IV Bibliothèque standard

- Ensemble de bibliothèques ISO regroupées dans l'espace de nom std
  - éléments de la bibliothèque standard C
    <cstddef> <climits> <cfloat> <cstdlib> <csignal> <ctime>
    <cstdarg> <cstdio> <cstring> <cctype> <cwchar> <cwtype>
    <cmath> <cassert> <clocale> <csetjmp> <cerrno> <ciso646>
  - des éléments spécifiques C++
    (string, exception, gestion mémoire, identification de types, composants numériques, etc.)
  - IO Stream Library
  - **Standard Template Library (STL)**
    structures de données et algorithmes

Standard Template Library

- Un ensemble de patrons génériques de classes et de fonctions
  - **conteneurs**
    modèles génériques permettant de stocker des collections d'objets (séquences, types abstraits pile,file et file de priorité, tables associatives)
  - **itérateurs**
    abstraction de l'idée de pointeur permettant d'accéder aux éléments d'un conteneur selon une interface unique
  - **algorithmes**
    non écrits pour des conteneurs particuliers puisque n'accédant aux données qu'à travers des itérateurs
- Utilisent les template et non l'héritage et les fonctions virtuelles par soucis d'efficacité

Conteneurs

- Ensemble de modèles génériques représentant les structures de données les plus répandues
  - pour stocker des **séquences** : vector, deque, list
  - pour stocker des collections de **clefs** : set, multiset
    , avec des valeurs associées : map, multimap
- Paramétrés par le type de leurs éléments
- Chacun de ses modèles offre des fonctionnalités spécifiques
  - En particulier, tous les conteneurs offrent les fonctions membres size_t size() et bool empty()

- Exemple :

```
#include <list>
std::list<int> li1, // liste d'entiers (vide)
               li2(9,55); // liste de 9 entiers valant chacun 55
int t[] = {5, 9, 1};
std::list<int> li3(t,t+3);

#include <vector>
std::vector<std::string> vs1,  // vecteur de string (vide)
                         vs2(5,a); // 5 string copies de a
char m[]="millenaire";
std::vector<char> mv(m,m+10);

#include<deque>
std::deque<double> dd1,      //file à double entrée (vide)
                   dd2(6), // deque de 6 doubles
                   dd3(dd1); // par copie
```

- vector (tableau dynamique)
  - accès direct aux éléments
    opérateur **[]**(int), fonction membre **at**(int),
    accès au dernier élément par fonction membre **back**()
  - insertion/suppression à la fin en temps constant
    fonctions membres **push_back**(*elt*), **pop_back**()
  - insertion/suppression en temps linéaire sinon
    fonctions membres **insert**, **erase**
  - nombre dynamique d'éléments avec gestion automatique d'extension mémoire si nécessaire
    fonctions membres **size**(), **resize**(*nvelletaille*, *elt*),
    capacity(), **reserve**(*nvellecap*), **max_size**()

Exemple :

```
std::vector<int> v;
v.push_back(5);
v.push_back(1);
v.insert( v.begin()+1, 9);
v.pop_back();
v.insert( v.begin(), 2,7);
std::cout <<v[1] << " " << v.at(2) << std::endl;
v[0]=v.at(1)=v.back()=1;
v.erase( v.begin()+2 );
v.insert( v.end(), 3);
std::cout << v.size() << std::endl;
v.resize(6,0);
v.resize(2);
```

Exemple :

```
std::vector<int> v;
v.push_back(5);                          // 5
v.push_back(1);                          // 5 1
v.insert( v.begin()+1, 9);               // 5 9 1
v.pop_back();                            // 5 9
v.insert( v.begin(), 2,7);               // 7 7 5 9
std::cout <<v[1] << " " << v.at(2) << std::endl;
v[0]=v.at(1)=v.back()=1;                 // 1 1 5 1
v.erase( v.begin()+2 );                  // 1 1 1
v.insert( v.end(), 3);                   // 1 1 1 3
std::cout << v.size() << std::endl;
v.resize(6,0);                           // 1 1 1 3 0 0
v.resize(2);                             // 1 1
```


- list (liste doublement chainée)
  - insertion/suppression en temps constant
    insert, **erase**, **push_front**(*elt*), **pop_front**(),
    push_back(*elt*), **pop_back**()
  - pas d'accès direct à tous les éléments
    fonctions membres **front**() et **back**()
  - possibilité de parcours séquentiel dans les 2 sens
  - gestion automatique de l'espace utilisé
    fonctions membres **size**(), **max_size**()
  - des fonctions membres spécifiques
    remove, unique, splice, reverse, sort, merge

Exemple :

```
std::list<double> l(4,1.5);
l.push_front(5.9);
l.pop_back();
l.insert( l.begin(), 7.7);
l.reverse();
l.sort();
l.remove(1.5);
l.push_back(7.7);
l.unique();
l.erase(l.begin());
```


Exemple :

```
std::list<double> l(4,1.5);   // 1.5  1.5  1.5  1.5
l.push_front(5.9);            //  5.9  1.5  1.5  1.5  1.5
l.pop_back();                 // 5.9  1.5  1.5  1.5
l.insert( l.begin(), 7.7);    // 7.7  5.9  1.5  1.5  1.5
l.reverse();                  // 1.5  1.5   1.5  5.9  7.7
l.sort();                     // 1.5  1.5  1.5  5.9  7.7
l.remove(1.5);                // 5.9   7.7
l.push_back(7.7);             // 5.9  7.7  7.7
l.unique();                   // 5.9   7.7
l.erase(l.begin());           // 7.7
```

- class deque (file à double entrée)
  - accès direct aux éléments en temps constant *
    opérateur **[ ]**(int), fonctions membres **at**(int), **back**() et **front**()
  - insertion/suppression au début ou à la fin en temps constant*
    fonctions membres **push_back**(*elt*), **pop_back**(),
    push_front(*elt*), **pop_front**()
  - insertion/suppression en temps linéaire*, sinon
    fonctions membres **insert**, **erase**
  - nombre dynamique d'éléments, avec gestion automatique d'extension mémoire si nécessaire
    fonctions membres **size**(), **resize**(*newtaille,elt*), **max_size**()
    Pas de fonctions membres **capacity**(), **reserve** (*taille)*

(*amorti)

Adaptateurs de séquences

- Fournis en association avec les séquences
  - Patrons de classes construits sur des conteneurs
- Définition d'une nouvelle interface pour un conteneur, afin de lui donner le comportement d'un type abstrait
  stack, queue ou priority_queue
- Exemple :

```
#include <stack>
std::stack<int, std::vector<int> > pi;
#include <queue>
std::queue <int, std::deque<int> > fi;
std::priority_queue<int, std::deque<int> > fip;
std::priority_queue<int, std::deque<int>,std::greater<int> >
fip2;
```

Adaptateur stack

template < class T, class Container=deque<T> >
class **stack**;

- Le conteneur utilisé à l'instanciation doit supporter les opérations back, push_back et pop_back

- Principales opérations sur les std::stack
  push(*elt*), **pop**(), **top**(),
  empty(), **size**()

Adaptateur queue

template < class T, class Container=deque<T> >
class **queue**;

- Le conteneur utilisé à l'instanciation doit supporter les opérations front, push_back, pop_front et back

- Principales opérations sur les std::queue
  push(*elt*), **pop**(), **front**(),
  empty(), **size**()

Adaptateur priority_queue

```
template <class T, class Container=vector<T>,
      class Compare=less<typename Container::value_type> >
class priority_queue;
```

Type abstrait queue de priorité offrant l'extraction de l'élément de plus grande priorité (au sens de la classe Compare *, qui teste l'**infériorité**)

- Le conteneur utilisé à l'instanciation doit supporter l'indexation ainsi que les opérations front, push_back et pop_back
- Principales opérations sur les priority_queue
  push(*elt*), **pop**(), **top**(), **empty**(), **size**()

(*Compare étant une classe d'objets "fonctions de comparaison")

Exemple :

```
std::priority_queue < int, std::vector<int> > p1;
std::priority_queue < int, std::deque<int>, std::greater<int> > p2;
p1.push(9); p2.push(9);
p1.push(7); p2.push(7);
p1.push(30); p2.push(30);
while(!p1.empty())
   { std::cout << p1.top()  << std::endl;
     p1.pop();
   } // 30  9  7
while(!p2.empty())
   { std::cout << p2.top()  << std::endl;
     p2.pop();
   } // 7  9  30
```

Fabrication d'une classe d'objets "fonctions de comparaison"

- Classe disposant d'une surcharge de l'opérateur **()** correspondant à une fonction de comparaison : fonction de 2 arguments renvoyant 1 booléen
- La bibliothèque standard a prévu un modèle de classe **binary_function*** pour "formaliser" cela ...

```
class MaCompare : public std::binary_function<int, int, bool>
{ bool operator ( ) (int i , int j)
   { … return ...}
}
int main()
{ std::cout << MaCompare( )(3,5) << std::endl; //Comparaison de 3 et 5
 std::priority_queue<int, std::vector<int>, MaCompare> p;
  ...
}
```

Construction d'un objet permettant de comparer 2 entiers

(* Dans <functional>, où on trouve également le modèle **unary_function**)

- Toute fonction peut être transformée en foncteur.
- Possibilité d'utiliser les algorithmes de la bibliothèque standard avec des fonctions classiques (moyennant cette petite transformation).
- Adaptateur fournit par la STL

```
template <class Arg1, Arg2, Result>
class pointer_to_binary_function :
public binary_function<Arg1, Arg2, Result>
{ public:
    explicit pointer_to_binary_function(Result (*fonction)(Arg1, Arg2));
    Result operator()(Arg1 argument1, Arg2 argument2) const;
};

template <class Arg, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
ptr_fun(Result (*fonction)(Arg1, Arg2));
```

```
#include <functional>
template <class T, class F>
T applique_foncteur(T i, T j, F foncteur)
    { return foncteur(i, j); }
    Le dernier argument de applique_foncteur doit être un foncteur

int f(int i, int j)
    { return i * j; }
int main(void)
 { // Appel adaptateur pour transformer pointeur de fonction
   // en foncteur :
     int res;
     res=applique_foncteur(2, 3, std::ptr_fun(&f));
    // ou bien :  res=applique_foncteur(2, 3,
    //                    std::pointer_to_binary_function<int,int,int>(&f));
    // ou bien : res=applique_foncteur(2, 3, f); //toléré
    return 0;
}
```


Conteneurs associatifs

Les conteneurs associatifs ont pour vocation de stocker et de retrouver (des informations associées à) des clefs, en exploitant un ordre strict faible sur les clefs

- **set et multiset (*):**
  - collection de clefs (sans association d'information)
  - les set ne peuvent contenir 2 clefs équivalentes (les multiset oui)

```
#include <set>
std::set<int> si;
std::multiset<double> md;
```

- **map et multimap (*):**
  - collection de valeurs associées à des clefs
  - les maps ne peuvent contenir 2 clefs équivalentes (les multimaps oui)

```
#include <map>
std::multimap<string, int> annuaire; //cles string, valeurs associees int
```

*« sorted » mais il existe aussi des « Hashed » conteneurs associatifs (hash_set, hash_multiset, hash_map, hash_multimap)

set et multiset

template <class Key, class Compare = less<Key> >
class **set**; // resp class multiset

- Principales opérations
  insert(*key*), **erase**(*key*), find(*key*), **size**(), **empty**()
- Les set et multiset ne correspondent pas à des séquences
  (les opérations [], at, push_front, push_back, pop_front, pop_back ne sont donc pas définies)
- Un multiset conserve les éléments équivalents
- exemple

```
std::set<int> s; std::multiset<int> ms;
for(int i=0 ; i< 10 ; i++)
      { s.insert(i%2); ms.insert(i%2);}
```

  Que contiennent les ensemble s et ms ?

  s contient 2 éléments : 0 et 1; ms contient 10 éléments : 0 0 0 0 0 et 1 1 1 1 1

map

template <class Key, class T,
Compare = less<Key> >
class **map**;

- Une table associative est une collections de couples (clef, valeur) offrant **l'accès à un couple à partir de sa clef**
- Principale opération :
  accès "direct" à un élément par **operator[ ](*key*)**
- Les couples (clef, valeur) sont insérées sous forme de std::pair<key,T>
- 2 éléments d'une map ne peuvent avoir des clefs équivalentes

- Exemple :

```
template<class K, class V>
std::ostream & operator << (std::ostream & os, std::pair<K,V> & e)
   {os << " (" << e.first << ',' << e.second << ") ";
    return os;
   }
int main()
   { std::map <std::string, int> tel;
     tel["Fanny"]=06555555;
     tel["Bozo"]=01444444;
     tel["Jeez"]=04777777;
     tel["Bozo"]=02345678;
     std::cout << tel.size()  // 3
               << tel["Jeez"] << tel["Fanny"]
               << tel["Bozo"] << std::endl; // 02345678
     return 0;
   }
```


multimap

- Les multimap peuvent conserver plusieurs éléments dont les clefs sont équivalentes
- Pas d'accès direct avec l'opérateur []
- Insertion/Suppression des couples (clef,valeur) avec les fonctions membre **insert**(*key*) et **erase**(*key*)

```
std::multimap<std::string,int> telm;
telm.insert(std::make_pair(std::string("Tonio"),06777777));
telm.insert(std::make_pair(std::string("Lili"),03222222));
telm.insert(std::make_pair(std::string("Tonio"),08111111));
std::cout << telm.size(); // 3
```

- Accès aux couples (clef,valeur) à l'aide d'itérateurs et des fonctions membres **find**(*key*) **lower_bound**(*key*) et **upper_bound**(*key*)

Remarque générale

– Les conteneurs possèdent leurs éléments
  - insertion d'un élément = introduction d'une copie*
  - la destruction d'un conteneur s'accompagne de celle de ses éléments
  - la copie d'un conteneur entraîne celle de tous ses éléments

(*Mais on peut toujours faire des conteneurs de pointeurs!)

Les itérateurs

- Concept d'*iterator*
  – désigne toute classe munie des opérateurs membres * et ->, et du test d'égalité
    - la valeur ou la référence retournée par l'opérateur * est dite "élément pointé"

  – abstraction de la notion de pointeur
    - un pointeur est un cas particulier d'itérateur

- Concept d'*input iterator*
  - iterator muni de l'opérateur **++**
  - accès en **lecture** à l'élément pointé (par retour de operator *())
- Concept d'*output iterator*
  - iterator muni de l'opérateur **++**
  - accès en **écriture** à l'élément pointé (retour d'une référence par operator *())
- Concept de *forward iterator*
  - désigne toute classe qui est à la fois un *input* et un *output iterator*

- concept de *bidirectional iterator*
  - forward iterator muni de l'opérateur **- -**
- concept de *random access iterator*
  - forward iterator muni
    - d'une **arithmétique** similaire à celle des pointeurs (addition ou soustraction d'un entier, soustraction entre 2 itérateurs),
    - de l'opérateur **[ ]**,
    - et supportant des **opérations de comparaison**

Opérateurs de déférencement

- Attention :
  L'opérateur surchargé (* ou ->) s'applique à un objet et non à un pointeur

- Surcharge de l'opérateur unaire *
  - Utile pour définir des pointeurs généralisés (cf Itérateur)
  - La définition d'une conversion vers un pointeur peut parfois éviter cette surcharge ainsi que celle de [ ]


```
class Tableau
{
      …
  operator int*();
};
Tableau tab;
```

Conversion utilisée dans *tab ou tab[i]

- Surcharge de l'opérateur unaire **->**

  Tableau tab;
  tab->membre s'interprète comme

  (tab.operator ->()) **->** membre;

  – Doit renvoyer :
    - un pointeur sur une classe possédant le nom de membre attendu
    - ou un objet d'une classe disposant de -> tel que ...

Surcharge de l'opérateur ++ (préfixé)

- Possibilité de donner un sens à ++z, avec z de type non primitif (ex : Complexe)
  – **soit en surchargeant ++ en opérateur membre sans argument**

  Complexe & Complexe::**operator ++()**;

  (++z interprété comme z.operator++())

– **soit en surchargeant ++ en fonction (amie) ayant pour unique argument un Complexe**

friend Complexe & **operator ++** (Complexe &);

(++z interprété comme operator++(z))

Surcharge de l'opérateur ++ (postfixé)

- Possibilité de donner un sens à z++, avec z de type non primitif (ex : Complexe)

– **soit en surchargeant ++ en opérateur membre ayant un argument entier**

Complexe operator ++**(int)**;
(z++ interprété comme z.operator++(0))

– **soit en surchargeant ++ en fonction amie ayant pour arguments un Complexe et un entier**

friend Complexe operator ++(Complexe &, **int**);

(z++ interprété comme operator++(z,0))

– **l'argument entier est juste un artifice**

- Chaque conteneur de la STL fournit :
  - un type local d'itérateur permettant d'accéder à ses éléments, et d'en faire un parcours exhaustif
    ex : std::list<int> ll;
    std::list<int>**::iterator** it;
  - une fonction membre **begin()** renvoyant un itérateur du type correspondant au conteneur
    - si le conteneur est un conteneur séquentiel, begin() pointe sur le premier élément de la séquence
  - une fonction membre **end()** renvoyant un itérateur du type correspondant au conteneur
    - si le conteneur est séquentiel, end() pointe **après** le dernier élément de la séquence

- Exemple :
  Parcours exhaustif et séquentiel d'une liste

```
std::list<int> l;
l.push_back(3);
l.push_front(5);
l.push_back(2);
for (std::list<int>::iterator it=l.begin() ; it!=l.end() ; it++)
   {std::cout << *it << std::endl; }
```


- Exemple :
  Parcours d'une multimap

```
std::multimap<std::string,int> telm;
telm.insert(std::make_pair(std::string("Tonio"),06777777));
telm.insert(std::make_pair(std::string("Lili"),03222222));
telm.insert(std::make_pair(std::string("Tonio"),08111111));
std::multimap<std::string,int>::iterator it;
for (it=telm.begin() ;  it!=telm.end() ; it++)
      {std::cout << (*it).first < std::endl; }
```

Parcours des éléments de clef donnée

```
for( it=telm.lower_bound("Tonio");
      it!=telm.upper_bound("Tonio");
      it ++)
      {std::cout << (*it).second << std::endl; }
```

Itérateurs d'insertion

- Itérateurs associés aux conteneurs de la STL :
  - accès aux éléments existants dans le conteneur
  - modification de ses éléments
  - mais pas d'ajout d'éléments …

- Itérateur d'insertion :
  output iterator **permettant d'étendre le contenu d'un conteneur**, par insertion de nouveaux éléments à l'endroit pointé

```
std::list<int> l;
std::back_insert_iterator<std::list<int> > it=std::back_inserter(l);
for(int i=0;i<10;i++)
  *it++= i; //equivaut l.pushback(i);
```

- Il existe également des itérateurs d'insertion au début

- Attention :
  - Certaines opérations sur un conteneur peuvent invalider des itérateurs pointant sur des éléments de ce conteneur

    (ex : extension mémoire suite à l'insertion d'un nouvel élément dans un vector ou deque : certains des éléments pointés ont pu "déménager")

Itérateurs de flux

- Possibilité d'associer un itérateur à un flux de sortie (ostream), de manière à pouvoir écrire dessus

```
std::ostream_iterator<int> os(std::cout," puis ");
//pour écrire des entiers séparés par " puis"
*os=8;
++os;
*os=9;
++os;
```

Affichage sur la sortie standard :
8 puis 9 puis

- Possibilité d'associer un itérateur à un flux d'entrée (istream) de manière à pouvoir y lire des données

```
std::istream_iterator<int> is(std::cin);
//pour lire des entiers sur l'entrée standard
int a,b;
a=*is;
++is;
b=*is;
++is;
```

Algorithmes

- Algorithmes applicables à différents conteneurs
- Comment?
  Accès aux éléments du conteneur, uniquement à travers des itérateurs
- De nombreux algorithmes (site web SGI) :
  - de copie (***copy***),
  - de recherche d'un élément (***find, min_element, max_element, search***),
  - de tri (***sort***)
  - d'opérations sur les tas
    (***make_heap, push_heap, pop_heap, sort_heap***),
  - d'application d'une fonction aux éléments d'un conteneur (***for_each***),
  - d'opérations élémentaire sur des ensembles (***set_union, includes***,…)
  - de mélange (**random_shuffle**)
  - ...

Exemple d'utilisation de copy

```
std::list<int> l(5,1);
std::vector<int> v(5);

// Copie des éléments de l dans v (qui en a la place!)
std::copy( l.begin() , l.end() , v.begin() );

// Affichage des éléments de v
std::ostream_iterator<int> os(std::cout," ");
std::copy( v.begin(), v.end(), os);

// Extension de v par copie de l
std::copy( l.begin(), l.end(), std::back_inserter(v));
```

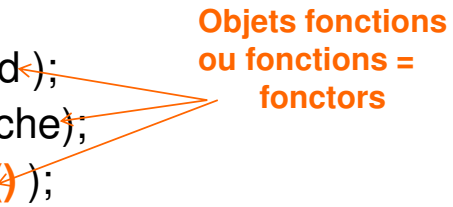
Exemple d'utilisation de find

```
std::list<std::string> l;
l.push_front("Lou");
l.push_front("Serge");
std::list<string>::iterator s;
s=std::find(l.begin(), l.end(), std::string("Anna"));
if (s != l.end())
   { …// la string "Anna" est dans la liste à la position s;}
```


Exemple d'utilisation de generate et de for_each

```
void affiche(int i){std::cout << i << std::endl;}
class Aff //classe d'objets fonctions
{public :
   void operator () (int i)
        {std::cout << "int : " << i << std::endl;}
};

std::vector<int> v(10);
std::generate( v.begin(), v.end(), rand );
std::for_each( v.begin(), v.end(), affiche);
std::for_each( v.begin(), v.end(), Aff() );
```



Exemple d'utilisation de sort

Tri par ordre croissant

```
std::vector<int> v(10);
std::generate( v.begin(), v.end(), rand );
std::sort( v.begin(), v.end(), std::less<int>() ) ;
```


#include<functional>

Exemple d'utilisation de transform

Affectation des éléments de la séquence s2, par application de la fonction sin aux éléments de la séquence s1

```
std::vector<int> s1(10);
std::list<int> s2(10);
std::generate( s1.begin(), s1.end(), rand );
std::transform( s1.begin(), s1.end(), s2.begin(), sin ) ;
```

- Exo :

```
#include <list>
#include <algorithm>
#include <iostream>
int main()
{
  std::list<int> lili(4);
  std::ostream_iterator<int> os(std::cout,"yo ");
  std::list<int>::iterator it=lili.begin();
  std::copy(lili.begin(),lili.end(),os); std::cout<<"\n";
  *it=1;  std::copy(lili.begin(),lili.end(),os); std::cout<<"\n";
  *it++=2;  std::copy(lili.begin(),lili.end(),os); std::cout<<"\n";
  *it++=3;   std::copy(lili.begin(),lili.end(),os); std::cout<<"\n";
  std::insert_iterator<std::list<int> > ins(lili,it);
  *ins=4;  std::copy(lili.begin(),lili.end(),os); std::cout<<"\n";
  *ins=8;  std::copy(lili.begin(),lili.end(),os); std::cout<<"\n";
  *ins++=5;  std::copy(lili.begin(),lili.end(),os); std::cout<<"\n";
   return 0;
}
```



```
0yo 0yo 0yo 0yo
1yo 0yo 0yo 0yo
2yo 0yo 0yo 0yo
2yo 3yo 0yo 0yo
2yo 3yo 4yo 0yo 0yo
2yo 3yo 4yo 8yo 0yo 0yo
2yo 3yo 4yo 8yo 5yo 0yo 0yo
```

Namespace

- Utilisation de différents modules et bibliothèques dans un programme
- Problème dit de « pollution de l'espace de noms » :
  Un même identificateur peut être utilisé par plusieurs modules ou bibliothèques
  – Risque d'ambiguité

- Concept d'espace de noms en C++ :
  donner un nom à un espace de déclaration

```
namespace mon_module
{//déclarations usuelles
  extern double taux;
  double conversion(double);
}
```

- Pour se référer à des identificateurs définis dans un espace de noms, on utilise l'opérateur :: de résolution de portée

```
double mon_module::taux=6.5; //définition
std::cout << mon_module::conversion(1);
```

On dit aussi que l'on se réfère à l'identificateur taux declaré dans la portée de mon_module

- A l'interieur de mon_module, on utilise directement le nom taux

```
double mon_module::conversion(double a)
{
 return a*taux; // mon_module::taux
}
```

- L'espace des déclarations globales d'un programme est aussi un espace de noms dit portée globale

  ::x fait référence à l'identificateur x de la portée globale

- using_declaration :
  permet de faire entrer (connaître) un identificateur dans la portée courante

```
using mon_module::taux;
std::cout << taux;
```

- exemple :
  Si on fait entrer taux dans la portée globale alors ::taux mon_module::taux deviennent des écritures équivalentes

- Attention : il ne doit pas y avoir d'autre taux dans la portée courante

- using_directive
  using namespace mon_module;
  permet de rendre visibles les noms de mon_module

```
using namespace mon_module;
std::cout << conversion(3);
```

- Risques d'ambiguïtés si plusieurs espaces de noms comportant des identificateurs identiques sont rendus visibles

- using namespace mon_module;
  //espace de nom déclarant taux
  using namespace son_module;
  //espace de nom déclarant taux
  std::cout << taux; //appel ambigu

- Faire appel à l'opérateur de résolution de portée

- Danger des using-directives :
  Le compilateur signale
  - les ambiguïtés entre identifiants des espaces de noms rendus visibles,
  - mais pas les surcharges de fonctions à travers les espaces de noms!


```
using namespace mon_module;
 int conversion(int s){return s*taux;}
 int main()
 { int taux=9;
   std::cout<< taux << “ “ <<::taux
            << " " << mon_module::taux
           << " " << conversion(1) << " "
            << conversion(1.5) << std::endl;
    return(0);
 }
 // 9  6.5  6.5  6  9.75
```

- Le fonctionnement d'un programme satisfaisant peut changer après introduction (totalement indépendante) d'une nouvelle fonction dans un des espaces de noms rendu visible
- il est parfois plus prudent d'utiliser une using_declaration qu'une using_directive!

- Les espaces de noms
  - peuvent être emboîtés
  - peuvent être étendus

```
namespace mon_module
{ ...
   namespace sous_module
   { ...
   }
}
namespace mon_module
{ ...
}
```

- smartPointers.ppt