

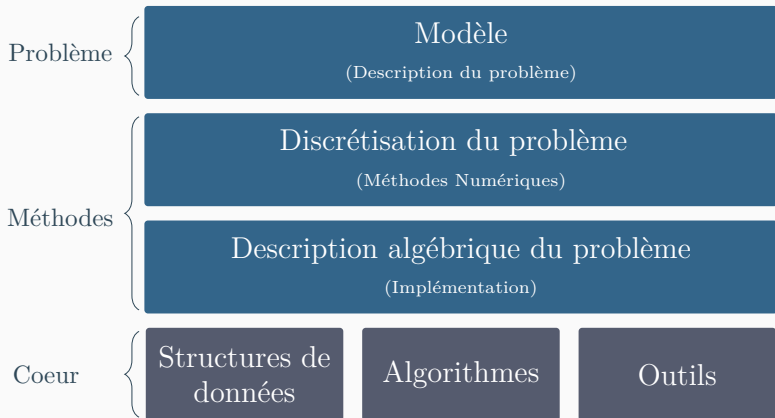
Modélisation et programmation

Cours 5

ENSIMAG 2A – MMIS

Ensimag

Rappels



La STL, c'est quoi?

Un ensemble de classes génériques et de fonctions qui possèdent une interface consistante. Elle se compose de

- Classes conteneurs
- Itérateurs
- Algorithmes génériques
- Des pointeurs intelligents C++11
- Des générateurs aléatoires C++11
- ...

Conteneurs

En utilisant les pointeurs, il est facile de définir deux structures pour gérer un ensemble très grand de données homogènes

- les tableaux
- les listes

La STL contient bien d'autres conteneurs que l'on classifie en deux catégories.

Tous les conteneurs sont des classes génériques

- `std::vector<Manchot>`
- `std::list<Videos>`
- `std::stack<Feuille>`
- `std::array<Haie,10>`
- `std::map<Coordonees,Villes>`

Conteneurs séquentiels

Ce sont les conteneurs dans lesquels les données sont stockés les uns après les autres

- `std::vector`
 - `std::array` C++11
 - `std::list`
 - `std::stack` ← FIFO
 - `std::queue`
- } Peut remplacer les tableaux C

Les éléments sont enregistrés de manière continue en mémoire

Les éléments dans un tableau peuvent être accédés soit

- L'opérateur d'accession `[]`
- La méthode `at()`

La classe vecteur n'a presque pas de surcout en accès par rapport à un tableau dynamique C.

Conteneur qui représente un tableau séquentiel dynamique

<code>(constructor)(size_t)</code>	défini la taille initiale
<code>operator[] (size_t)</code>	accède au énième élément
<code>at(size_t)</code>	accède au énième élément avec vérification des bornes
<code>resize(size_t)</code>	redimensionne le vecteur
<code>front()</code> , <code>back()</code>	accède le premier/dernier élément
<code>size()</code>	récupère la taille courante

Les éléments ne sont pas stockés de manière séquentielle en mémoire

Pas d'accès direct à un élément. Il est nécessaire de parcourir la liste

Insertion et délétion sont peu coûteuses.

Il est possible d'initialiser un conteneur en utilisant une liste

```
std::vector<int> lucky_numbers {12, 5, 42};  
std::list<char> the_word {'b', 'i', 'r', 'd'};
```

Des constructeurs similaires peuvent être définis pour des classes propres en utilisant le conteneur `initializer_list`

En C++11, la notation de constructeur `{}` sélectionnera de préférence le constructeur par liste d'initialisation

```
std::vector zero_vector {0}; // constr C++11  
std::vector zero_vector (0); // constr pre
```

Les éléments sont triés et les recherches sont très rapides

- **`std::set`**
collection triée d'éléments unique
- **`std::map`**
collection triée de paire (clé-valeur) avec des clés uniques

- `set<key_type>`
- `multi_set<key_type>`
- `map<key_type,value_type>`
- `multi_map<key_type,value_type>`
- `hash_set<key_type>`

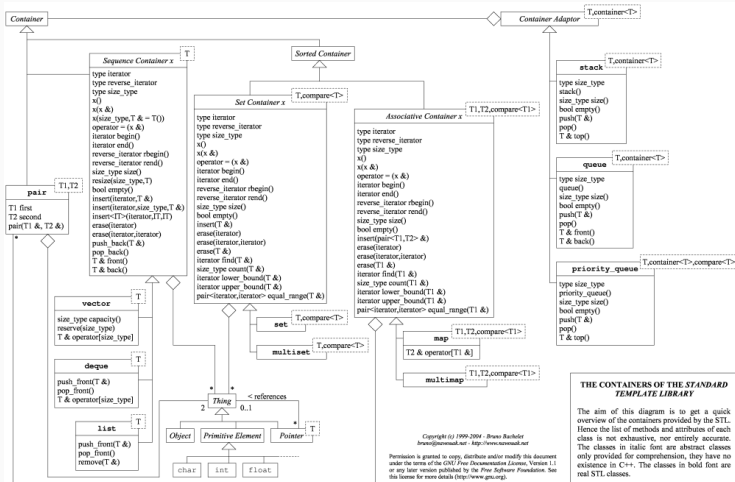
Complexité

	Accès	Recherche	Insertion
array	$O(1)$	$O(n)$	$O(n)$
stack	$O(n)$	$O(n)$	$O(1)$
list	$O(n)$	$O(n)$	$O(1)$
map	-	$O(1)$	$O(1)$
set	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

99% du temps vous choisirez
vector ou **array**

Dans les autres situations, vous choisirez le conteneur pour en fonction de l'opération la plus fréquente

- Insertion aléatoire d'éléments dans le conteneur
- Besoin en terme d'itérateurs.
- Ordre fixe
- Consistance mémoire.



Itérateurs

Pour de nombreux conteneurs de la STL, un accès direct à un élément n'est pas possible. Le structure doit être parcouru pour atteindre un élément.

Un itérateur pointe sur un élément d'un conteneur

```
*iterator
```

et il peut se déplacer sur l'élément suivant d'un conteneur

```
++iterator
```

Exemple d'itérateur

```
#include<set>
#include<cctype>

int main()
{
    std::set<char> lower_set {'a', 'q', 'k', 'p'};
    std::set<char> upper_set;

    for (auto it = lower_set.begin();
         it != lower_set.end(); ++it)
        upper_set.insert(std::toupper(*it));
}
```

La plupart des conteneurs de la STL possède un itérateur,
et l'interface des itérateurs est uniforme

```
container.begin(); // begin iterator  
container.end();   // end iterator
```

```
++iterator; // advance  
*iterator;  // dereference
```


Une méthode plus élégante de parcourir un conteneur

```
std::vector<Employee> employees;
```

```
//...
```

```
for (auto & worker : employees) {  
    worker.work();  
}
```

Déclarations des itérateurs

- Les itérateurs de la STL sont définis à l'intérieur des classes des conteneurs

```
1  template <typename T>
2  class list
3  {
4      public :
5          class iterator
6          {
7              ...
8          };
9          ...
10 };
```

- Il est donc nécessaire de faire référence au type du conteneur pointé lorsqu'il est déclaré:

```
1  list<double>::iterator it;
```

Classification

- Un itérateur peut avoir différents comportements:
 - **input_iterator**: lecture seule (opérations `*`, `==`, `!=`, `++`)
 - **output_iterator**: écriture seule (opérations `*`, `++`)
 - **forward_iterator**: lecture et écriture avec un déplacement vers l'avant.
 - **bidirectional_iterator**: lecture et écriture avec un déplacement vers l'avant ou vers l'arrière.
 - **random_iterator**
- Les itérateurs ainsi présentés sont classés du moins permissif au plus permissif.

Itérateur à accès aléatoire

Il permet toutes les opérations des pointeurs arithmétiques C.

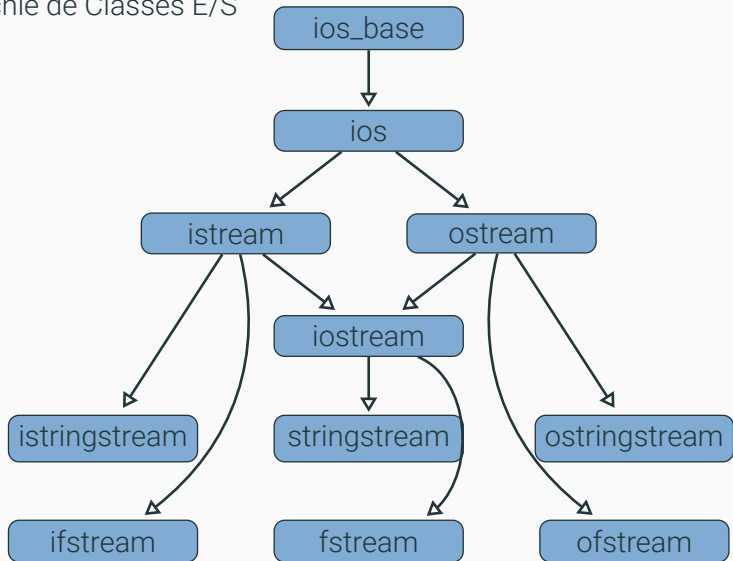
- Incrémentation : `++it;`
- Décrémentation : `--it;`
- Assignment : `*it = x;`
- Egalité : `it1 == it2;`
- Saut: `it += 21;`
- Opérateur d'élément : `it[i] = x;`

Conteneurs supportant l'accès aléatoire : **vector**, **array**, **deque**.

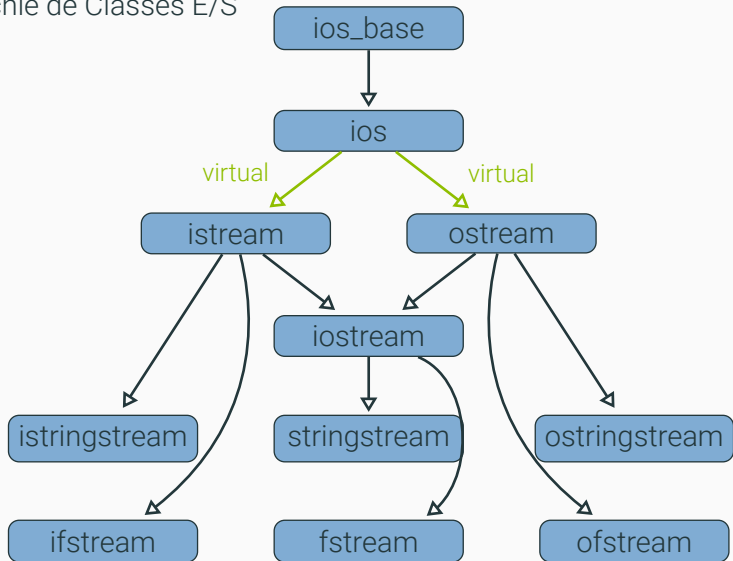
Streams est une manière standardisé de considérés les entrées/sorties des objets en **C++**

On peut créer son propre flux d'entrée ou de sortie qui ont des puits et sources différents que ceux utilisés jusqu'à présent

Hiérarchie de Classes E/S



Hiérarchie de Classes E/S



<code>(constructor)(std::string)</code>	ouvre un fichier donné
<code>operator«</code> <code>(...)</code>	écrit dans le fichier
<code>operator»</code> <code>(...)</code>	lit depuis le fichier
<code>open(std::string)</code>	ouvre un fichier donné
<code>close()</code>	ferme le buffer de fichier
<code>is_open()</code>	contrôle l'état du fichier


```
std::ofstream to_file_stream {"file.txt"};

if (!to_file_stream) {
    std::cerr << "Could not open file";
    return 1;
}

to_file_stream << "Hello world" << std::endl;
to_file_stream.close();
```

istream, ostream, stringstream

<code>(constructor)(std::string)</code>	initialise l'objet
<code>operator<<(...)</code>	écrit dans l'objet
<code>operator>>(...)</code>	lit depuis l'objet
<code>str()</code>	accède à l'objet sous jacent
<code>str(std::string)</code>	modifie la valeur de l'objet

Exemple de flux

```
void sayHello(std::ostream & os) {  
    os << "Hello" << std::endl;  
}
```

```
int main() {  
    std::ofstream ofs {"file.txt"};  
    sayHello(ofs);  
    ofs.close();  
  
    std::ostringstream oss;  
    sayHello(oss);  
  
    auto hello_string = oss.str();  
}
```

Foncteurs

Objets surchargés avec l'opérateur d'appel ()

```
struct Greater
{
    bool operator()(const double a, const double b)
    {
        return a > b;
    }
};

auto greater = Greater {};
```

Objets surchargés avec l'opérateur d'appel ()

```
auto greater = [](const double a, const double b)
{
    return a > b;
};
```

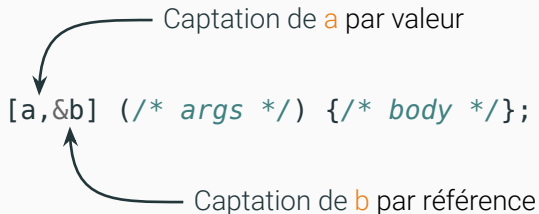
La STL propose une interface uniforme pour

```
template <class R, class... Args>  
class function<R(Args...)> {...};
```

qu'il est possible de lier avec l'opérateur d'appel correct.

```
void printInt(int i) {  
    std::cout << i;  
}  
  
struct IntPrint {  
    void operator() (int i) {  
        std::cout << i;  
    }  
};  
  
int main() {  
    std::function<void(int)> f1 = printInt;  
    std::function<void(int)> f2 = IntPrint {};  
    std::function<void(int)> f3 = [](int i){printInt(i);};  
}
```


Captation des arguments



Captation de **a** par valeur

`[a,&b] (/* args */) {/* body */};`

Captation de **b** par référence

```
[&] (/* args */) {/* body */};
```



Captation complète par référence

```
[=] (/* args */) {/* body */};
```



Captation complète par valeur

La fermeture est un concept qui permet aux fonctions de conserver des valeurs en interne

On peut utiliser la captation des fonctions lambdas

```
std::function<std::string(std::string)>
Surround(std::string surr) {
    return [surr](std::string expr) {
        return surr[0] + expr + surr[1];
    };
}

int main() {
    auto square_brackets = Surround("[ ]");
    auto quotation_marks = Surround("\"\"");

    std::cout << square_brackets("Hello") << std::endl;
    std::cout << quotation_marks("Hello") << std::endl;
}
```

Fermeture en C++

```
std::function<std::string(std::string)>
Surround(std::string surr) {
    return [surr](std::string expr) {
        return surr[0] + expr + surr[1];
    };
}
```

```
int main() {
    auto square_brackets = Surround("[ ]");
    auto quotation_marks = Surround("\"\"");

    std::cout << square_brackets("Hello") << std::endl;
    std::cout << quotation_marks("Hello") << std::endl;
}
```

affiche **[Hello]**

affiche **"Hello"**

Algorithmes

La STL se compose également d'un très grand nombre d'algorithmes

L'interface des algorithmes est conçue pour être utilisée par les itérateurs


```
template <class Itt, class T>
```

```
Itt find(Itt begin, Itt end, const T& value);
```

Trouver le premier élément égal **valeur** dans un conteneur, renvoie un itérateur sur l'élément, ou **end** si il n'est pas trouvé

```
template <class Itt, class Unary>
```

```
Unary for_each(Itt begin, Itt end, Unary f);
```

Applique la fonction **f** à chacun des éléments du conteneur
renvoie l'état final du foncteur **f**

```
template <class Itt>
```

```
void sort(Itt begin, Itt end);
```

Trie l'intervalle compris entre **begin** et **end**

```
template <class Itt, class Compare>  
  
void sort(Itt begin, Itt end, Compare compare);
```

Trie l'intervalle compris entre **begin** et **end** en utilisant une fonction de comparaison utilisateur

STL utilise l'opérateur < pour toutes les comparaisons

Égalité:

$!(a < b) \text{ and } !(b < a)$

Inégalité:

$(a < b) \text{ or } (b < a)$

Il est indispensable de bien implémenter l'opérateur <

```
std::vector<double> earnings {};  
  
// ...  
  
double total {0.};  
  
std::for_each(earnings.begin(), earnings.end(),  
    [&total](auto val){ total += val; });
```

La manipulation de la mémoire peut être risquée lorsqu'on utilise uniquement les pointeurs

Les pointeurs intelligents de la STL permettent de gérer correctement les cas complexes

Les pointeurs intelligents se comportent comme des pointeurs normaux, mais proposent des fonctionnalités complémentaires.

```
SmartPointer<Type> p {};  
auto q = p;
```



Que se passe t il ici?

L'objet possède complètement la ressource

La copie n'est pas autorisée, seulement le déplacement

```
std::unique_ptr<Type> p {};  
auto q = p; ← Erreur de compilation
```

L'objet possède complètement la ressource

La copie n'est pas autorisée, seulement le déplacement

```
std::unique_ptr<Type> p {};  
auto q = std::move(p); ← OK
```

Garde en mémoire l'ensemble des références des objets qui utilise une ressource

La ressource est détruite uniquement quand toutes les objets sont détruits

```
std::shared_ptr<Type> p {};
```

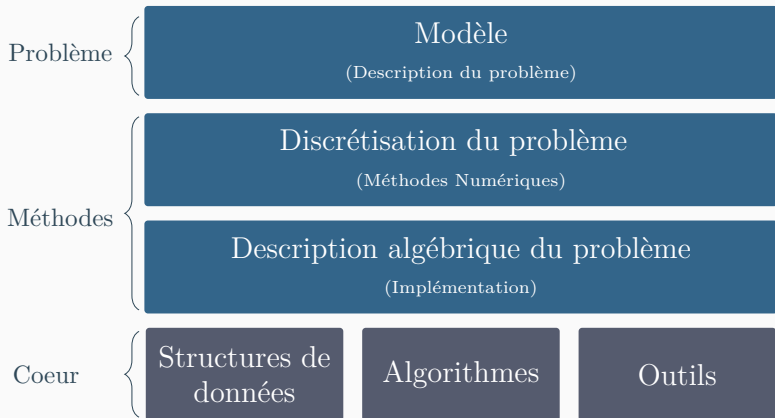
```
.use_count() = 1
```

```
auto q = p;
```

```
.use_count() = 2
```

Conclusion

- Structure de données concrètes.
- Séparer l'application en différents niveaux de réalisation.
- Exprimer les expressions mathématiques dans des langages de hauts niveaux.
- Utiliser les références pour éviter la copie de gros objets
- Utilisation des patrons de POO dans les bibliothèques populaires.



C++ Idioms

Définition

- Un idiome est une construction récurrente dans au moins 1 langage de programmation.
- En général, un idiome est l'expression simple d'un algorithme non prévu par défaut dans le langage.
- Connaître les idiomes dans un langage de programmation permet d'étendre sa compréhension.
- L'idiome est propre à un langage.
- L'idiome est moins généraliste que le design pattern ou que le paradigme de programmation.

- Objectif : s'assurer qu'un objet est bien initialisé avant son utilisation.

```
struct Bar {  
    Bar () { cout << "Bar::Bar()\n"; }  
    void f () { cout << "Bar::f()\n"; }  
};  
struct Foo {  
    Foo () { bar_.f (); }  
    static Bar bar_;  
};  
  
Foo f;  
Bar Foo::bar_;  
  
int main () {}
```

- Solution: cf. code

- Objectif : Spécialiser une classe de base à l'aide d'une classe dérivée en utilisant un patron. Permet d'augmenter les performances.
- Exemple: cf. code

- Objectifs: Créer un langage spécifique à un domaine.
Supporter l'évaluation paresseuse. Passer des expressions comme paramètre d'une fonction.

- C++ ne supporte pas par défaut le passage d'expression.
Exemple

```
int x;  
foo(x + x + x);
```

- Solution: cf. code.

Déduction du type de retour

- Objectif : déduire le type de la variable qui doit être assignée ou initialisée afin d'optimiser les opérations dans le contexte.

```
template <class Container>
Container getRandomN(size_t n)
{
    Container c;
    for(size_t i = 0; i < n; ++i)
        c.insert(c.end(), rand());
    return c;
}

int main (void)
{
    std::list<int> l = getRandomN<std::list<int> > (10);
    std::vector<long> v = getRandomN<std::vector<long> > (100);
}
```

- Solution : cf. code