Programmation fonctionnelle avec C++

Xavier JUVIGNY

ONERA

22 Juillet 2017

Plan du cours

Le C++ : un langage principalement orienté objet ?

Cas de la STL

- Beaucoup de classes;
- Très peu de classes sont conçues pour être dérivées;
- Très peu de fonctions virtuelles (pures ou non);
- Bien plus de POO dans `iostream`...

Questions

- Le commité de normalisation de C++ a-t'il oublié de concervoir la STL en orienté objet ?
- Est-e que le C++ est un langage orienté objet ?

Ben, ça dépend...

- Oui si vous suggérez que le C++ supporte la programmation orienté objet;
- Non si vous suggérez que c'est son principal paragdime de programmation.

Problèmes liés avec la POO

Ne répond pas à des questions essentielles

- Comment écrire de meilleurs algorithmes composables ? :
  La POO est basé sur le vieux principe de la programmation impérative;
- Comment écrire du code rapide et de haut niveau ? :
  Les structures de données de la POO souvent trop lents à cause des indirections et de la perte d'info sur le type à la compilation (type erasure);
- Comment maîtriser la complexité dans des logiciels multithreadés ? :
  La POO ne dit rien à ce sujet.

On a donc besoin d'un autre paradigme :
la plupart des programmeurs C++ le savent déjà...

Qu'est ce que la programmation fonctionnelle ?

Principes de base

- Les fonctions sont les acteurs principaux;
- On ne peut pas changer l'état des objets du programme;
- Les fonctions sont pures, composables, génériques et reutilisables;
- Fait un lourd usage du système de typage;
- Exemple de langage fonctionnel : Scheme (1975), Lisp (1984), Haskell (1987), Caml, ...

Et en C++ ?

- C++ n'est pas vraiment un langage fonctionnel;
- Mais beaucoup d'applications ont besoin d'être très rapides tout en gérant beaucoup de complexité;
- C++ est rapide et la programmation fonctionnelle aide à gérer la complexité.
- Les programmeurs C++ peuvent souvent bénéficier de fonctionnalités (et de codage) bien pensées;
- Le C++ 11 a introduit un nombre intéressant d'éléments de programmation fonctionnelle.

Programmer dans un style fonctionnel

Les services proposés par le C++

- Déduction automatique de type avec `auto` et `decltype`;
- Support les *fonctions lambda* :
- Application partielle fonctionnelle : `std::function` et `std::bind` + fonctions lambdas
- Fonctions de plus grand ordre dans la STL;
- *Manipulation de listes* au travers des **variadic templates**;
- Correspondance de motif à l'aide de la spécialisation partielle ou non des templates;
- Évaluation paresseuse avec `std::async`;
- Template avec types contraints (en utilisant `static_assert`).

Déduction automatique de type

Déduction automatique de type avec auto

- `auto` joue le rôle d'un paramètre template :

```
int x = 22;              // as before
const int cx = x;        // as before
const int& rx = x;       // as before
auto& v1 = x;            // v1's type   int& (auto   int)
auto& v2 = cx;           // v2's type   const int&  (auto   const int)
auto& v3 = rx;           // v3's type   const int&  (auto   const int)
const auto& v4 = x;      // v4's type   const int&  (auto   int)
const auto& v5 = cx;     // v5's type   const int&  (auto   const int)
const auto& v6 = rx;     // v6's type   const int&  (auto   const int)
```

Déduction des types par observation

- Utilisation du mot clef `decltype` $\equiv$ type déclaré pour nom
- Contrairement à `auto`, ne supprime jamais les const/volatile/references.

```
int x = 10;          // decltype(x)    int
const auto& rx = x; // decltype(rx)   const int&
int arr[10];
arr[0] = 5;          // decltype(arr[0])   int&
```

- Mais si on met une variable en "expression" :

```
int x; // decltype(x)   int, decltype((x))   int&
```

Niveau d'abstraction des fonctions

Qu'est ce donc ?

- Une fonction d'ordre plus élevé est une fonction qui prend pour argument des fonctions et retourne d'autres fonctions;
- Les fonctions dans `algorithm` en sont de bon exemples
- L'introduction de fonctions lambdas dans C++ 11 permet de dire officiellement que cette approche est encouragée en C++.

Les fonctions lambdas

Définition

- Les fonctions lambdas sont des objets fonctions anonymes;

```
// Trie par valeur absolue avec std::sort de algorithm :
std::sort(std::begin(v),std::end(v), [](double x, double y){ return std::abs(x)<std::abs(y); });
auto func = [] () { cout << "Hello world"; };
func();
```

- Capture de variables, plusieurs modes :
  - [] Ne capture rien (fonction standard)
  - [&] Capture les variables visibles par référence
  - [=] Capture les variables visibles en les copiant
  - [=,&foo] : Capture les variables visibles en les copiant sauf foo capturée par référence
  - [bar] : Ne capture que la variable bar;
  - [this] : Capture le pointeur sur l'objet courant.

```
double scal;
auto homothetie = [scal](double x){return scal*x;};
scal = 2; std::cout << homothetie(3.) << std::endl;// Affiche 6
scal = 4; std::cout << homothetie(3.) << std::endl;// Affiche 12
```

- Pour prendre un lambda en paramètre, utiliser un paramètre template :

```
template<typename Iterator, typename F> void for_each(Iterator b, Iterator e, F f ) {
    for (; b != e; ++b ) f(*b);
}
```

Propriétés des fonctions lambda

Avantages

- N'introduisent pas de surcout supplémentaire en temps d'appel;
- Chaque lambda est mis dans son propre type anonyme;
- Le corps de la fonction est mis dans l'opérateur membre `operator()` du type;
- Les variables capturées sont mises comme attributs de la classes
- les lambdas jouent subtilement entre comment les templates marchent et comment les compilateurs font du inlining pour l'appel des fonctions.

Inconvénients en C++ 11

- Ne peuvent pas avoir d'arguments génériques;
- Ils ne peuvent pas capturer des variables par déplacement;
- Le type anonyme ne peut pas être nommé donc les lambdas ne peuvent pas être retournées d'une fonction;
- Une fonction lambda ne peut pas être utilisée dans une constexpr.

Améliorations en C++ 14

- Les arguments peuvent être génériques : `auto f= [] ( auto& a, auto& b ) { return a*b; };`
- Déduction du type de retour automatique dans les fonctions standards : les lambdas peuvent être maintenant retournées par une fonction (dont les fonctions lambda !)
- Capture des listes généralisées;

Manipulation des fonctions lambdas

std::function

- Permet de passer une fonction lambda à une fonction non template ou bien de stocker une fonction lambda pour une utilisation ultérieure

```
std::function<int(int)> f = [] (int x) { return x * 2; };
```

- C'est un wrapper polymorphique autour de tout objet appelable;
- Mis en œuvre autour de technique d'oublie de types;
- Rajoute un surcoût de fait de l'appel indirect et du non inlining;
- Ce surcoût est le même que pour tout autre langage supportant les fonctions lambdas;
- Pour le passage d'une fonction en argument, préférer passer la fonction en template.

Combinaison d'ordre plus élevé

- Style fonctionnel pour combiner des fonctions pour ordre plus élevé possible en C++
- Les ingrédient clefs : fonctions lambdas, type automatique, déduction, etc.
- C++ rend les choses un peu plus complexe : gérer le passage parfait aux suivants, réduire les opérations de copie/ déplacement, etc.
- Beaucoup de librairies nous permettent d'avoir du fun sans trop de mal de tête : Boost Fusion, Boost Hana, Fit, range ...

Exemple : Composition de fonctions

Ce qu'on aimerait avoir

```
auto f(std::vector<float>) -> float;
auto g(std::string)        -> std::vector<float>
auto h(std::istream&)      -> std::string

auto fgh = compose(f,g,h);
int x = fgh(std::cin); // x = f(g(h(std::cin)));
```

Facile à faire en C++ 14

```
template<typename F> auto compose(F f) { return [=](auto x) { return f(x); } };
template<typename F, typename... Fs> auto compose(F f, Fs... fs) {
    return [=](auto x){ return f(compose(fs...)(x)); };
}
```

Curryfication

Curryfication

- Une fonction “curryfiée” est invoquée avec un seul paramêtre, même si elle en attend plusieurs;
- Dans ce cas, elle renvoie une fonction qui attend les arguments manquant.

Avec la STL

```
using namespace std::placeholders;
auto f = std::function<int(int,int,int)>([](int x, int y, int z) { return x*y+z; });
auto s = std::bind(f,4,_1,_2);  auto t = std::bind(s,5,_1);
auto w = t(6);
std::cout << "w = " << w << std::endl;
```

Avec Boost Hana (gratuit, template, compatible C++98, ...)

```
using namespace boost::hana;
auto f = curry<3>([](int x, int y, int z) { return x*y + z; });
auto s = f(4); auto t = s(5);
auto w = t(6);
std::cout << "w = " << w << std::endl;
```

Fonction de mappage

La fonction `transform`

- Tout langage fonctionnel a une fonction `map`
- L'équivalent de la STL C++ est la fonction `std::transform`.

```
std::vector<int> v = {1,2,3,4};
std::transform(std::begin(v),std::end(v), std::begin(v), [](int x) { return x * 2; });
```

- Simple, rapide...
- Mais si vous voulez changer le type d'éléments ?

```
std::vector<int> v = {1,2,3,4};
std::vector<std::string> outs;
std::transform(std::begin(v),std::end(v),std::back_inserter(outs),
               static_cast<std::string(*)(int)>(std::to_string));
std::for_each(std::begin(outs),std::end(outs),[](std::string s) { std::cout << s << "␣"; });
std::cout << std::endl;
```

- Gestion explicite de la taille du vecteur de sortie...(`std::back_inserter`);
- Puis je composer ces fonctions ?

```
std::vector<T1> input;
std::vector<T2> step_one;
std::vector<T3> output;
std::transform(std::begin(input),std::end(input),std::back_inserter(step_one), &f);
std::transform(std::begin(step_one),std::end(step_one),std::back_inserter(output), &g);
```

- Peu lisible et lent !

Utilitaires fonctionnels

Fonction objet surchargée (uniquement avec hana)

- Fonction spécialisant ses actions selon le type tout en restant générique

```
auto f = boost::hana::overload( [] (std::string s) { return s + s; },
                                [] ( auto       x) { return x * 2; });
std::vector<int> vint = {1,3,5,7,9,11};
std::vector<std::string> vstr {"Tin", "Ouaf", "Cou" };
std::transform(std::begin(vint), std::end(vint), std::begin(vint), f);
std::transform(std::begin(vstr), std::end(vstr), std::begin(vstr), f);
std::for_each(vint.begin(),vint.end(),[](int i) { std::cout << i << "␣"; });
std::cout << std::endl;
std::for_each(vstr.begin(),vstr.end(),[](std::string s) { std::cout << s << "␣"; });
std::cout << std::endl;
```

Application partielle d'opérateurs binaires

```
using hana::_;
std::vector<int> v = {1,2,3};
std::transform(std::begin(v),std::end(v), std::begin(v), _ * 2);
std::for_each(v.begin(),v.end(),[](int i) { std::cout << i << "␣"; });
std::cout << std::endl;
```

Filtres et réductions

filtres

- Enlève des éléments d'un ensemble
- Utilise `std::remove_if`

```
auto it= std::remove_if(vec.begin(),vec.end(), [](int i){ return !((i < 3) or (i > 8)) });
auto it2= std::remove_if(str.begin(),str.end(), [](string s){ return !(isupper(s[0])); });
```

Réductions

- Réduit un ensemble de valeur à une seule valeur en appliquant successivement un opérateur binaire;

```
std::accumulate(vec.begin(),vec.end(),1, [](int a, int b){ return a*b; });
std::accumulate(str.begin(),str.end(),string(""),[](string a,string b){ return a+":"+b; });
```

Fonctions pures et impures

Caractéristiques des fonctions pures

- Fonctions pures : Fonctions ne modifiant pas les valeurs de ses variables;
- Fonctions impures: les autres fonctions.

Fonction pure	Fonction impure
Produit toujours même résultat avec les mêmes paramêtres	Peut produire des résultats différents avec même paramètres
Aucun effet de bord	Peut avoir des effets de bord
N'altère jamais des états	peut altérer des états même globaux
Facile à reordonner ou exécuter multithread	Analyse difficile de reordonnancement

Fonctionalités manquantes

Évaluation paresseuse

- N'évalue une expression que si nécessaire;
- Exemple marchant en Haskell :

```
length [ 1\2, 4-3, 2*3+1, 1/0, 7*3 ]
```

- On peut faire de même en C++, mais uniquement à la compilation

```
 template <typename... Args> void mySize(Args... args) {
    cout << sizeof...(args) << endl; }
mySize("Rainer",1/0);
```

Compréhension de listes

- Une fonctionnalité range existe depuis C++ 2020 (comme Haskell, Python, ...)

```
a = [ i for i in range(1000) if i%3==0 and i%5==0 ]
```

- Prévu initialement pour C++17, mais reporté pour C++ 20;
- Néanmoins pour les versions antiérieures à C++ 20, on peut utiliser Boost.Range ou range d'Éric Niebler (c'est cette bibliothèque qui a servi de base pour la norme C++ 20)

Range en C++ 20

Exemple d'utilisation de range en C++ 20

```
#include <ranges>
#include <iostream>

int main()
{
    auto const ints = {0,1,2,3,4,5};
    auto even = [](int i) { return 0 == i % 2; };
    auto square = [](int i) { return i * i; };

    // "pipe" syntax of composing the views:
    for (int i : ints | std::views::filter(even) | std::views::transform(square)) {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    // a traditional "functional" composing syntax:
    for (int i : std::views::transform(std::views::filter(ints, even), square)) {
        std::cout << i << ' ';
    }
}
```

Encore récent, et pas beaucoup de documentation dessus (à part la norme...).

Exercices

Trie selon l'axe des abcisses

- Écrire un programme "minimaliste" permettant de trier des vecteurs de dimension 3 `std::array<double,3>` selon les abcisses;
- Pour l'algorithme de tri, on regardera la fonction `std::sort` proposée par la STL (dans `algorithm`).

Composition de fonctions et vecteur

- Utiliser la composition de template vu dans ce chapitre pour construire des expressions arithmétiques pouvant être appliquées sur chaque composante de vecteurs de dimension N.