11 – 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....

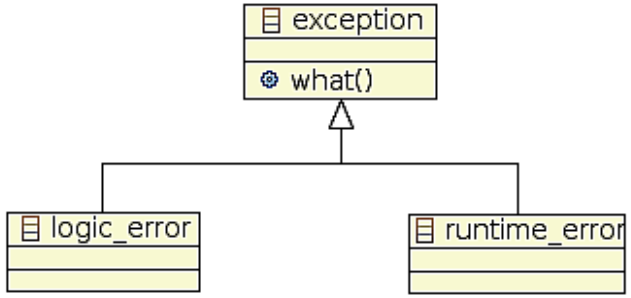
Plan

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

Exceptions de la STL

principes

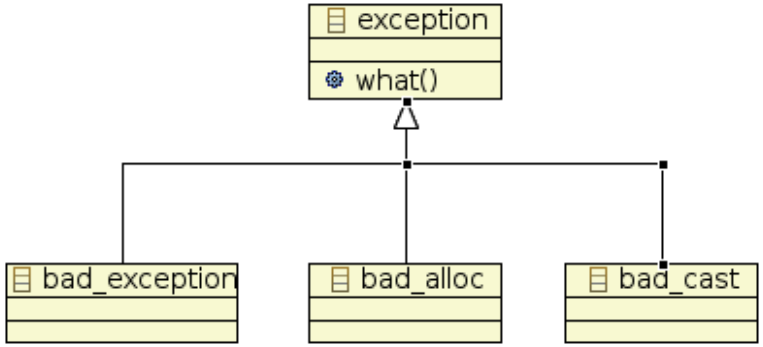
- **exception**
  - #include <exception>
  - #include <stdexcept>
  - Fonction What() doit être redéfinie (message intelligible)



- **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

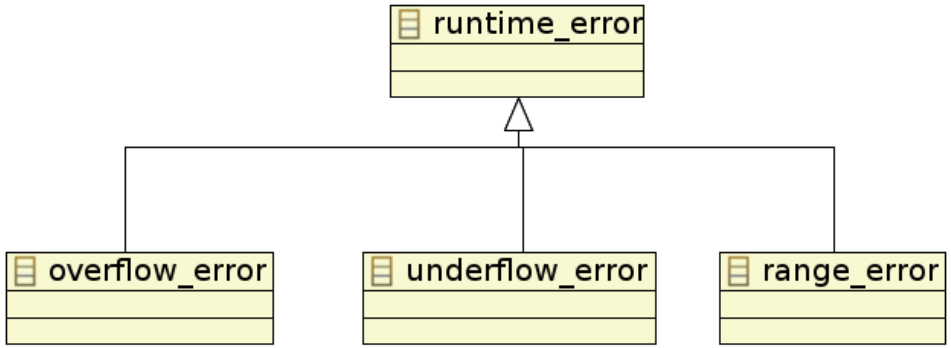
principes



- **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

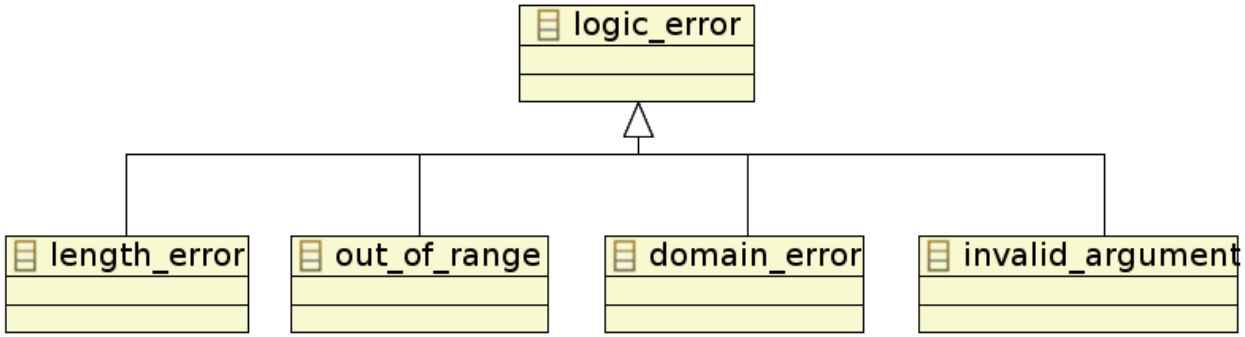
principes



- **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 Rational
{
  int num;
  int den;



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

     den=d;
   }




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

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

};
```

Exceptions de la STL

mise en œuvre

Lancer une exception

Throw list : permet de spécifier les exceptions possiblement lancées par la fonction
Déprécié depuis C++11. À éviter

```
class Rational
{
  int num;
  int den;


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

     den=d;
   }



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

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

};
```

Exceptions de la STL

mise en œuvre

Lancer une exception

```
class Rational
{
  int num;
  int den;



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

     den=d;
   }



   double asDouble () throw (std::range_error)
   {
      if (den == 0)
         throw std::range_error("Rational::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      au 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

Exceptions de la STL

mise en œuvre

```
class Sample1 {
  public:
    Sample1() {
      cout <<"Construct an instance of Sample1"<< endl;
    }
    ~Sample1() {
      cout <<"Destruct an instance of Sample1" << endl;
    }
};
```

```
class MyException: public std::exception{
public:
   const char * what() const noexcept override{
      return "this is a user defined exception";
   }
   MyException(){ cout << "Construct MyException" << endl;  }
   MyException(const MyException& me){cout<<"Copy MyException"<<endl; }
   ~MyException(){ cout << "destroying exception" << endl;  }
};
```

```
int main() {
   try {
      Sample1 s1;
      throw MyException();
   } catch(exception e) {
      cout << "Caught " << e.what() << endl;
   }
   cout << "before the end"<< endl;
}
```

```
Construct an instance of Sample1
Construct MyException
```

Exceptions de la STL

mise en œuvre

```
class Sample1 {
  public:
    Sample1() {
      cout <<"Construct an instance of Sample1"<< endl;
    }
    ~Sample1() {
      cout <<"Destruct an instance of Sample1" << endl;
    }
};
```

```
class MyException: public std::exception{
public:
   const char * what() const noexcept override{
      return "this is a user defined exception";
   }
   MyException(){ cout << "Construct MyException" << endl;  }
   MyException(const MyException& me){cout<<"Copy MyException"<<endl; }
   ~MyException(){ cout << "destroying exception" << endl;  }
};
```

```
int main() {
   try {
      Sample1 s1;
      throw MyException();
   } catch(exception e) {
      cout << "Caught " << e.what() << endl;
   }
   cout << "before the end"<< endl;
}
```

```
Construct an instance of Sample1
Construct MyException
Destruct an instance of Sample1
```

Exceptions de la STL

mise en œuvre

```
class Sample1 {
  public:
    Sample1() {
      cout <<"Construct an instance of Sample1"<< endl;
    }
    ~Sample1() {
      cout <<"Destruct an instance of Sample1" << endl;
    }
};
```

```
class MyException: public std::exception{
public:
   const char * what() const noexcept override{
      return "this is a user defined exception";
   }
   MyException(){ cout << "Construct MyException" << endl;  }
   MyException(const MyException& me){cout<<"Copy MyException"<<endl; }
   ~MyException(){ cout << "destroying exception" << endl;  }
};
```

```
int main() {
   try {
      Sample1 s1;
      throw MyException();
   } catch(exception e) {
      cout << "Caught " << e.what() << endl;
   }
   cout << "before the end"<< endl;
}
```

```
Construct an instance of Sample1
Construct MyException
Destruct an instance of Sample1
Caught std::exception
```

Exceptions de la STL

mise en œuvre

```
class Sample1 {
  public:
    Sample1() {
      cout <<"Construct an instance of Sample1"<< endl;
    }
    ~Sample1() {
      cout <<"Destruct an instance of Sample1" << endl;
    }
};
```

```
class MyException: public std::exception{
public:
   const char * what() const noexcept override{
      return "this is a user defined exception";
   }
   MyException(){ cout << "Construct MyException" << endl;  }
   MyException(const MyException& me){cout<<"Copy MyException"<<endl; }
   ~MyException(){ cout << "destroying exception" << endl;  }
};
```

```
int main() {
   try {
      Sample1 s1;
      throw MyException();
   } catch(exception e) {
      cout << "Caught " << e.what() << endl;
   }
   cout << "before the end"<< endl;
}
```

```
Construct an instance of Sample1
Construct MyException
Destruct an instance of Sample1
Caught std::exception
destroying exception
before the end
```

Exceptions de la STL

mise en œuvre

```
class Sample1 {
  public:
    Sample1() {
      cout <<"Construct an instance of Sample1"<< endl;
    }
    ~Sample1() {
      cout <<"Destruct an instance of Sample1" << endl;
    }
};
```

```
class MyException: public std::exception{
public:
   const char * what() const noexcept override{
      return "this is a user defined exception";
   }
   MyException(){ cout << "Construct MyException" << endl;  }
   MyException(const MyException& me){cout<<"Copy MyException"<<endl; }
   ~MyException(){ cout << "destroying exception" << endl;  }
};
```

```
int main() {
   try {
      Sample1 s1;
      throw MyException();
   } catch(MyException e) {
      cout << "Caught " << e.what() << endl;
   }
   cout << "before the end"<< endl;
}
```

Exceptions de la STL

mise en œuvre

```
class Sample1 {
  public:
    Sample1() {
      cout <<"Construct an instance of Sample1"<< endl;
    }
    ~Sample1() {
      cout <<"Destruct an instance of Sample1" << endl;
    }
};
```

```
class MyException: public std::exception{
public:
   const char * what() const noexcept override{
      return "this is a user defined exception";
   }
   MyException(){ cout << "Construct MyException" << endl;  }
   MyException(const MyException& me){cout<<"Copy MyException"<<endl; }
   ~MyException(){ cout << "destroying exception" << endl;  }
};
```

```
int main() {
   try {
      Sample1 s1;
      throw MyException();
   } catch(MyException e) {
      cout << "Caught " << e.what() << endl;
   }
   cout << "before the end"<< endl;
}
```

```
Construct an instance of Sample1
Construct MyException
Destruct an instance of Sample1
Copy MyException
Caught this is a user defined exception
destroying exception
destroying exception
before the end
```

Exceptions de la STL

mise en œuvre

```
class Sample1 {
  public:
    Sample1() {
      cout <<"Construct an instance of Sample1"<< endl;
    }
    ~Sample1() {
      cout <<"Destruct an instance of Sample1" << endl;
    }
};
```

```
class MyException: public std::exception{
public:
   const char * what() const noexcept override{
      return "this is a user defined exception";
   }
   MyException(){ cout << "Construct MyException" << endl;  }
   MyException(const MyException& me){cout<<"Copy MyException"<<endl; }
   ~MyException(){ cout << "destroying exception" << endl;  }
};
```

```
int main() {
   try {
      Sample1 s1;
      throw MyException();
   } catch(Exception& e) {
      cout << "Caught " << e.what() << endl;
   }
   cout << "before the end"<< endl;
}
```

```
Construct an instance of Sample1
Construct MyException
Destruct an instance of Sample1
Caught this is a user defined exception
destroying exception
before the end
```

Exceptions de la STL

mise en œuvre

```
class Sample1 {
  public:
    Sample1() {
      cout <<"Construct an instance of Sample1"<< endl;
    }
    ~Sample1() {
      cout <<"Destruct an instance of Sample1" << endl;
    }
};
```

```
class MyException: public std::exception{
public:
   const char * what() const noexcept override{
      return "this is a user defined exception";
   }
   MyException(){ cout << "Construct MyException" << endl;  }
   MyException(const MyException& me){cout<<"Copy MyException"<<endl; }
   ~MyException(){ cout << "destroying exception" << endl;  }
};
```

```
int main() {
   try {
      Sample1 s1;
      throw new MyException();
   } catch(Exception* e) {
      cout << "Caught " << e->what() << endl;
   }
   cout << "before the end"<< endl;
}
```

Fuite mémoire !
Inutile !

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 sont type
- **Algorithmes**
  - Opération générique sur l'ensemble des éléments d'un conteneur

Conteneurs de la STL

Les différents types

- **Séquences élémentaires**
  - **Vector, lists, deque (double ended queue), array, forward list**
- **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)		
`vector<T>`	`<vector>`	Tableau avec (re)allocation automatique
`deque<T>`	`<deque>`	Tableau avec indexation optimisée (début et fin)
`list<T>`	`<list>`	Doubly linked sequence (list)
Spécialisations de séquences élémentaires par défaut spécialise **deque** (sauf `priority_queue,` qui utilise **vector**)		
`stack<T>`	`<stack>`	Last In First Out (LIFO)
`queue<T>`	`<queue>`	First In First Out (FIFO)
`priority_queue<T>`	`<queue>`	Queue à priorité (`operator<()`)
Conteneurs associatifs (basé sur une paire (clef, valeur)		
`map<K, T>`	`<map>`	Tableau associatif sans duplication (**K** est la clef)
`multimap<K, T>`	`<map>`	Tableau associatif avec duplication (**K** est la clef)
`set<K>`	`<set>`	Set (sans duplication) of **K**
`multiset<K>`	`<set>`	Set (avec duplication) of **K** (a.k.a. *Bag*)

Conteneurs de la STL

Sequence containers implement data structures which can be accessed sequentially.

array (C++11)	static contiguous array (class template)
vector	dynamic contiguous array (class template)
deque	double-ended queue (class template)
forward_list (C++11)	singly-linked list (class template)
list	doubly-linked list (class template)

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

set	collection of unique keys, sorted by keys (class template)
map	collection of key-value pairs, sorted by keys, keys are unique (class template)
multiset	collection of keys, sorted by keys (class template)
multimap	collection of key-value pairs, sorted by keys (class template)

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$ amortized, $O(n)$ worst-case complexity).

unordered_set (C++11)	collection of unique keys, hashed by keys (class template)
unordered_map (C++11)	collection of key-value pairs, hashed by keys, keys are unique (class template)
unordered_multiset (C++11)	collection of keys, hashed by keys (class template)
unordered_multimap (C++11)	collection of key-value pairs, hashed by keys (class template)

Container adaptors provide a different interface for sequential containers.

stack	adapts a container to provide stack (LIFO data structure) (class template)
queue	adapts a container to provide queue (FIFO data structure) (class template)
priority_queue	adapts a container to provide priority queue (class template)

Conteneurs de la STL
compléxité des functions 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	
`Cont c;`	Crée un conteneur vide
`Cont c1(c2);`	Crée un conteneur par copie (les éléments de **c1** seront une copie de ceux de **c2**)
`Cont c(beg,end);` itérateurs	Crée un conteneur contenant les valeurs situées dans [**beg**, **end**[; rq: les types des valeurs doivent être identiques
`Cont c(n)`	Crée un conteneur de taille **n**
`Cont 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	
`c.size()`	Nombre d'éléments dans le conteneur
`c.empty()`	Vrai si le conteneur ne contient aucun élément
`c.max_size()`	Nombre maximum d'élément défini par l'implémentation du conteneur
`c.resize(n)`	Change la taille à **n**; si cela augmente la taille, les nouveaux éléments sont initialisé à leur valeur par défaut (zéro du type)
`c.resize(n,e)`	Change la taille à **n**; si cela augmente la taille, les nouveaux éléments sont initialisé à **e**
`c.capacity()`	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

```
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
```

```cpp
#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;
```

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** (==, !=, >, <, 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 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 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 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 !!

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 !
  - Opérateur**s** 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 ≡ !(a < b) && !(b < a)

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 ?*

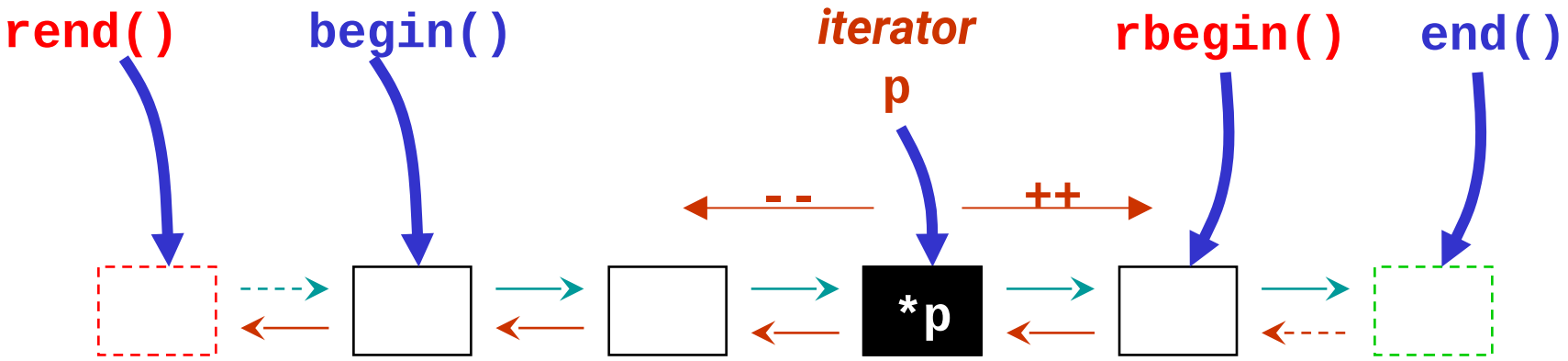
```
#include <functional>
#include <vector>
int main()
{
  std::vector<std::reference_wrapper<int>> iv;
  int a = 12;
  iv.push_back(a);  // or std::ref(a)
  // now iv = { 12 };
  a = 13;
  // now iv = { 13 };
}
```

Plan

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

principes

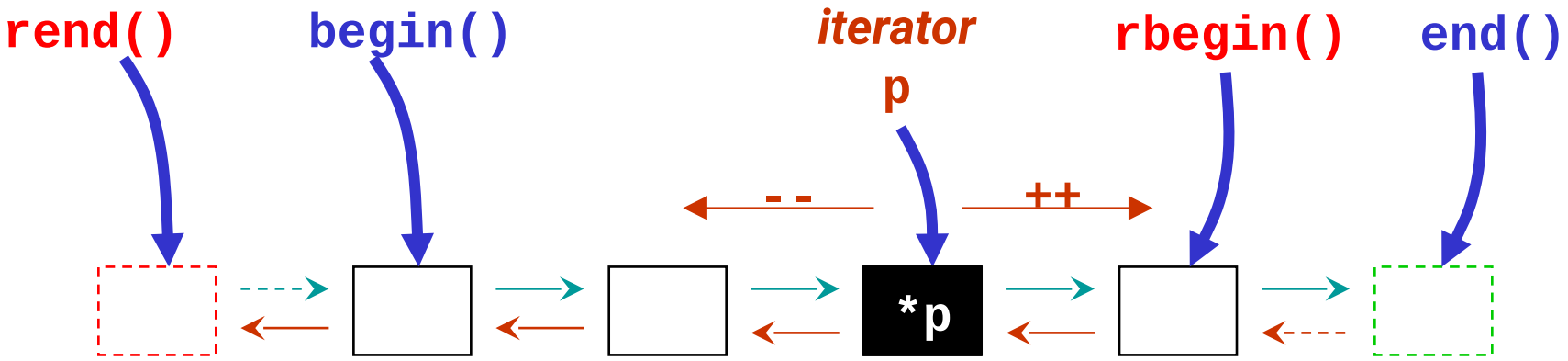
- Peuvent être vu comme un pointeur intelligent sur une liste doublement chainée



Itérateurs de la STL

principes

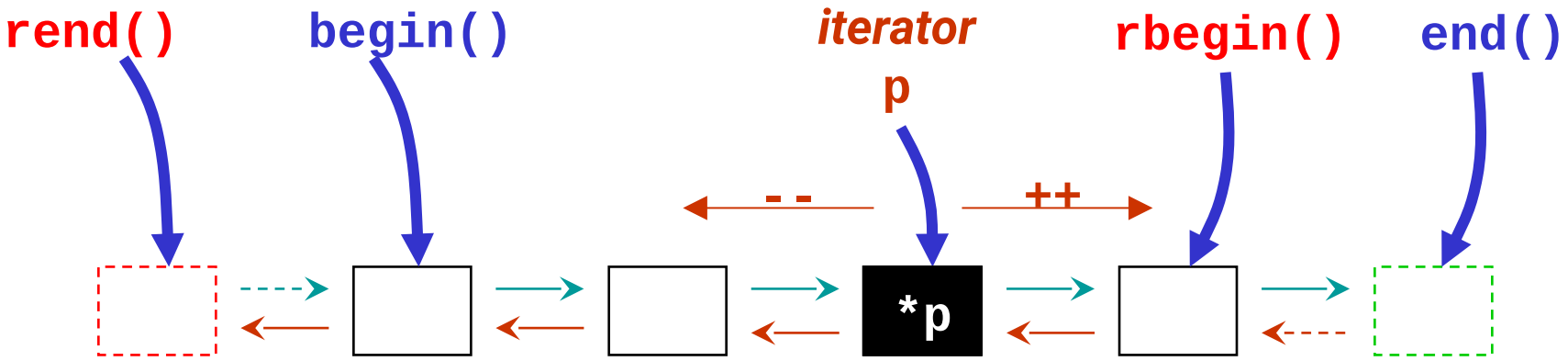
- Peuvent être vu comme un pointeur intelligent sur une liste doublement chainée




```
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 << ' ';
```

principes

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

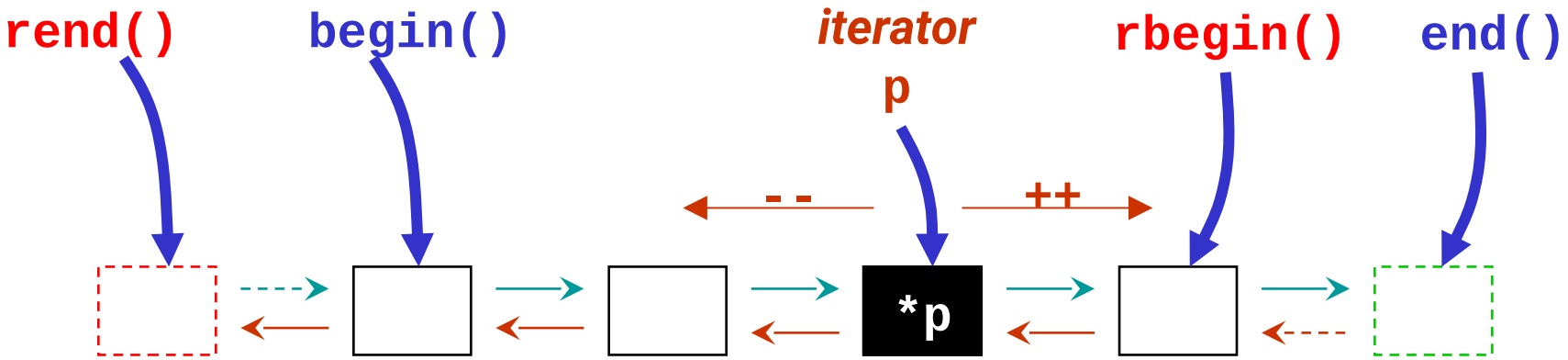



```
list<int> L;
for (int i = 0; i < N; i++) L.push_front(i);

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 chainée




```
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 << ' ';
```

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 exclue [... [

```
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	
`c.begin()`	Renvoie un itérateur sur le premier élément
`c.end()`	Renvoie un itérateur juste après le dernier élément
`c.rbegin()`	Idem c.begin() mais en ordre inverse
`c.rend()`	Idem c.end() mais en ordre inverse
Exemple de fonction utilisant des itérateurs	
`c.insert(pos,e)`	Insert un élément **e**. La valeur retournée ainsi que l'interprétation de **pos** dépend du type de conteneur
`c.erase(beg,end)`	Efface tous les éléments dans l'interval [**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**

principes

- Les algorithmes sont des *function templates*
- Ils opèrent sur une collection d'objets
- Ils visitents 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);
```

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

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

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()**

de plus prêt

- 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);
```

de plus prêt

- 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());
```

de plus prêt

- Passage d'opérateurs :

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

de plus prêt

- 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 nombreuse surcharge de cet opérateur)

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

de plus prêt

- Réification des opérateurs C++: functors prédéfinis

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

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 :
  - **inserter, back_inserter, front_inserter**

➔ 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_inserter(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());
```

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
             );
```

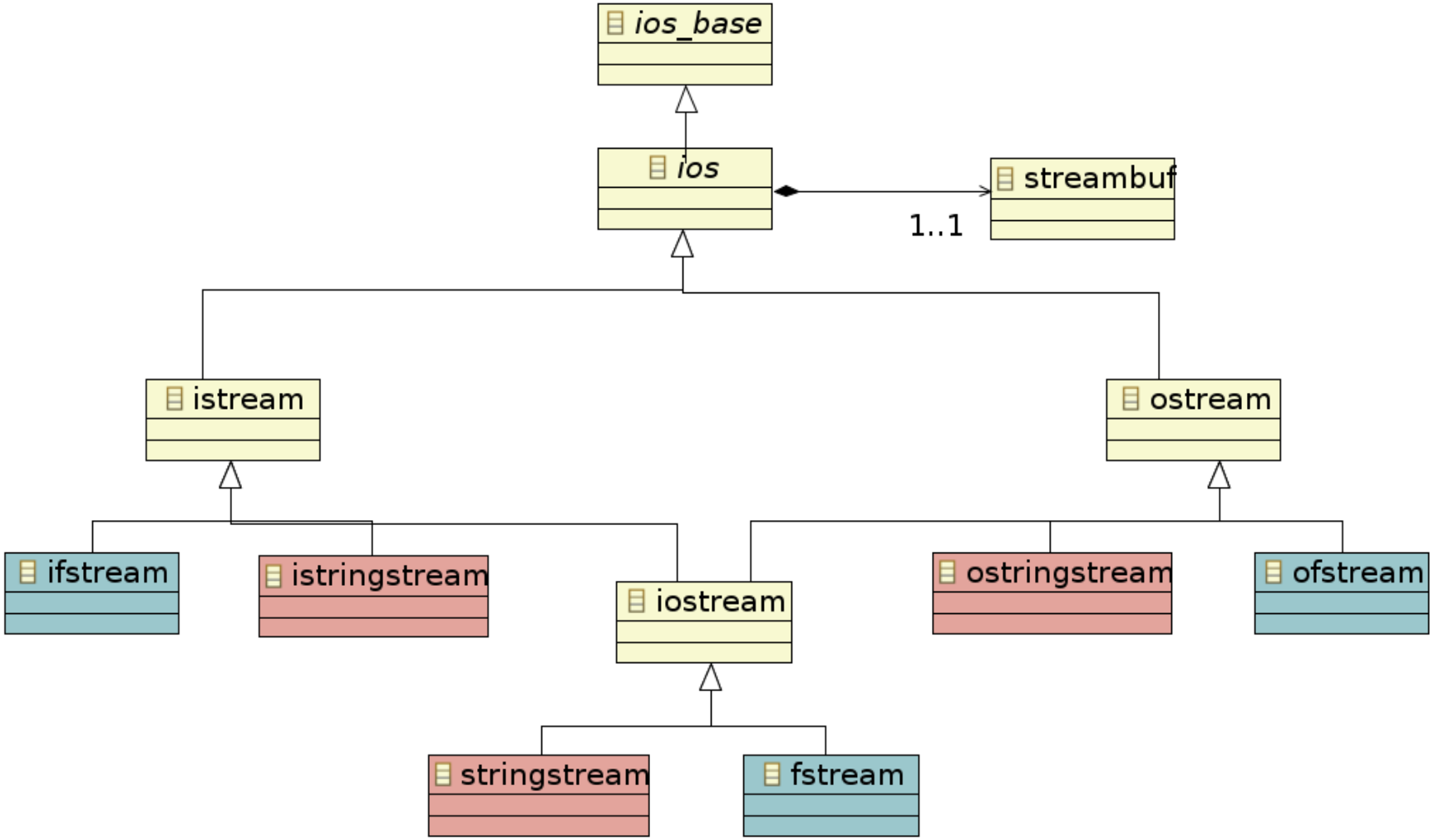
Plan

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

Flôt 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

Classification



Flôt 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;
}
```

Flôt 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...

Flôt 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 << boolapha << (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)
```

Flôt 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::function

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


void goodbye(const std::string& s)
{
   std::cout << "Goodbye " << s << '\n';
}
class Object {
public:
   void hello(const std::string& s)
   {
      std::cout << "Hello " << s << '\n';
   }
};
int main()
{
   using namespace std::placeholders;


   using ExampleFunction = std::function<void(const std::string&)>;
   Object instance;
   std::string str("World");


   ExampleFunction f = std::bind(&Object::hello, &instance, _1);
   f(str);  // equivalent to instance.hello(str)


   f = std::bind(&goodbye, std::placeholders::_1);
   f(str);  // equivalent to goodbye(str)
}
```

std::function and algorithms

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


bool is_gt_v(int i, int v) {
    return i > v;
}

int main()
{
    vector<int> vec = {1,2,3,4,5,6,7};
    int n = count_if(vec.begin(), vec.end(), std::bind(&is_gt_v, std::placeholders::_1, 4));
    cout << n;
}
```