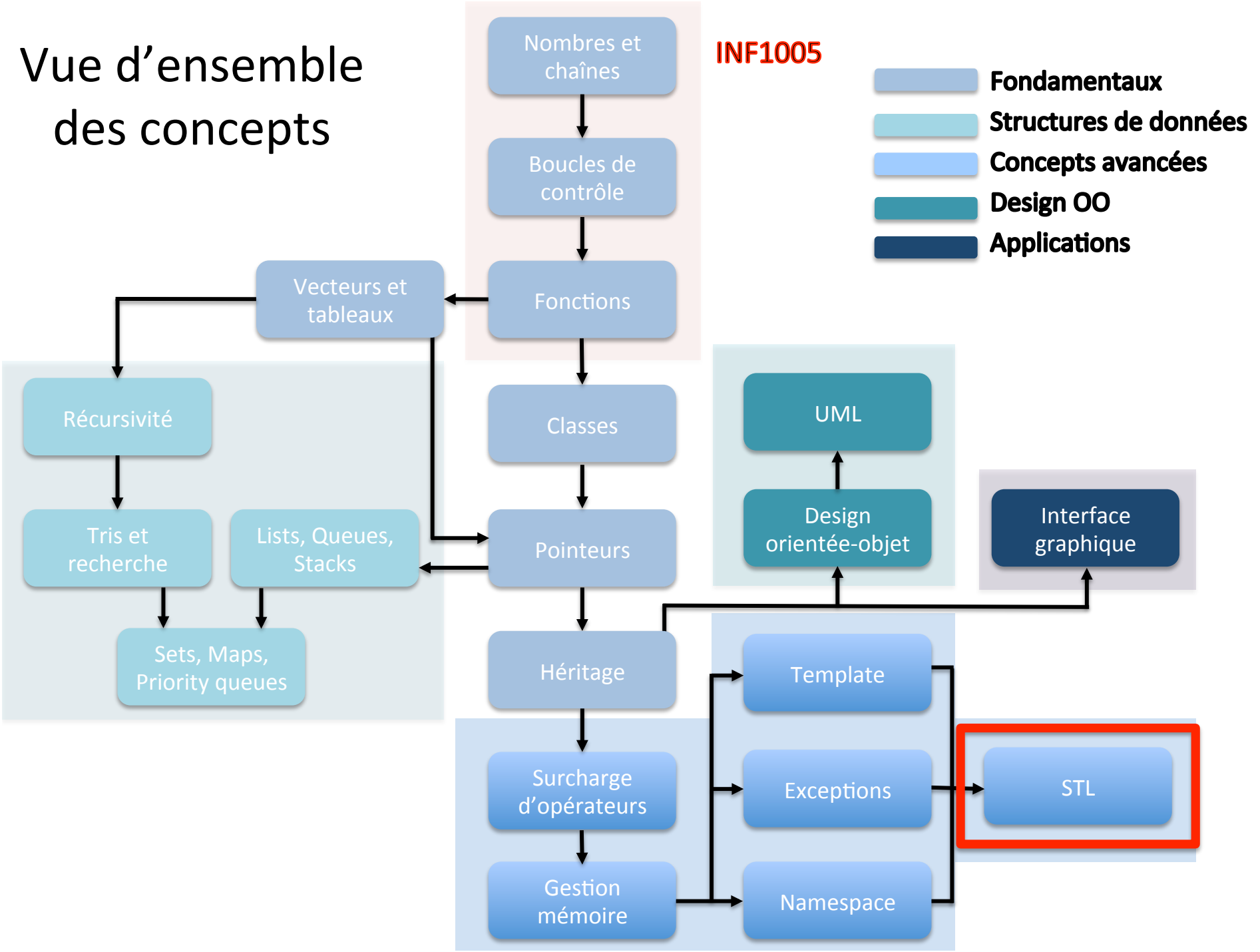
Programmation orientée objet

Foncteurs STL

Vue d'ensemble des concepts



Aperçu

- Introduction
- Foncteurs
- Prédicats
- Catégories :
  - Générateurs
  - Foncteurs unaires
  - Foncteurs binaires
- Foncteurs adaptables
- Adaptateurs de foncteurs
- Conclusion

Introduction

- **Foncteur** : Un foncteur (ou objet fonction) est un objet se comportant comme une fonction. Il repose sur la surcharge de l'opérateur ()
- Les foncteurs sont très utiles avec STL, car ils sont généralement passés aux algorithmes pour manipuler les éléments des conteneurs.
- Il y a deux types de foncteurs :
  - **Foncteurs** : permet seulement l'appel de l'opérateur `()`
  - **Foncteurs adaptables** : surchargent l'opérateur `()`, en plus de définir le type de ses arguments et du résultat.

Foncteurs – exemple 1

```
class PrintInt
{
  public:
    void operator() (int elem);
};


void PrintInt::operator() (int elem)
{
   cout << elem << ' ';
}
```

Foncteurs – exemple 1 (suite)

```
int main()
{
  vector<int> conteneur;
  PrintInt imprimer;
   ...
  size_t max = conteneur.size();
  for_(size_t i = 0; i < max; ++i) {
    imprimer(conteneur[i]);
   }
  cout << endl;

  return 0;
}
```

On déclare le foncteur.

On applique l'opérateur () à l'élément courant du conteneur.

Foncteurs – exemple 2

- Supposons que l'on veuille une fonction qui incrémente un élément d'une certaine valeur.
- Si la valeur est fixe et connue à l'avance, on peut utiliser une fonction normale:

Foncteurs – exemple 2 (suite)

```
void ajouter10(int &elem) // par adresse pour modifier
{
   elem += 10;
}

int main()
{
   vector< int > unTableau;
   ...
   size_t max = unTableau.size();
   for (size_t i = 0; i < max; ++i){
    ajouter10(unTableau[i]);
   }
   ...
}
```

Foncteurs – exemple 2 (suite)

Si la valeur de l'incrément est connue seulement lors de l'exécution, on pourrait utiliser un foncteur

```
class Ajouter
{
public:
  Ajouter(int v) : laValeur(v) {}
  void operator() (int& elem);
private:
   int laValeur;
};
void Ajouter::operator() (int& elem)
{
   elem += laValeur;
}
```

Comme tout objet, un foncteur peut avoir des attributs. Ici, il aura un attribut servant à mémoriser l'incrément qui sera appliqué à la valeur passée en paramètre.

Foncteurs – exemple 2 (suite)

```
int main()
{
   vector< int > unTableau;
   ...
   int increment;
   cin >> increment;
   Ajouter ajouter(increment);
   size_t max = unTableau.size();
   for (size_t i = 0; i < max; ++i){
       ajouter(unTableau[i]);
   }
   ...
   return 0;
}
```

On construit le foncteur.

On l'applique à chaque élément du vecteur.

Catégories

- La STL définie trois catégories de foncteurs :
  - Générateurs.
  - Foncteurs unaires.
  - Foncteurs binaires.
- De plus, elle définit une catégorie de foncteurs dont la tâche est de manipuler des foncteurs adaptables :
  - Adaptateurs de foncteurs.

Générateurs

- Les générateurs sont des foncteurs **sans paramètre** :

```
fonc0();
```

- Plusieurs appels successifs d'un générateur peuvent retourner différentes valeurs.
- Un générateur a le droit de faire des entrées et des sorties, changer l'état de ses variables locales ou toutes autres actions possibles pour une fonction.
- Ils sont souvent utilisés pour attribuer des valeurs aux éléments des conteneurs.

Exemple Foncteur Générateur

```
class Aleatoire
{ public:
   Aleatoire(int Inf, int Sup);
   void setInfSup(int Inf, int Sup);
   int operator()();
   private:
    int inf_,sup_;
};
Aleatoire::Aleatoire(int Inf, int Sup):inf_(Inf),sup_(Sup)
{  srand ( time(NULL) );
}
void Aleatoire::setInfSup( int Inf, int Sup)
{  inf_=Inf;
   sup_=Sup;
}
int Aleatoire::operator ()()
{ return inf_+ rand()%(sup_-inf_);
}
```

Exemple Foncteur Générateur

```
Aleatoire UnObjet(15,35);
for ( int i =0; i < 10;i++)
    cout << UnObjet()<< endl;
```

Foncteurs Unaires

- Foncteurs à **un seul paramètre** :

```
fonc1(type1 param1);
```

- Si le type de retour est `bool`, il s'agit d'un prédicat.
- Ils peuvent hériter de `unary_fonction` pour être adaptables.
- Ils ne sont pas obligés, même dans le cas de prédicat, de retourner la même valeur en passant plusieurs fois le même argument
- Ex : `negate<T>` est un foncteur adaptable de la STL qui agit comme si on multipliait par -1 :

```
negate<int> neg;
cout << neg(-8) << endl;     // affiche : 8
cout << neg(16) << endl;     // affiche : -16
```

Foncteurs Binaires

- Foncteurs à **deux paramètres** :

```
fonc2(type1 param1, type2 param2);
```

- Si le type de retour est `bool`, il s'agit d'un prédicat binaire.
- Ils peuvent hérités de `binary_fonction` pour être adaptables.
- Ils ne sont pas obligés, même dans le cas des prédicats binaires, de retourner la même valeur en passant plusieurs fois les mêmes arguments.
- Ex : `plus<T>` est un foncteur adaptable de la STL qui effectue une somme :

```
plus<int> add;
cout << add(14,-2) << endl;      // affiche : 12
cout << add(23,5)  << endl;      // affiche : 28
```

Foncteurs adaptables

- Les foncteurs adaptables peuvent être passés à des adaptateurs de foncteurs afin d'être altérés ou combinés.
- La STL fournit la plupart des opérateurs classiques sous la forme de foncteurs adaptables :
  - `plus<T>, minus<T>, …`
  - `equal_to<T>, greater<T>, …`
  - `logical_and<T>, logical_or<T>, …`

Foncteurs adaptables

- Les foncteurs adaptables sont des foncteurs qui utilisent des **`typedef`** pour définir le type des données qu'ils manipulent.
- De cette particularité, ils ne peuvent pas être des fonctions ou des pointeurs vers des fonctions C puisque les types sont appelés sous la forme : **`FoncX::data_type.`**
- Pour écrire des foncteurs adaptables, on peut soit définir les **`typedef`** requis ou tout simplement hériter de la classe spécifique au nombre de paramètres définis par la STL :
  - aucun pour les générateurs
  - **`unary_fonction`** : pour les fonctions unaires
  - **`binary_fonction`** : pour les fonctions binaires

Foncteurs Adaptables (suite)

- Dépendamment de la catégorie de foncteurs, ils définissent des types différents :
  - Générateurs :
    - F0::`result_type` : type de retour
  - Foncteurs unaires :
    - F1::`result_type` : type de retour
    - F1::`argument_type` : type du paramètre
  - Foncteurs binaires :
    - F2::`result_type` : type de retour
    - F2::`first_argument_type` : type du premier paramètre
    - F2::`second_argument_type` : type du deuxième paramètre

Foncteurs Adaptables (exemple)

- Construction d'un foncteur unaire adaptable :

```
class Ajouter : public unary_function<int,int>
{
public :
        Ajouter(int increment) : increment_(increment) {}
        operator() (int& valeur) { valeur+= increment_;}
};

    // Équivalent à

class Ajouter
{
public :
    typedef int argument_type;
    typedef int result_type;
    Ajouter(int increment) : increment_(increment) {}
    operator() (int& valeur) { valeur+= increment_;}
};
```

Adaptateurs de Foncteurs

- La STL définit quatre types d'adaptateurs de foncteurs :
  - Conversion d'un foncteur binaire en foncteur unaire en assignant une constante à l'un des paramètres :
    - `bind1st`
    - `bind2nd`
  - Conversion d'un foncteur non-adaptable en adaptable en produisant un nouveau foncteur qui définit les `typedef`s requis
    - `pointer_to_unary_function`
    - `pointer_to_binary_function`
  - Négation du résultat d'un prédicat :
    - `unary_negate`
    - `binary_negate`
  - Composition de foncteurs :
    - `unary_compose` f(x) et g(x) -> (f o g)(x) == f(g(x))
    - `binary_compose` f(x), g1(x) et g2(x) -> f(g1(x), g2(x))

Adaptateurs de Foncteurs (exemple)

```
// Trouve le premier élément dans la liste qui est dans l'intervalle [1, 10]

list<int> L;

/* … Remplissage de la liste L … */

list<int>::iterator it =
find_if
(
     L.begin(),
     L.end(),
     compose2
     (
          logical_and<bool>(),
          bind2nd(greater_equal<int>(), 1),
          bind2nd(less_equal<int>(), 10)
     )
);
```

Déclaration de la liste **L**

Déclaration d'un itérateur **it**

Adaptateurs de Foncteurs (exemple)

```
// Trouve le premier élément dans la liste qui est dans l'intervalle [1, 10]

list<int> L;


/* … Remplissage de la liste L … */

list<int>::iterator it =
find_if
(
     L.begin(),
     L.end(),
     compose2
     (
          logical_and<bool>(),
          bind2nd(greater_equal<int>(), 1),
          bind2nd(less_equal<int>(), 10)
     )
);
```

find_if : algorithme de la STL qui retourne le premier itérateur pour lequel le prédicat retourne vrai

Adaptateurs de Foncteurs (exemple)

```
// Trouve le premier élément dans la liste qui est dans l'intervalle [1, 10]

list<int> L;


/* … Remplissage de la liste L … */

list<int>::iterator it =
find_if
(
     L.begin(),
     L.end(),
     compose2
     (
          logical_and<bool>(),
          bind2nd(greater_equal<int>(), 1),
          bind2nd(less_equal<int>(), 10)
     )
);
```

L.begin() et **L.end()** délimite l'intervalle sur lequel l'algorithme sera appliqué

Adaptateurs de Foncteurs (exemple)

```
// Trouve le premier élément dans la liste qui est dans l'intervalle [1, 10]

list<int> L;


/* … Remplissage de la liste L … */

list<int>::iterator it =
find_if
(
     L.begin(),
     L.end(),
     compose2
     (
          logical_and<bool>(),
          bind2nd(greater_equal<int>(), 1),
          bind2nd(less_equal<int>(), 10)
     )
);
```

Fonction aidante qui crée un foncteur `binary_compose`

Adaptateurs de Foncteurs (exemple)

```
// Trouve le premier élément dans la liste qui est dans l'intervalle [1, 10]

list<int> L;


/* … Remplissage de la liste L … */

list<int>::iterator it =
find_if
(
     L.begin(),
     L.end(),
     compose2
     (
          logical_and<bool>(),
          bind2nd(greater_equal<int>(), 1),
          bind2nd(less_equal<int>(), 10)
     )
);
```

Foncteur englobante
f(g1(x), g2(x)**)**

Foncteur argument1
f(**g1(x)**, g2(x))

Foncteur argument2
f(g1(x), **g2(x)**)

Conclusion

- Ils sont très utilisés par la STL, car les algorithmes ne manipulent pas directement les éléments des conteneurs, c'est plutôt les foncteurs qui en prennent la responsabilité
- Les foncteurs jouent **un grand rôle dans la réutilisabilité du code en réduisant le couplage entre données et algorithmes**
- Les adaptateurs de foncteurs sont très intéressants, car ils permettent de produire des manipulations complexes à partir de foncteurs simples comme `less<T>` (opérateur<) et `logical_and<T>` (opérateur&&)