Généricité, concepts et C++14

Flob90 a écrit un article sur les services en C++11 : Écriture de services en C++11, Partie 1/2. Dans celui-ci, il utilise beaucoup les concepts, mais ne présente pas d'implémentation pour les vérifier. De plus, cet article ne s'adresse pas à des débutants et il ne leur sera probablement pas facile de comprendre en quoi les concepts sont un moyen simple et puissant d'améliorer la généricité de leur code. Dans cet article, je vais tenter de montrer comment des débutants en C++ peuvent les utiliser.

Généricité

Le premier code d'exemple du tutoriel de Flob90 se base sur la fonction advance. Celle-ci permet d'incrémenter une variable ayant une sémantique d'itérateur de N éléments dans un conteneur de séquences. Pour rappel, un conteneur de séquences est une collection ordonnée d'éléments qui définit un premier et un dernier élément. Par exemple, un tableau ou une liste chaînée sont des conteneurs de séquences, alors qu'un graphe n'en est pas un. Un itérateur (`InputIterator`) peut être décrit comme une forme de pointeur, qui référence un élément et qui permet de passer à l'élément suivant dans le conteneur.

Imaginons le cas d'utilisation suivant : vous avez un tableau d'entiers (`std::vector<int>`) et vous souhaitez déplacer un itérateur de N éléments. Un exemple d'implémentation est de créer une boucle qui incrémente N fois l'itérateur :

```
#include <iostream>
#include <vector>

void advance(std::vector<int>::iterator & it, unsigned int N)
{
    while (N--)
        ++it;
}
```

```
int main()
{
    std::vector<int> v = { 0,1,2,3,4,5,6,7,8,9 };
    std::vector<int>::iterator it = v.begin();
    advance(it, 5);
    std::cout << *it << std::endl;
}
```

On comprend vite le problème posé par ce code : si vous utilisez un autre type de conteneur (par exemple `std::vector<double>` ou `std::list<object>`), alors le code précédent ne fonctionnera pas. Il faudra donc écrire plusieurs fonctions advance, pour chaque type de conteneur que vous utiliserez.

Pour éviter cela, vous allez créer une fonction générique, qui accepte n'importe quel type de conteneur de séquences, en utilisant les templates C++. Le code précédant devient alors :

```
template <class T>
void advance(T & it, unsigned int N) {
    while (N--)
        ++it;
}
```

Lors de la compilation, le type `T` sera remplacé par les types réellement utilisés et le compilateur générera une fonction `advance` pour chaque type utilisé.

Remarque : une amélioration possible d'implémentation est de surcharger la fonction pour les conteneurs à accès aléatoire, qui autorisent d'écrire directement `it+=N`. Ce type d'approche sort du cadre de cet article, le lecteur intéressé pourra rechercher "tag dispatching".

Erreurs de compilation

Imaginons que l'on appelle la fonction advance avec une variable constante :

```
const std::vector<int>::iterator it = v.begin();
```

```
advance(it, 10);
```

Il n'est bien sûr pas possible d'incrémenter une variable constante et ce code produit donc naturellement une erreur :

```
In file included from
/usr/include/c++/4.8/bits/stl_algobase.h:66:0,
                 from /usr/include/c++/4.8/vector:60,
                 from prog.cpp:1:
/usr/include/c++/4.8/bits/stl_iterator_base_funcs.h: In
instantiation of ‘void
    std::__advance(_RandomAccessIterator&, _Distance,
std::random_access_iterator_tag) [with
    _RandomAccessIterator = const
__gnu_cxx::__normal_iterator<int*, std::vector<int> >;
    _Distance = int]’:
/usr/include/c++/4.8/bits/stl_iterator_base_funcs.h:177:61:
required from ‘void std::advance(
    _InputIterator&, _Distance) [with _InputIterator = const
__gnu_cxx::__normal_iterator<int*,
    std::vector<int> >; _Distance = int]’
prog.cpp:12:19:   required from here
/usr/include/c++/4.8/bits/stl_iterator_base_funcs.h:156:11:
error: passing ‘const
    __gnu_cxx::__normal_iterator<int*, std::vector<int> >’
as ‘this’ argument of
    ‘__gnu_cxx::__normal_iterator<_Iterator, _Container>&
__gnu_cxx::__normal_iterator<_Iterator,
    _Container>::operator+=(const difference_type&) [with
_Iterator = int*; _Container = std::vector<int>;
    __gnu_cxx::__normal_iterator<_Iterator,
_Container>::difference_type = int]’ discards qualifiers
    [-fpermissive]
       __i += __n;
           ^
```

On arrive à l'une des critiques habituelles sur les templates C++ : les messages d'erreurs sont parfois un peu… cryptiques.

Un autre problème est qu'il n'y a pas de contrainte sur les arguments templates utilisés : alors que l'on a correctement défini des conditions

d'utilisation de la fonction advance, il n'est pas possible d'écrire explicitement ces contraintes dans le code.

Les concepts en C++11

Définition

Les concepts apportent une solution aux problématiques décrites précédemment. Commençons par définir ce que sont les concepts :

> Les concepts sont un ensemble de contraintes que devra respecter les arguments templates.

En pratique, si l'on reprend l'exemple de la fonction advance, cela veut dire que l'on ne va pas accepter n'importe quel type pour T, mais uniquement les types qui respectent la contrainte "est un itérateur valide, non constant et possédant l'opérateur ++".

Pour simplifier, les concepts sont donc un moyen de vérifier les conditions d'utilisation des classes et fonctions. Ces conditions sont en général définies dans la documentation, mais ne font pas partie de l'interface de la classe ou de la fonction. En cas d'erreur lors de l'utilisation de ces classes ou fonctions, l'erreur de compilation est générée à l'endroit du code qui nécessite le respect de ces contraintes et non directement au début de la classe ou fonction.

Un exemple d'utilisation des concepts est la bibliothèque standard (STL). Si l'on regarde la documentation de std::advance par exemple, on voit que la définition utilise deux paramètres template nommés `InputIt` et `Distance` et que `InputIt` doit respecter la contrainte des InputIterator (c'est-à-dire qu'il doit être un itérateur, être comparable avec l'égalité, être déréférençable, être incrémentable et d'autres conditions).

Ajouter une contrainte dans une fonction

Pour implémenter les concepts en C++11, vous allez utiliser deux nouvelles fonctionnalités du C++11 : les assertions statiques et les traits.

Les assertions permettent de valider une expression et déclenchent une erreur lorsque cette expression est fausse. Elles existent sous deux formes : les assertions à la compilation (en C++11 ou avec Boost) et à l'exécution. La première forme utilise la fonction `static_assert` et prend deux paramètres : une expression constante retournant un booléen et un message d'erreur à afficher :

```
static_assert(1==2, "1 n'est pas égal à 2");
```

Ce code provoque le message d'erreur suivant à la compilation :

```
prog.cpp: In function 'int main()':
prog.cpp:4:1: error: static assertion failed: 1 n'est pas
égal à 2
 static_assert(1==2, "1 n'est pas égal à 2");
 ^
```

La seconde forme utilise la macro `assert`, qui prend comme paramètre une expression qui retourne un booléen :

```
#include <cassert>
assert(1==2);
```

Ce code retourne le message d'erreur suivant à l'exécution :

```
prog: prog.cpp:5: int main(): Assertion `1==2' failed.
```

Les traits permettent d'obtenir des informations sur des types ou de les modifier. Vous trouverez sur la page de documentation du C++ la liste des traits disponibles : Type support – Type Traits. Vous allez pouvoir utiliser les traits `is_xxx` avec `static_assert` pour vérifier qu'un type respecte une condition. Ainsi, pour revenir au cas d'exemple de la fonction `advance`, vous pouvez écrire le code suivant :

```
template <class T>
```

```
void advance(T & it, unsigned int N) {
    static_assert(std::is_pointer<T>::value,
        "Function advance: the T type is not a pointer");

    while (N--)
        ++it;
}
```

Dans ce cas, la contrainte “est un pointeur” est vérifiée dès la compilation et retourne un message d’erreur spécifique et compréhensible. Si on appelle cette fonction avec un type non valide (par exemple une référence), on obtient le message d’erreur suivant :

```
prog.cpp: In instantiation of ‘void advance(T&, unsigned int)
[with T = Object]’:
prog.cpp:20:25:   required from here
prog.cpp:8:5: error: static assertion failed: Function
advance: the T type is not a pointer
     static_assert(std::is_pointer <T>::value,
     ^
prog.cpp:12:9: error: no match for ‘operator++’ (operand
type is ‘Object’)
         ++it;
         ^
```

On voit qu’il subsiste encore des problèmes de lisibilité des messages d’erreur : le message ajouté dans l’assertion est mélangé parmi du bruit, ce qui complique un peu la lecture.

Remarque : avec cette implémentation des concepts, ceux-ci n’apparaissent pas explicitement dans l’interface de la fonction, ce qui réduit leur intérêt. Les concepts de la prochaine norme du C++ seront directement définis dans l’interface des classes et fonctions, voir la dernière partie de ce tutoriel.

Ajouter plusieurs contraintes à une fonction

Dans le cas général, vous n’aurez pas une seule contrainte à vérifier,

mais plusieurs. Vous avez deux solutions pour tester ces contraintes : soit vous regroupez plusieurs expressions avec l'opérateur AND dans une seule insertion, soit vous écrivez plusieurs assertions. La seconde version permet d'avoir un message d'erreur spécifique pour chaque expression, la première est plus concise.

Si l'on reprend le code d'exemple sur la fonction advance, on peut ajouter la contrainte "n'est pas constant" avec la première forme :

```
static_assert(std::is_pointer<T>::value &&
              !std::is_const<T>::value,
    "The T type is not a mutable pointer");
```

Avec la seconde forme, le code serait le suivant :

```
template <class T>
void advance(T & it, unsigned int N) {
    static_assert(std::is_pointer<T>::value,
        "The T type is not a pointer");
    static_assert(!std::is_const<T>::value,
        "The T type is const");

    while (N--)
        ++it;
}
```

Hiérarchie de concepts et regroupement des contraintes

On voit qu'un type doit finalement respecter un nombre important de contraintes et il va être lourd d'exprimer explicitement dans chaque fonction chacune de ces contraintes. Pour éviter cela, on va créer des hiérarchies de concepts, chaque concept étant défini à partir d'autres concepts et des contraintes propres.

Si on regarde par exemple les itérateurs de la biblitohèque standard, on voit qu'il existe une hiérarchie dans les concepts :

- `RandomAccessIterator` est un `BidirectionalIterator` avec un accès aléatoire ;
- `BidirectionalIterator` est un `ForwardIterator` avec la décrémentation ;
- `ForwardIterator` est un `InputIterator` permettant le passage multiple ;
- `InputIterator` est déréférençable et incrémentable.

On retrouve ce type de hiérarchie dans d'autres modules de la bibliothèque standard, comme par exemple les entrées-sorties.

Ainsi, nous allons modifier le code d'exemple précédent pour refactoriser : au lieu d'exprimer chaque contrainte directement dans la fonction `advance`, nous allons créer un trait `is_iterator`, regroupant les contraintes appliquées sur les itérateurs.

Pour cela, nous allons simplement créer une nouvelle structure `is_iterator` qui contient une seule variable membre (constante et statique) value. La valeur de cette variable est initialisée à partir des expressions booléennes permettant de vérifier que le type est un pointeur et n'est pas constant. Le code suivant présente un exemple d'implémentation :

```
template <class T>
struct is_iterator {
    static const bool value = std::is_pointer<T>::value &&
                              !std::is_const<T>::value;
};
```

Le code de la fonction `advance` devient alors :

```
template <class T>
void advance(T & it, unsigned int N) {
    static_assert(is_iterator<T>::value,
        "The T type is not a iterator");

    while (N--)
        ++it;
}
```

Remarque 1 : le concept d'itérateur est plus complexe que simplement vérifier que le type est un pointeur et n'est pas constant. Dans le cadre de cet article, nous n'entrerons pas plus dans les détails.

Remarque 2 : la création de ces traits supplémentaires peut paraître lourd, surtout si ne vous les utilisez qu'à un seul endroit dans votre code. Cela nécessitera de créer des fichiers supplémentaires, qui ne seront utilisés qu'une seule fois. Cependant, en termes de conception, cette approche respecte le principe ouvert-fermé (chaque trait étant minimaliste et spécialisé, il ne sera pas nécessaire de le modifier, sauf erreur de conception ou d'implémentation) et renforce donc la réutilisation du code et sa robustesse. N'hésitez pas à vous créer votre propre bibliothèque de traits réutilisables (ce sont des templates, donc facilement déployables – il faut juste inclure les fichiers – et n'ont pas de coût à l'exécution) ou à utiliser Boost.TypeTrait.

Les concepts en C++14

Les concepts devaient initialement être ajoutés dans le C++14, mais ils seront finalement disponibles sous forme d'un *Technical Specification* dans le draft Concepts Lite N3929. Nous allons voir rapidement dans cette partie la syntaxe proposée par la prochaine norme.

Remarque : les codes d'exemples présentés dans cette partie sont directement extraits du draft.

Le draft propose deux écritures pour utiliser les concepts. Dans la première, la contrainte est exprimée directement dans les paramètres templates, en remplaçant le nom-clé `class` ou `typename` par le concept que doit respecter le paramètre template.

```
template<Sortable Cont>
void sort(Cont& container);
```

La seconde écriture utilise un nouveau mot-clé `requires`, qui permet d'ajouter une expression booléenne permettant d'exprimer la contrainte :

```
template<typename Cont>
```

```
    requires Sortable<Cont>()
void sort(Cont& cont)
```

Un point important avec les concepts-lite du C++14, c'est que les concepts font partie intégrante de l'interface, contrairement à l'utilisation de `static_assert`. Les messages d'erreurs sont également plus compréhensibles :

```
error: no matching function for call to `sort(list<int>&)'
    sort(l);
         ^
note: candidate is:
note: template<Sortable T> void sort(T)
    void sort(T t) { }
         ^
note: template constraints not satisfied because
note:   `T' is not a/an `Sortable' type [with T = list<int>]
note:   failed requirement with ('list<int>::iterator i',
std::size_t n)
note:     `i[n]' is not valid syntax
```

L'utilisation de `requires` permet de placer des contraintes sur les paramètres template de classe au niveau des fonctions membres :

```
template<Object T, Allocator A>
class vector
{
    vector(const T& v)
        requires Copyable<T>();
    void push_back(const T& x)
        requires Copyable<T>();
};
```

L'implémentation de nouveaux concepts peut être réalisée en déclarant une nouvelle fonction template retournant un booléen :

```
template<typename T>
constexpr bool Equality_comparable()
{
    return requires (T a, T b) {
       {a == b} -> bool;
```

```
        {a != b} -> bool;
    };
}
```

Conclusion et remerciements

On voit au final que l'implémentation de la vérification des concepts (qui ne représentent qu'une partie de l'utilité des concepts) est relativement simple et légère en C++11. J'espère avoir réussi à vous montrer dans cet article l'intérêt à les utiliser, même pour les débutants.

Merci à Winjerome…