante

```
int const taille = 5;
std::array<int, taille> tableauFixe =  1, 2, 3, 4, 5 ;
// compile ou pas suivant const
```

Tableau de taille fixe (2)

Parcourir le tableau avec une boucle basée sur un intervalle

```
for(int nombre : tableauFixe) {
    cout << nombre << endl;
}
```

Accéder aux éléments d'un tableau

```
tableauFixe[0] = 10;
tableauFixe.at(1) = 40;
```

La méthode `at()` fait une vérification des limites

Une exception `std::out_of_range` sera levée !

Les méthodes `size()`, `front()`, `back()`, `empty()`, `fill()`, `begin()` et `end()` fonctionnent normalement.

Tableau de taille fixe (3)

Modifier la définition d'un Echiquier.

Solution

```
typedef std::array<std::array<Piece *, 8>, 8> Board8;
typedef std::array<Piece *, 64> Board64;

class Coordinate{
    public:
        Coordinate():m_x(m_max),m_y(m_max){}
        Coordinate(int x, int y):m_x(x),m_y(y){}
        int x() const {return m_x;}
        int y() const {return m_y;}
        bool isValid() const { return m_x<=m_max && m_y <=m_max; }
    private:
        static const int m_max = 8; int m_x; int m_y;
};
```

Tableau de taille fixe (4)

Modifier la définition d'une Pièce (coordonnées, couleur, a été déplacée, a été mangée ...)

Solution (suite)

```
inline bool operator == (Coordinate const & first,
                         Coordinate const & second) {
    if(!first.isValid()) return !second.isValid();
    return first.x() == second.x() &&
           first.y() == second.y();
}

inline int toIndex(int x, int y);
inline int toIndex(Coordinate c);
inline Coordinate toCoordinate(int index);
```

Tuple

Un tuple est une collection de dimension fixe d'objets de types différents.

Block

```
#include <tuple>
std::tuple<int, int, bool> m_tuple;
std::tuple<int, int, bool> roi_blanc(5, 1, true);
std::get<0>(roi_blanc) = 6;
```

Remplacer la définition d'une pièce.

Solution

```
typedef std::tuple<Coordinate, Color, bool, bool> tuple_piece;
```

Programmation par contrat (1)

En C++11, il existe un mot clé `static_assert` permettant d'implémenter un précondition, une postcondition ou un invariant. Ces propriétés seront vérifées à la compilation !

Exemples

```
int i;
static_assert(sizeof(void *) >= sizeof i);
foo((void *) i);

static_assert(CHAR_MIN < 0); // char type is signed

static_assert(-5 / 2 == -2);
```

Il est également possible d'ajouter le mot clé `final` permettant d'exprimer qu'une classe n'est pas destinée à être (davantage) dérivée.

Programmation par contrat (2)

- `std::is_base_of` : vérification du type par rapport à son héritage
- `std::is_same` : le type de gauche doit être identique au type de droite
- `std::enable_if` : demande au compilateur de fournir une implémentation si la condition est vérifiée

Exemples

```
cout << std::is_base_of<Piece, Roi>::value << endl;     // true
cout << std::is_base_of<Roi, Piece>::value << endl;     // false
cout << std::is_same<int, unsigned int>::value << endl; // false
cout << std::is_same<int, signed int>::value << endl;   // true
```

Classes ayant sémantique de valeur/entité (1)

Sémantique de valeur : `Color`, `Coordinate`

- généralement constantes : si vous modifiez une des valeurs qui les composent, vous obtenez un objet totalement différent
- copiables
- affectables
- comparables, au minimum par égalité
- peu enclines à intervenir dans une relation d'héritage, ni en tant que classe de base, ni en tant que classe dérivée

Classes ayant sémantique de valeur/entité (2)

Sémantique d'entité : `Piece`, `Joueur`, `Echiquier`

- généralement modifiables : au moins pour tout ce qui n'intervient pas dans la définition de ce qui permet de l'identifier de manière unique
- non copiables
- non affectables
- non comparables (dans son intégralité)
- candidats à intervenir dans une relation d'héritage, que ce soit comme classe de base ou comme classe dérivée

Interdire la copie et l'affectation

Mot clé `delete`

Déclarer le constructeur par copie et l'opérateur d'affectation comme étant "deleted" pour indiquer au compilateur qu'il ne doit pas fournir son implémentation par défaut pour ces deux fonctions et au contraire en interdire l'utilisation.

```
class NonCopyable{
public:
    NonCopyable(NonCopyable const &) = delete;
    NonCopyable & operator = (NonCopyable const & ) = delete;
protected:
    // protege car utilisable uniquement par les classes derivees
    NonCopyable(){}
    ~NonCopyable(){}
};
```

Fonction Lambda (1)

C'est une fonction ... qui n'a pas de nom ! Elle peut-être passée en paramètre ou stockée dans une variable. Comme une fonction ordinaire : il y a l'accolade ouvrante, les instructions et l'accolade fermante.

Fonction anonyme : [captures] (paramètres) {corps}

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    vector<int> nombres = { 1, 2, 3, 4, 5 };
    for_each(nombres.begin(), nombres.end(), [](int element) {
        cout << element << endl; }
      );
}
```

Fonction Lambda (2)

Assigner une fonction anonyme à une variable :

```
std::function<int (int, int)> addition = [](int x, int y) -> int {
    return x + y;
};
```

En pratique, on utilisera `auto` :

```
auto addition = [](int x, int y) -> int {
    return x + y;
};
```

Le type de retour de la fonction anonyme est facultatif

S'il n'est pas précisé, le type sera calculé par `decltype()`

Fonction Lambda (3)

Dispatch table

```
map<const char, function<double(double,double)>> tab;

tab.insert(make_pair('+',[](double a,double b){return a + b;}));
tab.insert(make_pair('-',[](double a,double b){return a - b;}));
tab.insert(make_pair('*',[](double a,double b){return a * b;}));
tab.insert(make_pair('/',[](double a,double b){return a / b;}));

cout << "3.5+4.5= " << tab['+'](3.5,4.5) << endl; // 8
cout << "3.5*4.5= " << tab['*'](3.5,4.5) << endl; // 15.75

tab.insert(make_pair('^',
                     [](double a,double b){return pow(a,b);}));

cout << "3.5^4.5= " << tab['^'](3.5,4.5) << endl; // 280.741
```

Fonction Lambda (4)

Map-Filter-Reduce

- Map `std::transform` : applique une fonction à chaque élément d'une liste
- Filter `std::remove_if` : retire des éléments d'une liste
- Reduce `std::accumulate` : réduit une liste à une valeur unique en appliquant successivement une opération binaire

Exemple

```
vector<int> vec{1,2,3,4,5,6,7,8,9};
vector<string>str{"Programming","in","a","functional","style."};
```

Fonction Lambda (5)

Exemple Map

```
transform(vec.begin(),vec.end(),vec.begin(),
          [](int i){ return i*i; });
// [1,4,9,16,25,36,49,64,81]
transform(str.begin(),str.end(),back_inserter(vec2),
          [](string s){ return s.length(); });
// [11,2,1,10,6]
```

Exemple Filter

```
auto it= remove_if(vec.begin(),vec.end(),
          [](int i){ return !((i < 3) or (i > 8)) });
// [1,2,9]
auto it2= remove_if(str.begin(),str.end(),
          [](string s){ return !(isupper(s[0])); });
// [“Programming”]
```

Fonction Lambda (6)

Exemple Reduce

```
accumulate(vec.begin(),vec.end(),1,
           [](int a, int b){ return a*b; });
// 362800
accumulate(str.begin(),str.end(),string(""),
           [](string a,string b){ return a+":"+b; });
// ":Programming:in:a:functional:style."
```

Fermeture (1)

Une fermeture est une fonction qui capture des variables à l'extérieur de celle-ci. Il y a deux facons de capturer une variable :

- par référence : les modifications apportées à la variable s'appliqueront à la variable capturée

par référence, il suffit de précéder son nom par & :

```
[&somme](vector<int> vecteur {
    for(int &x : vecteur) {
        somme += x;
    }
}
```

Fermeture (2)

Exemple complet

```
int somme(0);
std::function<void (std::vector<int>)> sommeVecteur =
    [&somme](std::vector<int> vecteur) {
        for(int &x : vecteur) {
            somme += x;
        }
    };
std::vector<int> nombres = { 1, 2, 3, 4, 5 };
sommeVecteur(nombres);
std::cout << somme << std::endl;
```

Fermeture (3)

- par valeur : les modifications n'affecteront pas la variable capturée

par valeur, il suffit d'écrire son nom :

```
[x](int &a) {
    a = x;
}
```

Fermeture (4)

Syntaxe

- [] : pas de variable externe ;
- [x, &y] : x capturée par valeur, y capturée par référence ;
- [&] : toutes les variables externes sont capturées par référence ;
- [=] : toutes les variables externes sont capturées par valeur ;
- [&, x] : x capturée par valeur, toutes les autres variables, par référence ;
- [=, &x] : x capturée par référence, toutes les autres variables, par valeur.

Fermeture (5)

Exemple de fonction Lambda avec Fermeture

```
auto ajouteur = [](int i) {
    return ([=](int j) -> int {
        return i + j;
    });
};
auto ajoute10 = ajouteur(10);
ajoute10(1); // retourne 11
```

Lambda ou Fermeture pour le jeu d'echec ???

Pointeur Intelligent (1)

Avec les pointeurs intelligents, on possède un mécanisme pour résoudre le problème de gestion de ressources. RAII (Ressource Acquisition Is Initialisation) : ce principe nous dit que pour chaque acquisition de ressource, on crée aussi un objet qui va garantir sa bonne gestion.

Le C++ ne dispose pas de mécanisme de "garbage collector"

- La libération des ressources est déterministe, à la destruction du pointeur (`unique_ptr`) ou du dernier pointeur utilisant un objet (`shared_ptr`).
- La gestion des cycles pose un problème (`weak_ptr`).
- `unique_ptr` ne demande pas de compteur de référence, c'est un pointeur très léger.

Pointeur Intelligent (2)

Si une fonction retourne un pointeur, il n'y a pas de moyen de savoir si la fonction appelante doit libérer la mémoire de l'objet ou de variable pointée.

Que faire avec le résultat ?

```
some_type* ambiguous_function();
```

La fonction appelante récupère la possession du résultat

```
unique_ptr<some_type> obvious_function();
```

Pointeur Intelligent (3)

std::unique_ptr<some_type>

`unique_ptr` représente un pointeur qui, comme son nom l'indique, est le seul pointeur à pointer sur un objet. Quand le pointeur est détruit, l'objet est détruit. Que se passe-t-il quand on copie ce pointeur ? Il ne peut rien se passer, car la copie en est interdite. Par contre, il est possible de déplacer ce pointeur, pour transférer la responsabilité de l'objet pointé à quelqu'un d'autre. `unique_ptr` ne demande pas de compteur de référence, c'est un pointeur très léger.

Il est conseillé de ne plus utiliser `auto_ptr` qui disparaitera dans C++ 17

Pointeur Intelligent (3)

std::unique_ptr<some_type>

```
std::unique_ptr<int> p1(new int(42));
auto p1 = std::make_unique<int>(42); // idem
std::unique_ptr<int> p2 = p1; // ne compile pas !
std::unique_ptr<int> p3 = std::move(p1);
// transfert la possession
// p3 possede maintenant la mémoire et p1 est rendu invalide

p3.reset(); // delete
p1.reset(); // ne fait rien
```

Il est conseillé de ne plus utiliser `auto_ptr` qui disparaitera dans C++ 17

Pointeur Intelligent (4)

std::shared_ptr<some_type>

Quand un pointeur est détruit, il vérifie s'il n'était pas le dernier à pointer sur son objet. Si c'est le cas, il détruit l'objet, puisqu'il est sur le point de devenir inaccessible. Un compteur de référence est associé à l'objet, à chaque fois qu'un nouveau pointeur pointe sur l'objet, le compteur de référence est incrémenté. A chaque fois que le pointeur arrête de pointer sur cet objet (parce qu'il est détruit, ou s'apprête à pointer sur un autre), le compteur est décrémenté. Si le compteur atteint 0, il est temps de détruire l'objet.

La fonction `use_count()` retourne la valeur du compteur de référence d'un `shared_ptr`.

Pointeur Intelligent (4)

std::shared_ptr<some_type>

```
std::shared_ptr<int> p1(new int(42));
auto p1 = std::make_shared<int>(42); // idem
std::shared_ptr<int> p2 = p1;
// les deux pointeurs sont en possession de la memoire

p1.reset(); // pas de liberation memoire du fait de p2
p2.reset(); // delete, car plus personne ne possede la memoire
```

La fonction `use_count()` retourne la valeur du compteur de référence d'un `shared_ptr`.

Pointeur Intelligent (5)

Conversion entre `shared_ptr`

```
class A {/*...*/};
class B : public A {/*...*/};

B* b1(new B);
A* a(b1);
B* b2 = dynamic_cast<B*>(a);

shared_ptr<B> sb1(new B);
shared_ptr<A> sa(sb1);
shared_ptr<B> sb2 = dynamic_pointer_cast<B>(sa);
```

`dynamic_pointer_cast` retourne un pointeur ne pointant sur rien si la conversion n'a pu avoir lieu. Il existe de même une fonction `static_pointer_cast` mimant l'effet d'un `static_cast` pour un pointeur nu.

Pointeur Intelligent (6)

Risque de fuite mémoire : Cycle

```
class A                          class B
{                                {
    // ...                           // ...
    shared_ptr<B> myB;               shared_ptr<A> myA;
};                               };

shared_ptr<A> a = new A;
shared_ptr<B> b = new B;
cout << a.use_count() << ", " << b.use_count() << endl; // 1, 1
a->myB = b;
cout << a.use_count() << ", " << b.use_count() << endl; // 1, 2
b->myA = a;
cout << a.use_count() << ", " << b.use_count() << endl; // 2, 2
a.reset(); // b->myA fait survivre l'objet de type A !
b.reset(); // a->myB fait survivre l'objet de type B !
```

Pointeur Intelligent (7)

std::weak_ptr<some_type>

`weak_ptr` a été conçu spécifiquement pour travailler en collaboration avec `shared_ptr`, pour casser les cycles. L'idée de base est de dire qu'en fait, parmi les pointeurs sur un objet, certains (les `shared_ptr`) se partagent la responsabilité de faire vivre ou mourir ce dernier, le possèdent, et d'autres (les `weak_ptr`) y ont un simple accès, mais sans aucune responsabilité associée. Un `weak_ptr` n'impacte donc pas le comptage de référence de l'objet sur lequel il pointe.

Pointeur Intelligent (7)

std::weak_ptr<some_type>

Supposons par exemple un tiroir rempli de chaussettes, chaque chaussette ayant besoin de savoir où elle est rangée.

```
class Tiroir
{
    vector<shared_ptr<Chaussette> > mesChaussettes;
};
class Chaussette
{
    weak_ptr<Tiroir> monTiroir;
};
```

Si on jette un tiroir à la poubelle, on a automatiquement jeté les chaussettes qu'il contient (sauf une chaussette qui serait tenue par quelqu'un d'autre).

Pointeur Intelligent (7)

std::weak_ptr<some_type>

Supposons par exemple un tiroir rempli de chaussettes, chaque chaussette ayant besoin de savoir où elle est rangée.

```
class Tiroir
{
    vector<weak_ptr<Chaussette> > mesChaussettes;
};
class Chaussette
{
    shared_ptr<Tiroir> monTiroir;
};
```

Tant qu'on utilise une chaussette, il faut pouvoir la ranger, et donc le tiroir doit survivre tant qu'il lui reste une chaussette.

Pointeur Intelligent (8)

Obtenir un `shared_ptr` à partir de `this`

```
class Good : std::enable_shared_from_this<Good>
{
    std::shared_ptr<Good> getptr() {
        return shared_from_this();
    }
};

class Bad
{
    std::shared_ptr<Bad> getptr() {
        return std::shared_ptr<Bad>(this);
    }
    ~Bad() { std::cout << "Bad::~Bad() called\n"; }
};
```

Pointeur Intelligent (9)

double-delete pour la classe Bad

```
int main()
{
    // Good: les deux shared_ptr partagent le meme objet
    std::shared_ptr<Good> gp1(new Good);
    std::shared_ptr<Good> gp2 = gp1->getptr();
    cout << "gp2.use_count()=" << gp2.use_count() << endl; // 2

    // Bad: chaque shared_ptr pense posseder l'objet
    std::shared_ptr<Bad> bp1(new Bad);
    std::shared_ptr<Bad> bp2 = bp1->getptr();
    cout << "bp2.use_count()=" << bp2.use_count() << endl; // 1

    // Bad::~Bad() called
    // Bad::~Bad() called
}
```

Exercice 9.

Utiliser des pointeurs intelligents et retirer toutes les instructions `delete` ! On utililsera des `shared_ptr` pour compter les réferences sur les Pieces.

Gestion du temps

Chrono

```
#include <chrono>
#include <iostream>

int main() {
    auto t1 = std::chrono::system_clock::now();
    usleep(100000);
    auto t2 = std::chrono::system_clock::now();
    cout << (t2 - t1).count() << " microsecondes." << endl;
    return 0;
}
```

Conversion dans une autre unité de temps

```
std::chrono::nanoseconds nbrNanoSecondes = (t2 - t1);
cout << nbrNanoSecondes.count() << " nanosecondes." << endl;
```

Threads (1)

Lancer un groupe de threads

```
#include <thread>
static const int num_threads = 10;

void call_from_thread(int id) {
    cout << id << std::this_thread::get_id() << endl;
}
int main() {
    std::thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i);
    }
    cout << t[0].get_id() << endl;
    for (int i = 0; i < num_threads; ++i) {
        t[i].join();
    }
}
```

Threads (2)

Version pointeur de fonction

```
void thread_function() {
    cout << "Thread function Executing" << endl;
}

int main() {
    std::thread t(thread_function);
    // ...
    cout << "Waiting For Thread to complete" << endl;
    t.join();
    cout << "Exit of Main function" << endl;
}
```

Threads (3)

Version foncteur

```
class DisplayThread {
public:
    void operator()() {
        cout << "Display Thread Executing" << endl;
    }
};

int main() {
    std::thread t( (DisplayThread()) );
    // ...
    cout << "Waiting For Thread to complete" << endl;
    t.join();
    cout << "Exiting from Main Thread" << endl;
}
```

Threads (4)

Version fonction lambda

```
int main() {
    std::thread t([]{
        cout << "Display Thread Executing" << endl;
        });
    // ...
    cout << "Waiting For Thread to complete" << endl;
    t.join();
    cout << "Exiting from Main Thread" << endl;
}
```

Threads (5)

Version fonction lambda avec fermeture

```
int main() {
    int x=1,y=2;
    std::thread t([x,&y]{
        cout << "Display Thread Executing" << x << y << endl;
        });
    // ...
    cout << "Waiting For Thread to complete" << endl;
    t.join();
    cout << "Exiting from Main Thread" << endl;
}
```

Threads (6)

Passage par référence

```
void callback(int const & x) {
    int & y = const_cast<int &>(x);
    y++;
    cout << "Inside Thread x = " << x << endl;
}

int main() {
    int x = 42;
    cout << "Before Thread Start x = " << x << endl;
    std::thread t(callback,std::ref(x));
    t.join();
    cout << "After Thread Join x = " << x << endl;
}
```

Sans `std::ref(x)` le résultat est faux (42) car le paramètre `x` est placé dans la pile locale du thread.

Appel d'une fonction membre

```
class DummyClass {
public:
    DummyClass() {}
    DummyClass(const DummyClass & obj) {}
    void sampleMemberFunction(int x) {
        cout << "Inside sampleMemberFunction " << x << endl;
    }
};

int main() {
    DummyClass obj;
    int x = 10;
    std::thread t(&DummyClass::sampleMemberFunction, &obj, x);
    t.join();
}
```

Threads (8)

`std::future` et `std::promise`

- `std::future` est une classe générique qui stocke de manière interne une valeur qui sera affectée dans le futur, mais aussi un méchanisme d'accès avec la fonction membre `get()`. Si on tente d'accèder à cette valeur avant qu'elle ne soit disponible, la fonction `get()` sera bloquée jusqu'à ce qu'elle le devienne.
- `std::promise` est aussi une classe générique et chaque objet de cette classe est associé à un objet de la classe `std::future` qui fournira la valeur une fois renseignée par l'objet `std::promise`.

Threads (8)

`std::future` et `std::promise`

```
#include <future>

void initiazer(std::promise<int> * pObj) {
    pObj->set_value(42);
}

int main() {
    std::promise<int> promiseObj;
    std::future<int> futureObj = promiseObj.get_future();
    std::thread t(initiazer, &promiseObj);
    cout << futureObj.get() << endl;
    t.join();
}
```

[rappel déclaration d'une variable (partagée) `volatile`]

Threads (9)

`std::mutex` : méthodes `lock()` et `unlock()`

```
#include<mutex>

class Wallet {
    int m_money;
    std::mutex m;
public:
    Wallet() : m_money(0) {}
    int getMoney() { return m_money; }
    void addMoney(int money) {
        m.lock();
        for(int i = 0; i < money; ++i)
            m_money++;
        m.unlock();
    }
};
```

Threads (9)

std::mutex : méthodes lock() et unlock()

```
int main()
{
    Wallet w;
    std::vector<std::thread> threads;
    for(int i = 0; i < 6; ++i) {
        threads.push_back(std::thread(&Wallet::addMoney, &w, 7));
    }
    for(int i = 0; i < threads.size() ; i++) {
        threads.at(i).join();
    }
    cout << "Money in Wallet = " << w.getMoney() << endl; // 42
}
```

Threads (9)

`std::mutex` : méthodes `lock()` et `unlock()`

```
class Wallet {
    int m_money;
    std::mutex m;
public:
    Wallet() : m_money(0) {}
    int getMoney() { return m_money; }
    void addMoney(int money) {
        std::lock_guard<std::mutex> guard(m); // RAII pour m
        // lock() auto du mutex dans le constructeur de guard
        for(int i = 0; i < money; ++i)
            m_money++;
        // unlock() auto du mutex dans le destructeur
        // meme en cas d'exception !
    }
};
```

Semaphore et variable condition

```
#include <mutex>
#include <condition_variable>

class semaphore
{
private:
    std::mutex mutex_;
    std::condition_variable condition_;
    unsigned long count_;
public:
    semaphore() : count_() {}
    void notify() {
        std::lock_guard<std::mutex> lock(mutex_);
        ++count_;
        condition_.notify_one();
    }
    void wait() {
        std::lock_guard<std::mutex> lock(mutex_);
        while(!count_) condition_.wait(lock);
        --count_;
    }
};
```

Pour aller plus loin

Facilitez-vous la vie

- la documentation à portée de main : `doxygen`
- garder l'historique : `git`, `mercurial`
- traquer la bestiole : `gdb`, `valgrind`, `cppcheck`
- automatiser la compilation : `make`, `cmake`
- intégration continue : `gcov`, `jenkins`

C++17 et C++20

`https://linuxfr.org/news/les-coulisses-du-standard-cpp`