

? – Introduction à la STL : *Standard Template Library*

Julien Deantoni

Plan

- **Introduction**
- Contenu de la STL
 - Espace de nommage
 - Exception
 - Conteneurs
 - Itérateurs
 - Algorithmes
- Conclusion

Introduction

- Volonté d'apporter aux programmeurs C++ un canevas de programmation
 - **Fiable** : largement utilisé, écrit par des spécialistes
 - **Efficace** : sûrement plus efficace qu'un code “maison”
 - **Générique** : utilisation intensive de template, intégré au standard (facilite la réutilisabilité)
 - **Compréhensible** : tout programmeur doit pouvoir lire un code utilisant la STL

Plan

- Introduction
- **Contenu de la STL**
 - **Espace de nommage**
 - Exception
 - Conteneurs
 - Itérateurs
 - Algorithmes
- Conclusion

Espace de nommage

principes

- Mécanisme de remplacement partiel d'un *package*
 - Imbrication possible
 - Définition possible dans plusieurs fichiers
 - Pas de lien avec la notion de visibilité
 - soient deux classes **A** et **B**. **A** a les mêmes privilèges d'accès aux membres de B qu'ils soient dans le même *package* ou pas; et réciproquement.

Espace de nommage

mise en œuvre

- Déclaration / Définition

```
namespace itsName;    //déclaration
```

```
namespace itsName //définition
{
    class A
    {
        // définition de la classe A
    };

    const double PI=3.1415927;

    class B
    {
        // définition de la classe B
    };
}
```

Espace de nommage

mise en œuvre

- Accès aux membres
 - Au sein d'un même espace de nommage
 - Rien de particulier

```
namespace itsName           //définition
{
    class A
    {
        // définition de la classe A
    };

    const double PI=3.1415927;

    class B
    {
        A myA;
        ...
    };
}
```

- En dehors de l'espace de nommage
 - Nom qualifié
 - La clause *using namespace*

Espace de nommage

mise en œuvre

- Accès aux membres
 - Au sein d'un même espace de nommage
 - Rien de particulier
 - En dehors de l'espace de nommage
 - Nom qualifié

```
namespace itsName           //définition
{
    class A
    {
        // définition de la classe A
    };

    const double PI=3.1415927;
}
class B
{
    itsname::A myA;
    ...
}
```

- La clause *using namespace*

Espace de nommage

mise en œuvre

- Accès aux membres
 - Au sein d'un même espace de nommage
 - Rien de particulier
 - En dehors de l'espace de nommage
 - Nom qualifié

```
namespace itsName           //définition 1
{
    namespace anotherName //définition 2
    {
        class A
        {
            // définition de la classe A
        };
    }
}

class B
{
    itsname::anotherName::A myA;
    ...
}
```

- La clause *using namespace*

Espace de nommage

mise en œuvre

- Accès aux membres
 - Au sein d'un même espace de nommage
 - Rien de particulier
 - En dehors de l'espace de nommage
 - Nom qualifié
 - La clause *using namespace*



```
namespace itsName           //définition
{
    class A
    {
        // définition de la classe A
    };
}

using namespace itsName;
class B
{
    A myA;
    ...
}
```

Espace de nommage

mise en œuvre

- Accès aux membres
 - Au sein d'un même espace de nommage
 - Rien de particulier
 - En dehors de l'espace de nommage
 - Nom qualifié
 - La clause *using namespace*

```
namespace itsName           //définition
{
    class A
    {
        // définition de la classe A
    };
    const double PI=3.1415927;
}
using itsName::A;
class B
{
    A myA;
    double myPI = itsName::PI;
}
```

Espace de nommage

mise en œuvre

- Diverses utilisations

```
namespace Color {  
    enum Color {Red,Blue,Green } ;  
};  
  
namespace Apple {  
    enum Apple {Golden, Red, Green } ;  
};
```

```
-----  
  
Apple fruit_kind = Apple::Red ;  
Color fruit_color = Color::Red;
```

Évite les confusions....

Espace de nommage

mise en œuvre

- Diverses utilisations

```
//renommage  
namespace fs = boost::filesystem ;
```

Facilite l'utilisation de *namespaces* trop verbeux....

Plan

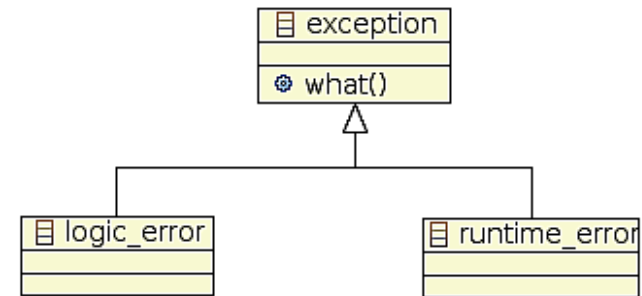
- Introduction
- Contenu de la STL
 - Espace de nommage
 - **Exception**
 - Conteneurs
 - Itérateurs
 - Algorithmes
- Conclusion

Exceptions de la STL

principes

- **exception**

- `#include <exception>`
- `#include <stdexcept>`
- Fonction `What()` doit être redéfinie (message intelligible)



- **logic_error**

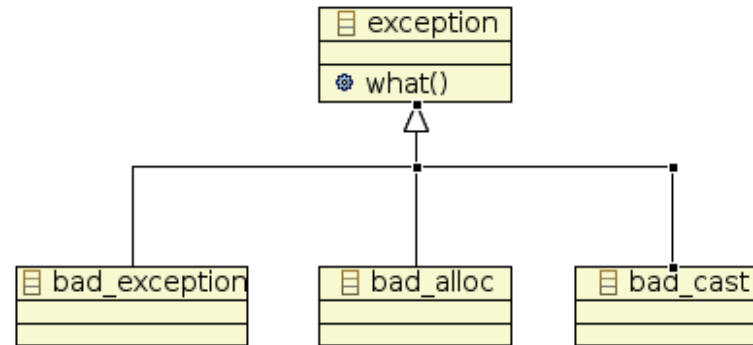
- Erreur due à la logique interne d'un programme
- Permettre de réagir pour ajouter de la robustesse à un programme

- **runtime_error**

- Erreur liée à l'environnement du programme
- Problème de précision / défaillance du matériel

Exceptions de la STL

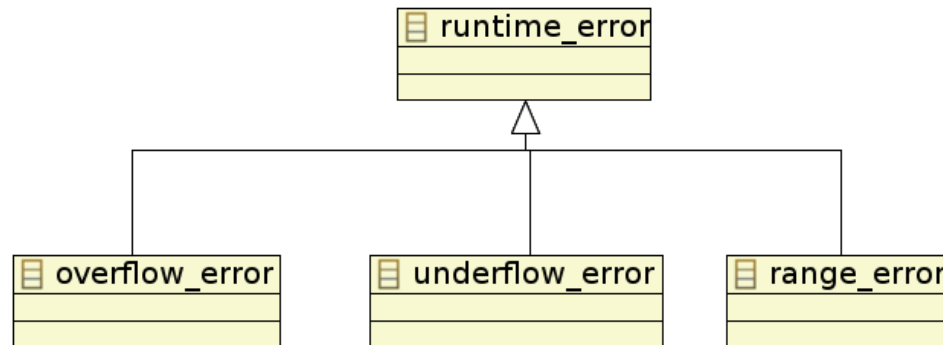
principes



- **exception**
 - **bad_exception** : exception non prévue dans la déclaration de la fonction
 - **bad_alloc** : problème d'allocation lors de l'exécution de new
 - **bad_cast** : mauvaise utilisation de dynamic_cast

Exceptions de la STL

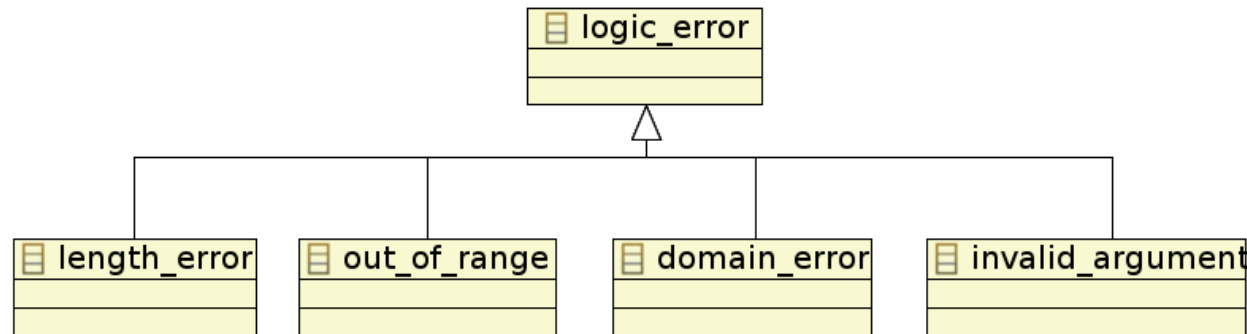
principes



- **runtime_error**
 - **overflow_error / underflow_error** : problème de précision sur les calculs arithmétiques
 - **range_error** : erreur interne de calcul d'un argument (non liée au programmeur)

Exceptions de la STL

principes



- **logic_error**

- **length_error** : tentative de création d'un objet de trop grande taille
- **out_of_range** : typiquement, un accès en dehors des limites d'index d'un tableau
- **domain_error / invalid_argument** : passage d'un argument de type ou de domaine inattendu

Exceptions de la STL

mise en œuvre

Lancer une exception

```
class Rationnal
{
    int num;
    int den;

    void setDen(int d) throw (std::invalid_argument)
    {
        if (d == 0)
            throw std::invalid_argument("Rationnel::setDen, denominator is 0");

        den=d;
    }

    double asDouble () throw (std::range_error)
    {
        if (den == 0)
            throw std::range_error("Rationnel::asDouble : division by 0");

        return static_cast<double> (num)/den;
    }
};
```

Exceptions de la STL

mise en œuvre

Lancer une exception

```
class Rationnal
{
    int num;
    int den;

    void setDen(int d) throw (std::invalid_argument)
    {
        if (d == 0)
            throw std::invalid_argument("Rationnel::setDen, denominator is 0");

        den=d;
    }

    double asDouble () throw (std::range_error)
    {
        if (den == 0)
            throw std::range_error("Rationnel::asDouble : division by 0");

        return static_cast<double> (num)/den;
    }
};
```

Throw list : permet de spécifier les exceptions possiblement lancées par la fonction
non obligatoire

Exceptions de la STL

mise en œuvre

Lancer une exception

```
class Rationnal
{
    int num;
    int den;

    void setDen(int d) throw (std::invalid_argument)
    {
        if (d == 0)
            throw std::invalid_argument("Rationnel::setDen, denominator is 0");
        den=d;
    }

    double asDouble () throw (std::range_error)
    {
        if (den == 0)
            throw std::range_error("Rationnel::asDouble : division by 0");

        return static_cast<double> (num)/den;
    }
};
```

Creation d'une instance de classe `std::invalid_argument`. Initialisation du message renvoyé lors de l'appel à `What()`

Exceptions de la STL

mise en œuvre

gérer une exception

```
int nouveauDen = 0;
bool OK = true;
...
do
{
    OK = true;
    std::cout << "Saisissez un denominateur";
    std::cin >> nouveauDen ;

    try
    {
        r1.setDen(nouveauDen);
    }
    catch (std::invalid_argument& e)
    {
        std::cout << e.what();
        OK=false;
    }
}while(OK);
```

Exceptions de la STL

mise en œuvre

gérer une exception

```
int nouveauDen = 0;
bool OK = true;
...
do
{
    OK = true;
    std::cout << "Saisissez un nouveau denominateur";
    std::cin >> nouveauDen ;

    try
    {
        r1.setDen(nouveauDen);
    }
    catch (std::invalid_argument &e)
    {
        std::cout << e.what();
        OK=false;
    }

}while(OK);
```

Block pendant lequel les exceptions sont « surveillées »

Exceptions de la STL

mise en œuvre

gérer une exception

```
int nouveauDen = 0;
bool OK = true;
...
do
{
    OK = true;
    std::cout << "Saisissez un nouveau denomina
    std::cin >> nouveauDen ;

    try
    {
        r1.setDen(nouveauDen);
    }
    catch (std::invalid_argument& e)
    {
        std::cout << e.what();
        std::cout << "Dans la fonction foo(), setDen() failed";
        OK=false;
    }
}while(OK);
```

Block exécuté lors de la
détection de l'exception
std::invalid_argument

Exceptions de la STL

mise en œuvre

gérer une exception

```
int nouveauDen = 0;
bool OK = true;
...
do
{
    OK = true;
    std::cout << "Saisissez un nouveau denominateur";
    std::cin >> nouveauDen;
    try
    {
        r1.setDen(nouveauDen);
    }
    catch (std::invalid_argument &e)
    {
        std::cout << e.what();
        std::cout<<("Dans la fonction foo(), setDen() failed");
        OK=false;
    }
    catch (...)
    {
        std::cout << "Dans la fonction foo(), une autre exception a été détectée ";
    }
}while(OK);
```

Block exécuté lors de la détection de n'importe quelle exception

Plan

- Introduction
- Contenu de la STL
 - Espace de nommage
 - Exception
 - **Conteneurs**
 - Itérateurs
 - Algorithmes
- Conclusion

Trois principes fortement liés de la STL

- **Conteneurs**
 - Collection de types homogènes
 - Structures de données classiques
 - Interfaces homogènes entre les différents conteneurs
- **Itérateurs**
 - Manière systématique et homogènes de parcourir un conteneur quelque soit son type
- **Algorithmes**
 - Opération générique sur l'ensemble des éléments d'un conteneur

Conteneurs de la STL

Les différents types

- Séquences élémentaires
 - Vecteurs, listes, DQ (double ended queue)
- Spécialisation des séquences élémentaires
 - Pile (Stack), File (Queue), File à priorité
- Conteneurs associatifs
 - Map, Set

Conteneurs de la STL

description

Séquences élémentaires (a.k.a. Conteneurs élémentaires)

<code>vector<T></code>	<code><vector></code>	Tableau avec (re)allocation automatique
<code>deque<T></code>	<code><deque></code>	Tableau avec indexation optimisée (début et fin)
<code>list<T></code>	<code><list></code>	Doubly linked sequence (list)

Spécialisations de séquences élémentaires

par défaut spécialise **deque** (sauf **priority_queue**, qui utilise **vector**)

<code>stack<T></code>	<code><stack></code>	Last In First Out (LIFO)
<code>queue<T></code>	<code><queue></code>	First In First Out (FIFO)
<code>priority_queue<T></code>	<code><queue></code>	Queue à priorité (<code>operator<()</code>)

Conteneurs associatifs (basé sur une paire (clef, valeur))

<code>map<K, T></code>	<code><map></code>	Tableau associatif sans duplication (K est la clef)
<code>multimap<K, T></code>	<code><map></code>	Tableau associatif avec duplication (K est la clef)
<code>set<K></code>	<code><set></code>	Set (sans duplication) of K
<code>multiset<K></code>	<code><set></code>	Set (avec duplication) of K (a.k.a. <i>Bag</i>)

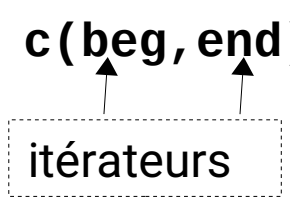
Conteneurs de la STL

complexité des fonctions associées

	Indexation	Insertion partout	Operation en début	Operation en fin
vector	$O(1)$	$O(n)+$		$O(1)+$
list		$O(1)$	$O(1)$	$O(1)$
deque	$O(1)$	$O(n)$	$O(1)$	$O(1)$
stack		$O(n)$		$O(1)$
queue		$O(n)$	$O(1)$	$O(1)$
priority_queue		$O(\log n)$	$O(\log n)$	
map	$O(\log n)+$	$O(\log n)+$		
multimap		$O(\log n)+$		
set		$O(\log n)+$		
multiset		$O(\log n)+$		

Conteneurs de la STL

mise en œuvre

Construction	
<i>Cont</i> c;	Crée un conteneur vide
<i>Cont</i> c1(c2);	Crée un conteneur par copie (les éléments de c1 seront une copie de ceux de c2)
<i>Cont</i> c(beg, end); <div style="text-align: center;">  </div>	Crée un conteneur contenant les valeur située dans [beg, end[; rq: les types des valeurs doivent être identiques
<i>Cont</i> c(n)	Crée un conteneur de taille n
<i>Cont</i> c(n, x)	Crée un conteneur de taille n dont les valeurs sont initialisées à x

En gris, seulement pour les séquences élémentaires

Conteneurs de la STL

mise en œuvre

Opérations sur la taille	
<code>c.size()</code>	Nombre d'éléments dans le conteneur
<code>c.empty()</code>	Vrai si le conteneur ne contient aucun élément
<code>c.max_size()</code>	Nombre maximum d'élément défini par l'implémentation du conteneur
<code>c.resize(n)</code>	Change la taille à n ; si cela augmente la taille, les nouveaux éléments sont initialisé à leur valeur par défaut (zéro du type)
<code>c.resize(n,e)</code>	Change la taille à n ; si cela augmente la taille, les nouveaux éléments sont initialisé à e
<code>c.capacity()</code>	Nombre maximum d'élément que peut contenir un vecteur avant redimensionnement

En bleu, seulement pour les vecteurs

En gris, seulement pour les séquences élémentaires

Conteneurs de la STL

```
#include <functional>
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>
```

```
using std::cout;
using std::endl;
using std::vector;
using std::string;
```

```
int main(){
```

```
    int i=1,j=2,k=3;
```

```
    vector<std::reference_wrapper<int>> v1;
    vector<std::reference_wrapper<int>> v2;
```

```
    cout << "v1 capacity/size: " << v1.capacity() << "/" << v1.size() << endl;
```

```
    v1.push_back(i);
    cout << "v1 capacity/size: " << v1.capacity() << "/" << v1.size() << endl;
```

```
    v1.push_back(j);
    cout << "v1 capacity/size: " << v1.capacity() << "/" << v1.size() << endl;
```

```
    v1.push_back(k);
    cout << "v1 capacity/size: " << v1.capacity() << "/" << v1.size() << endl;
```

```
    v1.push_back(i);
    cout << "v1 capacity/size: " << v1.capacity() << "/" << v1.size() << endl;
```

```
    v1.push_back(j);
    cout << "v1 capacity/size: " << v1.capacity() << "/" << v1.size() << endl;
```

```
    v1.push_back(k);
    cout << "v1 capacity/size: " << v1.capacity() << "/" << v1.size() << endl;
```

```
v1 capacity/size: 0/0
v1 capacity/size: 1/1
v1 capacity/size: 2/2
v1 capacity/size: 4/3
v1 capacity/size: 4/4
v1 capacity/size: 8/5
v1 capacity/size: 8/6
```

Conteneurs de la STL

mise en œuvre

- Classes d'opérations
 - **Accès aux éléments** (premier, dernier, indexé, aléatoire, etc)
 - **Insertion / suppression** (début, fin, index, élément)
 - **Comparaison entre conteneurs** ($=$, \neq , $>$, $<$, etc)
 - **Affectation** ($=$, swap, etc)

Conteneurs de la STL

conteneurs associatifs

- Maps (et multimaps)

```
map<string, int> m;  
  
m["Perus"] = 164468443;  
m["Arnault"] = 164468424;  
m["Jouvin"] = 164468932;  
m["Lefebvre"] = 169155240;  
m["Garnier"] = 164468551;  
m["Hrivnacova"] = 169156594;
```

Conteneurs de la STL

conteneurs associatifs

- Maps (et multimaps)
 - Leurs valeurs sont des paires
 - `pair<const K, T>`

```
map<string, int> m;  
pair<string, int> p;  
const string key_value="Crncovik";  
  
p = make_pair(key_value, 12456123);  
//p.first() accède à key_value  
//p.second() accède à l'élément 12456123
```

Conteneurs de la STL

conteneurs associatifs

- Maps (et multimaps)
 - Leurs valeurs sont des paires
 - `pair<const K, T>`
 - On peut utiliser une indexation classique sur `k` qui renvoie `T`
 - Si l'élément n'existe pas, création d'un emplacement (une entrée) avec l'élément initialisé au zéro de son type

```
map<string, int> m;  
pair<string, int> p;  
const string key_value="hello";  
  
p = make_pair(key_value, 12);  
m.insert(m.begin(), p);  
    //Iterator m.begin() ne sert à rien  
  
cout << m[key_value];
```

➡ `./ourTest`
`12`

Conteneurs de la STL

conteneurs associatifs

- Maps (et multimaps)
 - Leurs valeurs sont des paires
 - `pair<const K, T>`
 - On peut utiliser une indexation classique sur `k` qui renvoie `T`
 - Si l'élément n'existe pas, création d'un emplacement (une entrée) avec l'élément initialisé au zéro de son type

```
map<string, int> m;  
pair<string, int> p;  
const string key_value="hello";  
  
m[key_value] = 12;  
  
cout<< m[key_value];  
cout<< m["hello"];
```

➡ `./ourTest`
`12 12`

Conteneurs de la STL

conteneurs associatifs

- Maps (et multimaps)
 - Leurs valeurs sont des paires
 - `pair<const K, T>`
 - On peut utiliser une indexation classique sur `k` qui renvoie `T`
 - Si l'élément n'existe pas, création d'un emplacement (une entrée) avec l'élément initialisé au zéro de son type

```
map<string, int> m;
```

```
cout<< m["nothing"];
```

```
//crée une nouvelle entrée dans la map
```



\$/OurTest
0

Conteneurs de la STL

conteneurs associatifs

- Maps (et multimaps)
 - Leurs valeurs sont des paires
 - `pair<const K, T>`
 - On peut utiliser une indexation classique sur **k** qui renvoie **T**
 - Si l'élément n'existe pas, création d'un emplacement (une entrée) avec l'élément initialisé au zéro de son type

```
map<string, int> m;  
map<string, int>::iterator it= m.find("nothing");  
if( it != m.end() ){  
    cout<< m["nothing"];  
}
```


Conteneurs de la STL

conteneurs associatifs

- Maps (et multimaps)
 - Leurs valeurs sont des paires
 - `pair<const K, T>`
 - On peut utiliser une indexation classique sur `k` qui renvoie `T`
 - Si l'élément n'existe pas, création d'un emplacement (une entrée) avec l'élément initialisé au zéro de son type

```
map<string, int> m;  
auto it= m.find("nothing");  
if( it != m.end() ){  
    cout<< m["nothing"];  
}
```

Conteneurs de la STL

conteneurs associatifs

- Maps (et multimaps)
 - Leurs valeurs sont des paires
 - `pair<const K, T>`
 - **K** est une information utile à l'indexation
 - On peut utiliser une indexation classique sur **k** qui renvoie **T**



- K doit être une constante (pour tous les types associatifs)
 - Ne pas changer la valeur de K car elle est utilisée pour l'indexation !!

Un autre conteneur

Class string

- Spécialisation classique de la classe `basic_string`
 - `String = basic_string<char>`
- Propriétés (non exhaustif)
 - Construction, destruction, conversion depuis `char *`
 - Conversion vers `char * (c_str)`
 - Copies
 - Extraction de sous-chaîne (`substr`)
 - Concaténation (`operator+`, `append...`)
 - Comparaison (`==`, `!=`, ...)

remarques

- **Contraintes sur les éléments d'un conteneur**
 - Constructeur par défaut
 - Définit le zéro d'un type !
 - Opérateurs de copies *//et donc pas de références*
 - Pour les conteneurs “triés”, l'opérateur '<' (“less than”)
 - Remarque : si l'opérateur d'égalité n'est pas défini, l'opérateur '<' est utilisé :
$$a == b \equiv !(a < b) \ \&\& \ !(b < a)$$

Conteneurs de la STL

remarques

- **Contraintes sur les éléments d'un conteneur**
 - Constructeur par défaut
 - Définit le zéro d'un type !
 - Opérateurs de copies *//et donc pas de références ?*

```
#include <functional>
#include <vector>
int main()
{
    std::vector<std::reference_wrapper<int>> iv;
    int a = 12;
    iv.push_back(a); // or std::ref(a)
    // now iv = { 12 };
    a = 13;
    // now iv = { 13 };
}
```

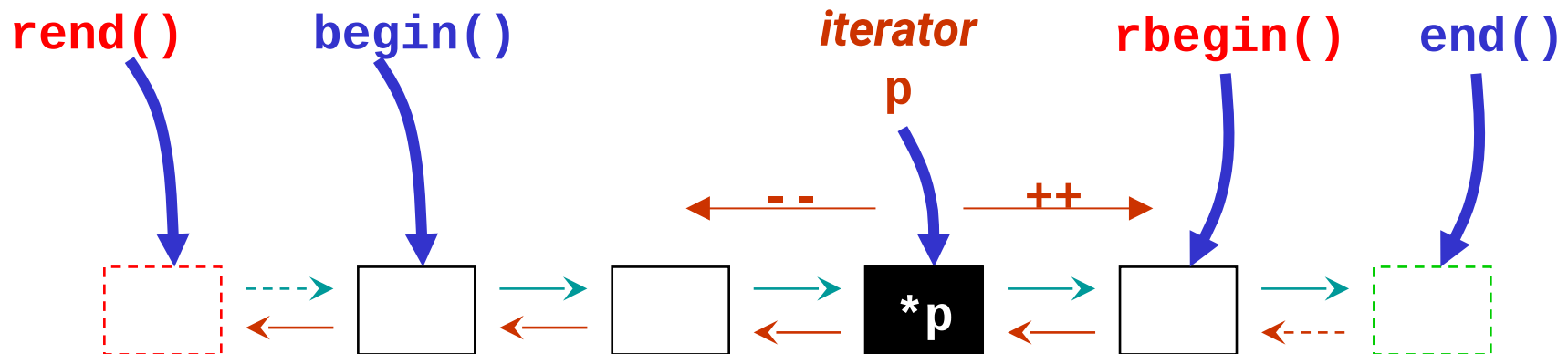
Plan

- Introduction
- Contenu de la STL
 - Espace de nommage
 - Exception
 - Conteneurs
 - **Itérateurs**
 - Algorithmes
 - Les entrées / sorties

Itérateurs de la STL

principes

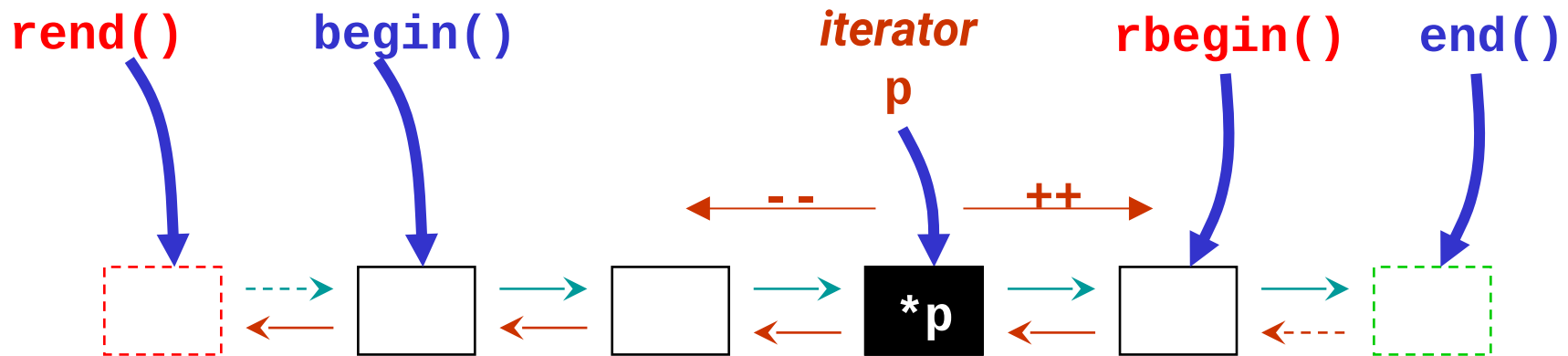
- Peuvent être vu comme un pointeur intelligent sur une liste doublement chaînée



Itérateurs de la STL

principes

- Peuvent être vu comme un pointeur intelligent sur une liste doublement chaînée



```
list<int> L;  
for (int i = 0; i < N; i++) L.push_front(i);
```

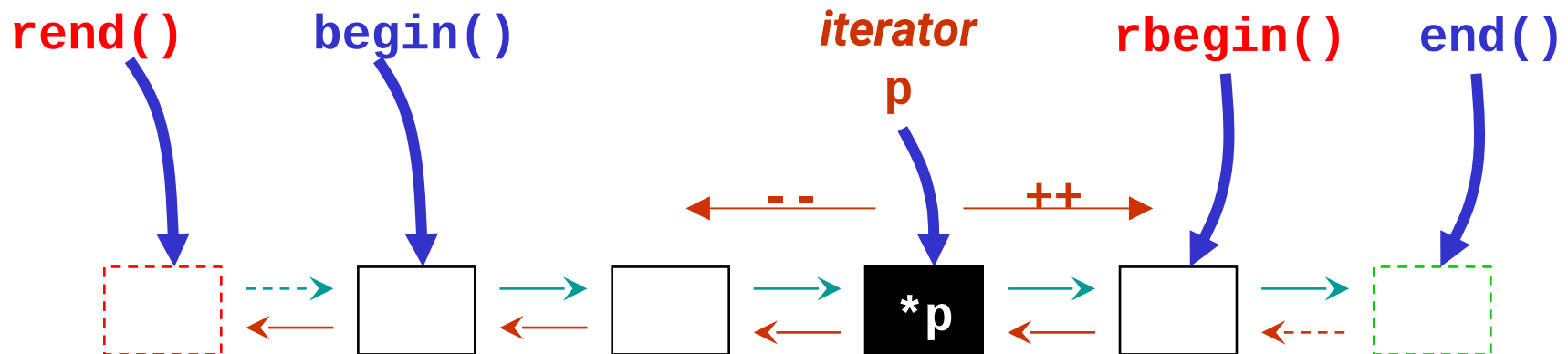
```
list<int>::iterator it;
```

```
for (it = L.begin(); it != L.end(); ++it)  
    cout << *it << ' ';
```


Itérateurs de la STL

principes

- Peuvent être vu comme un pointeur intelligent sur une liste doublement chaînée



```
list<int> L;  
for (int i = 0; i < N; i++) L.push_front(i);
```

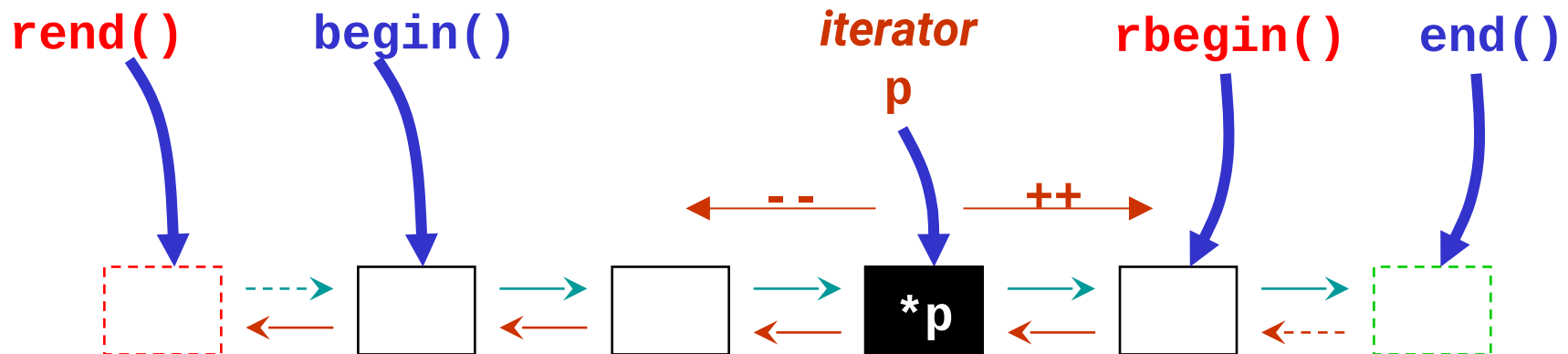
```
list<int>::iterator it;
```

```
for (it = L.begin(); it != L.end(); ++it)  
    *it=0;
```

Itérateurs de la STL

principes

- Peuvent être vu comme un pointeur intelligent sur une liste doublement chaînée



```
list<int> L;  
for (int i = 0; i < N; i++) L.push_front(i);
```

```
list<int>::const_iterator it;
```

```
for (it = L.begin(); it != L.end(); ++it)  
    cout << *it << ' ';
```

Itérateurs de la STL

principes

- Il existe de nombreux types d'itérateurs plus ou moins restrictifs mais 4 sont incontournables
 - Iterator
 - const_iterator
 - reverse_iterator
 - const_reverse_iterator

```
list<int> L;  
for (int i = 0; i < N; i++) L.push_front(i);
```

```
list<int>::const_iterator it;
```

```
for (it = L.begin(); it != L.end(); ++it)  
    cout << *it << ' ';
```

Itérateurs de la STL

principes

- Il existe de nombreux types d'itérateurs plus ou moins restrictifs mais 4 sont incontournables
 - Iterator
 - const_iterator
 - reverse_iterator
 - const_reverse_iterator

```
list<int> L;  
for (int i = 0; i < N; i++) L.push_front(i);
```

```
list<int>::const_iterator it;
```

```
for (it = L.begin(); it != L.end(); ++it)  
    cout << *it << ' ';
```



```
$./OurTest  
N ... 3 2 1 0
```

Itérateurs de la STL

principes

- Il existe de nombreux types d'itérateurs plus ou moins restrictifs mais 4 sont incontournables
 - Iterator
 - const_iterator
 - reverse_iterator
 - const_reverse_iterator

```
list<int> L;  
for (int i = 0; i < N; i++) L.push_front(i);
```

```
list<int>::const_reverse_iterator it;
```

```
for (it = L.rbegin(); it != L.rend(); ++it)  
    cout << *it << ' ';
```



```
$./OurTest  
0 1 2 3 ... N
```

Containers and Iterators

Models of Iterators

- Input iterator
 - Move forward only ($++$)
 - May provide only read access
- Output iterator
 - Move forward only
 - Provide read/write access
 - Do not have comparison operators
- Forward iterator
 - Both input and output
 - Have comparison operators
- Bidirectional iterator
 - Move both ways ($++$, $--$)
- Random access iterators
 - Bidirectional
 - Have arithmetic and comparison operators
- The model of iterator depends on the type of the container
- Member functions and algorithms specify which model they need

Itérateurs de la STL

principes

- Comme illustré dans l'exemple, les itérateurs peuvent borner un parcours... MAIS

Les itérateurs définissent un interval dont la borne gauche est incluse et la droite excluse [... [

```
list<int> L;  
for (int i = 0; i < N; i++) L.push_front(i);  
  
auto it;  
  
for (it = L.begin(); it != L.end(); ++it)  
    cout << *it << ' ';
```

Itérateurs de la STL

principes

- Comme illustré dans l'exemple, les itérateurs peuvent borner un parcours... MAIS

Les itérateurs définissent un interval dont la borne gauche est incluse et la droite exclue [... [

```
list<int> L;  
for (int i = 0; i < N; i++) L.push_front(i);  
  
for (auto it = L.begin(); it != L.end(); ++it)  
    cout << *it << ' ';
```


Itérateurs de la STL

mise en œuvre

Interface d'Itérateur

<code>c.begin()</code>	Renvoie un itérateur sur le premier élément
<code>c.end()</code>	Renvoie un itérateur juste après le dernier élément
<code>c.rbegin()</code>	Idem <code>c.begin()</code> mais en ordre inverse
<code>c.rend()</code>	Idem <code>c.end()</code> mais en ordre inverse

Exemple de fonction utilisant des itérateurs

<code>c.insert(pos, e)</code>	Insert un élément e . La valeur retournée ainsi que l'interprétation de pos dépend du type de conteneur
<code>c.erase(beg, end)</code>	Efface tous les éléments dans l'intervall [beg, end[

For loop : syntactic sugar

```
vector<int> v = {1, 2, 3, 4, 5, 6, 7} ;

for (const int& i : v) // access by const reference
    std::cout << i << ' ';

for (auto i : v) // access by copy
    std::cout << i << ' ';

for (auto& i : v) // access by rvalue reference
    std::cout << i << ' ';

for (int n : {0, 1, 2, 3, 4, 5}) // looping on a braced-init-list
    std::cout << n << ' ';

int a[] = {0, 1, 2, 3, 4, 5};
for (int n : a) // looping on an array
    std::cout << n << ' ';
```

Plan

- Introduction
- Contenu de la STL
 - Espace de nommage
 - Exception
 - Conteneurs
 - Itérateurs
 - **Algorithmes**
 - **Les entrées / sorties**

Algorithmes

principes

- Les algorithmes sont des *function templates*
- Ils opèrent sur une collection d'objets
- Ils visitent les éléments d'une collection et appliquent la même opération sur chacun
- Le résultat global est fonction de l'ensemble des résultats sur les éléments
- Le type de retour dépend de la fonctionnalité de l'algorithme
- À inclure pour les utiliser : `<algorithm>`

Algorithmes

qui font quoi ?

- Environ 60 algorithmes prédéfinis
 - Certains ne modifient pas le conteneur...
 - D'autres si !
 - Algorithmes simples (find, count, ...)
 - Algorithmes sous ensemblistes (copy, copy_n, ...)
 - Algorithmes moins simples :-) (transform, replace_copy_if, ...)
 - Algorithme de trie, de fusion (sort, merge, ...)
 - Algorithmes sur les collections triées (set_union , set_intersection, ...)
 - Algorithmes numériques (partial_sum, ...)

principes

- Les algorithmes utilisent doublement les itérateurs
 - (1) Pour spécifier leur interval d'action
 - (2) Pour récupérer le résultat

```
list<int> L;  
list<int>::iterator it;  
it = find(L.begin(), L.end(), 7);
```

- Les autres paramètres spécifie l'action à réaliser sur les éléments

```
list<int> L;  
list<int>::iterator it;  
it = find(L.begin(), L.end(), 7);
```

exemple

- Compter les éléments supérieur à 4

```
bool is_gt_4(int i) {  
    return i > 4;  
}
```

```
list<int> L;  
int n = count_if(L.begin(), L.end(), is_gt_4);
```

Algorithmes

exemple

- Compter les éléments supérieur à 4

```
bool is_gt_4(int i) {  
    return i > 4;  
}
```

```
list<int> L;  
int n = count_if(L.begin(), L.end(), is_gt_4);
```

Pointeur sur fonction

Algorithmes

principes

- Transformation de séquence :
 - Passage d'une liste de chaîne de caractère en un vector d'entier

```
int string_length(string s) {  
    return s.size();  
}  
  
list<string> LS(10, "hello ");  
vector<int> VI(LS.size());  
transform(LS.begin(),LS.end(),VI.begin(),string_length);
```

principes

- Transformation de séquence :
 - Passage d'une liste de chaîne de caractère en un vector d'entier

```
int string_length(string s) {  
    return s.size();  
}  
list<string> LS(10, "hello ");  
vector<int> VI(LS.size());  
  
transform(LS.begin(), LS.end(), VI.begin(), string_length);
```



Aucun algorithme de la STL ne modifie la taille d'un conteneur

Algorithmes

de plus prêt

- Comment est construit un algorithme ?

```
template <typename InputIterator, typename Predicate>
int count_if(InputIterator beg, InputIterator end,
             Predicate pred)
{
    int n = 0;
    for (InputIterator it = beg; it != end; ++it) {
        if (pred(*it)) ++n;
    }
    return n;
}
```

- **pred** peut être un pointeur de fonction ou une instance de classe définissant un opérateur : **operator()**

- Un **functor** est une classe définissant **operator()**
 - Ces instances sont des **function objects**

```
class Is_Gt_4 {  
public:  
    bool operator()(int i) {  
        return i > 4;  
    }  
};  
  
list<int> L;  
// ...  
Is_Gt_4 comp_op; // a function object  
  
int n = count_if(L.begin(), L.end(), comp_op);
```

- Un **functor** est une classe définissant **operator()**
 - Ces instances sont des **function objects**

```
class Is_Gt_4 {  
public:  
    bool operator()(int i) {  
        return i > 4;  
    }  
};  
  
list<int> L;  
// ...  
Is_Gt_4 comp_op; // a function object  
int n = count_if(L.begin(), L.end(), comp_op);  
  
int n = count_if(L.begin(), L.end(), Is_Gt_4());
```

- Passage d'opérateurs :

```
list<int> L;  
// ...  
transform(L.begin(), L.end(), L.begin(), operator-);
```

- Passage d'opérateurs :

```
list<int> L;  
// ...  
transform(L.begin(), L.end(), L.begin(), operator-);
```

→ Ne compile pas !

- C++ ne peut pas deviner le type exact de la fonction à appeler (il existe de nombreuses surcharges de cet opérateur)

```
transform(L.begin(), L.end(), L.begin(), negate<int>());
```

Algorithmes

de plus prêt

- Réification des opérateurs C++: functors prédéfinis

<code>negate<T>(p)</code>	<code>-p</code>
<code>plus<T>(p1, p2)</code>	<code>p1 + p2</code>
<code>minus<T>(p1, p2)</code>	<code>p1 - p2</code>
<code>multiplies<T>(p1, p2)</code>	<code>p1 * p2</code>
<code>divides<T>(p1, p2)</code>	<code>p1 / p2</code>
<code>equal_to<T>(p1, p2)</code>	<code>p1 == p2</code>
<code>not_equal_to<T>(p1, p2)</code>	<code>p1 != p2</code>
<code>less<T>(p1, p2)</code>	<code>p1 < p2</code>
(and greater, less_equal, greater_equal)	
<code>logical_not<T>(p)</code>	<code>!p</code>
<code>logical_and<T>(p1, p2)</code>	<code>p1 && p2</code>
<code>logical_or<T>(p1, p2)</code>	<code>p1 p2</code>

Algorithmes

de plus prêt

- Les algorithmes ne modifient jamais la taille d'un conteneur
 - Ils peuvent modifier l'ordre des éléments
 - Ainsi que la valeur des éléments

```
deque<double> dq(10, 3.14); // 10 elements
list<double> ld; // empty by default
copy(dq.begin(), dq.end(), ld.begin());
// erreur fatale !!
```

de plus prêt

- Des itérateurs spéciaux :
 - **insérer, back_insérer, front_insérer**
 - ➔ Les opérations ++ (et --) sur ces itérateurs allouent de l'espace mémoire pour les éléments

```
deque<double> dq(10, 3.14); // 10 elements  
list<double> ld; // empty by default  
copy(dq.begin(), dq.end(), back_insérer(ld));  
// les nouveaux éléments sont insérés à la fin de la liste!!
```

fait maison (DIY)

```
template<class InputIt>
void delete_each(InputIt first, InputIt last)
{
    for (; first != last; ++first) {
        delete *first;
    }
    return;
}
```

```
delete_each(collectionOfPointers.begin(), collectionOfPointers.end());
```

Algorithmes

fait maison (DIY)

```
template<class InputIt, class MemberFunction>
MemberFunction for_eachCallF(InputIt first, InputIt last, MemberFunction f)
{
    for (; first != last; ++first) {
        ((*first)->*f)();
    }
    return f;
}
```

```
for_eachCallF( collectionOfPointers.begin(),
               collectionOfPointers.end(),
               &Object::draw
               );
```

Plan

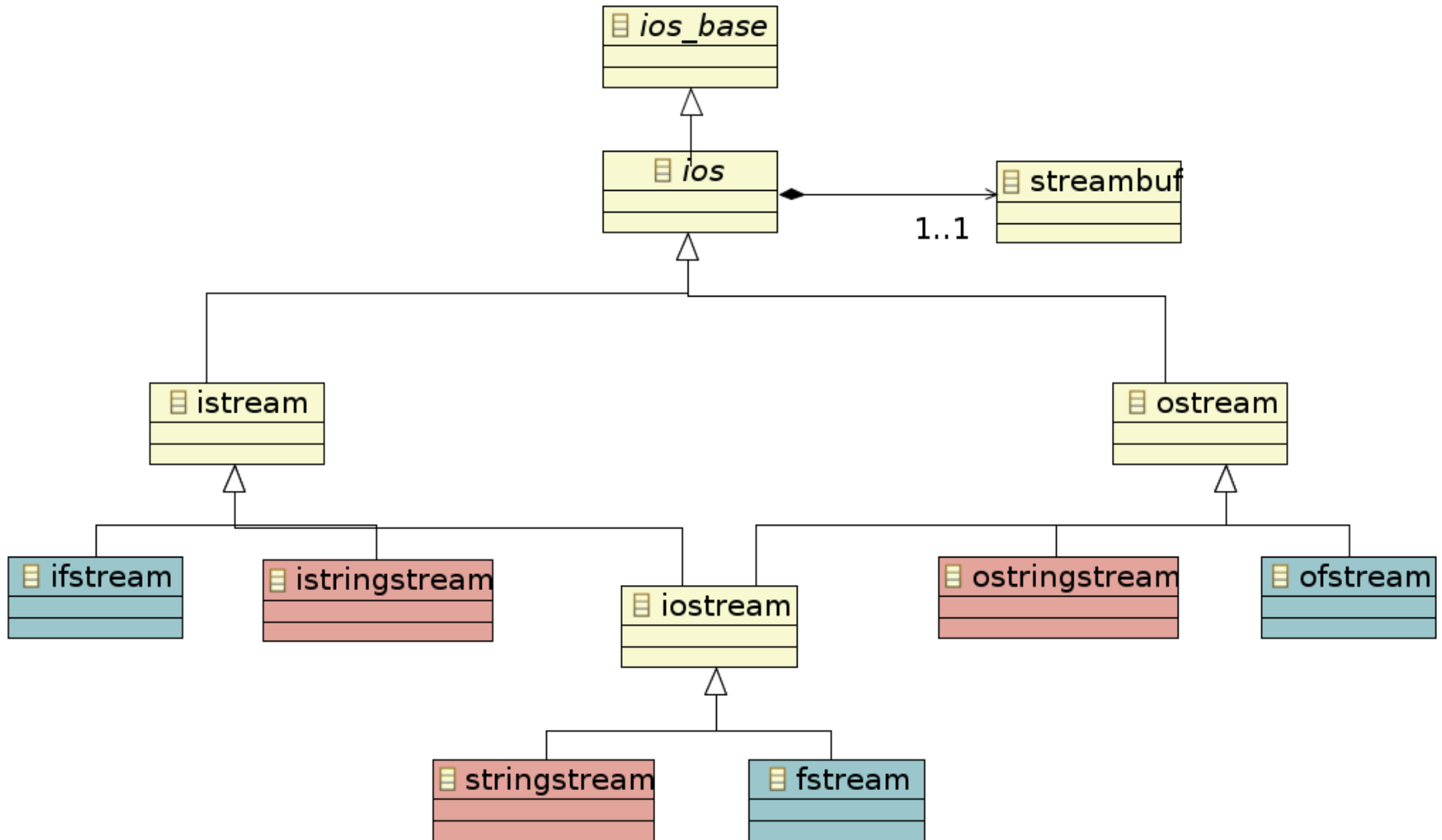
- Introduction
- Contenu de la STL
 - Espace de nommage
 - Exception
 - Conteneurs
 - Itérateurs
 - Algorithmes
 - **Les entrées / sorties**

Flût d'entrées / sorties

- Interaction avec utilisateur aisée
 - Remplace avantageusement printf / scanf
- Basée sur une approche orientée objet
 - Extensible
 - Générique
 - Basé sur les types
- Trois types principaux
 - Three gobal iostreams
 - **cin**: standard input
 - **cout**, **cerr**: standard output, standard error

Flût d'entrées / sorties

Classification



Flût d'entrées / sorties

exemples

- Entrées / sortie de caractères

```
int i;
while ((i = cin.get()) != EOF) {
    cout.put(i);
}

/*****/
char c;
while (cin.get(c)){
    cout << c;
}

/*****/
int j;
while (! cin.eof()){
    cin >> j;
    cout << j;
}
```


Flût d'entrées / sorties

shift operators

- Défini pour les types de base
 - Int
 - String
 - Pointeurs
 - ...
- Peut être étendu à n'importe quel type
 - surcharge d'opérateurs...

Flût d'entrées / sorties

shift operators, exemple

- Opérations pratiques

using namespace std ;

```
int x, y, z, t;  
cout << x << flush; // flush the stream buffer  
cout << x << endl;   // new line and flush  
cout << boolalpha << (x < 5) << endl;  
  
cout << hex << x; // print x in hexadecimal  
  
cout << oct << x << hex << y << dec << z;  
cout << t; // x printed in octal, y in hexadecimal,  
           // z in decimal, t in decimal too (last  
           // state of the stream)
```

Flût d'entrées / sorties

shift operators, exemple 2

- Manipulation de fichiers

```
int x, y;  
ofstream os("fichier.txt"); //open file for writing  
  
cin >> x >> y ;  
os << "from_user: " << x << " " << y << endl;  
    // write into the file  
  
string s;  
int i,j;  
ifstream is("fichier.txt"); //open file for reading  
  
is >> s >> i >> j ;    // read from the file  
cout << s << i << j << endl;
```

Flôts d'entrées / sorties

itérateurs, algorithmes, ...

- Les flôts sont des conteneurs, donc itérateurs et algorithmes peuvent être utilisés...

```
vector<string> v;  
copy(istream_iterator<string>(cin),  
      istream_iterator<string>(),  
      back_inserter(v));  
  
copy(v.begin(), v.end(),  
      ostream_iterator<string>(cout, "\n"));
```

`std::tuple` is a fixed-size collection of heterogeneous values