C++, Qt, OpenGL, CUDA

predicats

Chapitre précédent	Sommaire principal	Chapitre suivant

Les foncteurs et fonctions lambdas

foncteur ? fonction objet ? opérateur ? autre ?

Les foncteurs par défaut

Vous avez vu dans le chapitre précédent comment utiliser la fonction `std::sort` pour trier une collection :

```
std::vector<int> v { 1, 5, 2, 4, 3 };
std::sort(begin(v), end(v));

std::string s { "hello, world!" };
std::sort(begin(s), end(s));
```

Cet algorithme est dit “modifiant” puisqu'il modifie directement le conteneur sur lequel on applique la fonction.

Fondamentalement, cet algorithme fonctionne de la façon suivante : il parcourt les éléments de la collection et réalise des tests de comparaison par paire d'éléments. Pour faire cette comparaison, l'algorithme utilise l'opérateur de comparaison `<` sur les éléments. Par exemple, pour faire le tri d'un tableau d'entiers (`vector<int>`), l'algorithme réalise des comparaisons d'entiers (`valeur 1 < valeur 2`).

Dit autrement, cela veut dire que si on utilise un `vector<un_type>`, il faut que la comparaison `<` ait un sens pour ce type `un_type` (ce qui sera le cas avec la majorité des types de base du C++).

On dit que l'opérateur `<` est le prédicat utilisé par l'algorithme de trie `std::sort`. Plus généralement, un prédicat est une expression qui retourne un booléen (`true` ou `false`). Les différents algorithmes de la bibliothèque standard n'utilisent pas tous l'opérateur `<`, certains utilisent l'opérateur d'égalité `==`, d'autres n'utilisent pas de prédicat.

Imaginons maintenant que l'on souhaite trier une collection dans l'ordre inverse, c'est-à-dire du plus grand au plus petit. Une première solution serait de trier dans l'ordre par défaut (plus petit au plus grand), puis d'inverser l'ordre des éléments. Une autre solution serait de réécrire un algorithme de tri (appelé `reverse_sort` par exemple) et qui trie dans l'ordre inverse (du plus grand au plus petit).

Ces deux solutions ne sont pas correctes en termes de C++ moderne. La première est inutilement plus compliqué (il faut écrire deux lignes au lieu d'une seule), la seconde demande de réécrire l'algorithme de tri.

Les foncteurs de la bibliothèque standard

Heureusement, la bibliothèque standard a été conçue pour être le plus générique possible, suivant les principes de la programmation moderne. Pour cela, la majorité des algorithmes de la bibliothèque standard existe en deux versions. La première utilise les foncteurs par défaut, la seconde admet un argument supplémentaire permettant de fournir un foncteur personnalisé. Par exemple, la fonction `sort` peut s'utiliser avec le prédicat par défaut (utilisation de `<`) ou un foncteur personnalisé :

```
std::sort(begin(v), end(v));              // foncteur par défaut
std::sort(begin(v), end(v), un_foncteur); // foncteur personnalisé
```

Les cas les plus génériques, comme trier du plus grand au plus petit, sont déjà implémentés dans la bibliothèque standard. Ces prédicats sont définis dans le fichier d'en-tête `<functional>`. Par exemple, pour trier du plus grand au plus petit, il est possible d'utiliser le prédicat `greater` (“plus grand que”) de la façon suivante :

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

int main() {
    std::vector<int> v { 1, 5, 2, 4, 3 };
    std::sort(begin(v), end(v));
    for (auto const value: v)
        std::cout << value << std::endl;
    std::cout << std::endl;

    std::sort(begin(v), end(v), std::greater<int>());
    for (auto const value: v)
        std::cout << value << std::endl;
}
```

affiche :

```
1
2
3
4
5

5
4
3
2
1
```

Le prédicat `greater` est une classe template, il faut préciser le type que l'on souhaite comparer comme argument template (donc entre chevrons). Les parenthèses permettent d'instancier un objet à partir de la classe `greater`. Vous n'avez pas encore vu les classes, vous verrez cela en détail dans la partie sur la programmation objet. Pour le moment, le plus important est de se souvenir de la syntaxe.

Il est possible d'utiliser les foncteurs directement, sans passer par un algorithme. Pour cela, il faut créer un objet du type `std::greater`, puis l'appeler comme une fonction (d'où leur nom : foncteur = “function object”) :

main.cpp

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

int main() {
    std::greater<int> greater_operator {};
    std::cout << std::boolalpha;
    std::cout << greater_operator(1, 2) << std::endl;
    std::cout << greater_operator(3, 2) << std::endl;
}
```

Pour bien comprendre la différence entre les divers types de fonction, revoyons les multiples syntaxes pour créer une variable (nommée `object`) d'un type donné (`MyObject`) et sur laquelle nous appliquons une fonction :

```
MyObject object {};

foo(object);  // fonction libre
object.foo(); // fonction membre
object();     // foncteur
```

Pour le moment, vous avez seulement vu comment utiliser ces types de fonction. Vous verrez dans les chapitres sur les fonctions et sur la programmation objet comment les créer. En pratique, il vous suffit de lire la documentation pour savoir quelle est la syntaxe correcte d'une fonction (sa signature, ses paramètres, savoir si c'est une fonction libre ou membre, etc.) Avec l'expérience, vous vous souviendrez des fonctions les plus utilisées et vous n'aurez plus besoin de lire la documentation pour vérifier la syntaxe. Un éditeur de code avec l'auto-complétion (qui affiche la signature des fonctions) peut également être très utile pour gagner du temps.

Vous pouvez consulter la liste des différents foncteurs de la bibliothèque standard dans la page de documentation Function objects [http://en.cppreference.com/w/cpp/utility/functional]. Vous voyez dans cette page que les foncteurs sont rangés en plusieurs catégories :

- les opérations arithmétiques (*arithmetic operations*) : `plus`, `minus`, `multiplies`, `divides`, `modulus` et `negate` ;
- les comparaisons (*comparisons*) : `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal` et `less_equal` ;
- les opérations logiques (*Logical operations*) : `logical_and`, `logical_or` et `logical_not` ;
- les opérations sur les bits (*Bitwise operations*) : `bit_and`, `bit_or`, `bit_xor` et `bit_not`.

composition de fonction, not1, not2, bind ?

Plus généralement, il est possible d'utiliser n'importe quelle fonction de la bibliothèque standard, à partir du moment où celle-ci respecte la signature attendue par un algorithme donné. En particulier, il est possible d'utiliser les fonctions de manipulations de caractères, définies dans le fichier d'en-tête `<cctype>` : Null-terminated byte strings [http://en.cppreference.com/w/cpp/string/byte] (ou leur équivalent pour `wstring` : Null-terminated wide strings [http://en.cppreference.com/w/cpp/string/wide]). Ces fonctions se divisent en deux catégories :

- les fonctions de test (*Character classification*) :
  - `isalnum` (alphanumérique) ;
  - `isalpha` (alphabétique) ;
  - `islower` (minuscule) ;
  - `isupper` (majuscule) ;
  - `isdigit` (chiffre décimal) ;
  - `isxdigit` (chiffre hexadécimal) ;
  - `iscntrl` (caractère de contrôle) ;
  - `isgraph` (caractère graphique) ;
  - `isspace` (espace) ;
  - `isblank` (caractère blanc) ;
  - `isprint` (caractère affichable) ;
  - `ispunct` (ponctuation).
- les fonctions de modification (*Character manipulation*) :
  - `tolower` (convertie en minuscule) ;
  - `toupper` (converti en majuscule).

La syntaxe pour utiliser les fonctions dans un algorithme est différente de celle pour les objets-fonctions. Il faut simplement mettre la nom de la fonction. Par exemple, pour convertir une chaîne de caractères en majuscule, il est possible d'utiliser l'algorithme `transform` (que vous verrez par la suite) et la fonction `toupper` :

main.cpp

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

```
#include <algorithm>

int main() {
    std::string s { "abcdef" };
    std::transform(begin(s), end(s), begin(s), toupper);
    std::cout << s << std::endl;
}
```

affiche :

```
ABCDEF
```

Exercices

- trier avec d'autres prédicats
- combiner des prédicats

Les fonctions lambdas

Si vous avez regardé la documentation des foncteurs de la bibliothèque standard, vous avez peut être compris que ce sont de simples classes. Vous apprendrez dans la partie sur la programmation orientée objet comment créer vos propres foncteurs (que l'on appelle aussi “function object”). Mais il est possible de créer des foncteurs plus simplement, en utilisant des fonctions lambdas.

Il est important que vous sachiez créer des fonctions, c'est un point fondamental en C++, vous les utiliserez dans tous vos codes. Et plus important, ce qui sera fondamental est de savoir découper correctement les problèmes complexes en fonctions plus simples. Ce chapitre ne sera pas suffisant pour étudier toutes les possibilités offertes par les fonctions, nous reviendrons dessus en détail par la suite. Cette partie se focalise sur l'utilisation simple des fonctions lambdas avec les algorithmes de la bibliothèque standard.

Les fonctions lambdas sont une technique issue de la programmation fonctionnelle. Vous avez déjà utilisé des fonctions (membres ou libres) et vous avez déjà défini une fonction : la fonction `main`. Pour rappel, une fonction permet de réaliser une tâche particulière et est constituée de :

- un nom de fonction (`main`) ;
- des valeurs optionnelles (arguments) en entrée et sortie ;
- un corps de la fonction, entre crochets, qui contient les instructions à exécuter lorsque l'on appelle la fonction ;
- il n'est pas possible de définir une fonction dans une autre fonction.

La fonction `main` est une fonction particulière :

- elle est obligatoire ;
- elle ne peut être appelée que par le système (jamais par l'utilisateur) ;
- elle doit être unique ;
- sa signature (c'est-à-dire sa façon d'être écrite) est définie par la norme C++.

```
int main() {
    // corps de la fonction main
}
```

```
Programmation fonctionnelle
```

Une fonction lambda est une fonction particulière. Elle n'a pas de nom (elle est dite anonyme) et peut être déclarée dans le corps d'une autre fonction. A part cela, elle se comporte comme une fonction classique et peut recevoir des arguments, retourner une valeur. Il existe plusieurs façon d'écrire des fonctions lambdas, mais

comme nous les utilisons ici dans les algorithmes de la bibliothèques standard, il est nécessaire de respecter la signature imposée par les algorithmes.

La définition d'une fonction lambda se décompose en trois parties :

```
[capture](paramètres){corps de la fonction}
```

La capture et les paramètres permettent de passer des informations dans la fonction. Pour le moment, nous n'utiliserons pas la capture, uniquement les paramètres. Le corps de la fonction contient le code a exécuter lorsque la fonction est appelée. Les différents algorithmes n'attendent pas la même signature pour les foncteurs, vous verrez dans les prochains chapitres chaque algorithme et la signature de foncteur en détail. Nous verrons ici que l'utilisation de `std::sort` comme exemple de fonction lambda.

Vous connaissez déjà la signature que doit avoir une fonction lambda avec `sort`. En effet, vous savez que cet algorithme peut s'utiliser avec les opérateurs de comparaison `<` ou `>` (`greater`) par exemple. Et vous savez que ces opérateurs prennent deux valeurs, les comparent et retourne une valeur booléenne.

Les valeurs en entrée (paramètres) s'écrivent entre les parenthèses dans la fonction lambda. Chaque paramètre s'écrit en indiquant un type et un nom et les paramètres sont séparés par des virgules. Par exemple, pour écrire une fonction lambda qui prend deux entiers en entrée, on écrit :

```
[](int a, int b){}
```

Pour simplifier l'écriture des fonctions lambdas et avoir un code plus générique, il est possible d'utiliser l'inférence de type et le mot-clé `auto` (voir le chapitre La déduction de type pour rappel). Il est classique de nommer les paramètres `rhs` (*right hand side*, “côté droit”) et `lhs` (*left hand side*, “côté gauche”). La fonction devient donc :

```
[](auto lhs, auto rhs){}
```

Il faut également que cette fonction lambda retourne une valeur booléenne. Pour cela, il faut utiliser le mot-clé `return`, suivi d'une valeur ou d'une expression. Le type retournée est déduit automatiquement en fonction de l'expression écrite après `return`. Par exemple, si on écrit :

```
[](){ return true; }        // retourne une valeur booléenne
[](){ return 123; }         // retourne une valeur entière
[](){ return 12.4 * 45.6; } // retourne une valeur réelle (double)
```

Le corps de la fonction lambda (entre les accolades) peut contenir plusieurs lignes (séparées par un point-virgule), déclarer des variables, appeler d'autres fonctions, etc. Bref, vous pouvez mettre dans le corps d'une fonction lambda toutes les instructions que vous souhaitez.

Pour écrire une fonction lambda qui prend deux paramètres, compare les valeurs et retourne un booléen, on peut donc écrire :

```
[](auto lhs, auto rhs){ return lhs < rhs; }
```

La valeur retournée correspond à l'expression à droite du mot-clé `return`, donc `lhs < rhs`. Le résultat de cette expression est `true` ou `false`, c'est donc bien un booléen. Le code complet avec la fonction `sort` s'écrit :

```
std::sort(begin(v), end(v), [](auto lhs, auto rhs){ return lhs < rhs; });
```

Ce code contient beaucoup d'informations sur une seule ligne, il vous faudra probablement un peu de temps pour réussir à repérer en un coup d'oeil les différents éléments qui la compose. Avec l'expérience, cela ne posera plus de problèmes.

On peut alors facilement écrire une fonction lambda pour trier un tableau du plus petit au plus grand ou l'inverse :

main.cpp

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

int main() {
    std::vector<int> v { 1, 5, 2, 4, 3 };

    std::sort(begin(v), end(v), [](auto lhs, auto rhs){ return lhs < rhs; });
    for (auto const value: v)
        std::cout << value << std::endl;
    std::cout << std::endl;

    std::sort(begin(v), end(v), [](auto lhs, auto rhs){ return lhs > rhs; });
    for (auto const value: v)
        std::cout << value << std::endl;
}
```

affiche :

```
1
2
3
4
5

5
4
3
2
1
```

Remarque : ce code est bien sûr un code d'exemple pour illustrer l'utilisation des fonctions lambdas avec la fonction `sort`. En pratique, la première fonction lambda ne sert à rien, puisque cela reproduit le comportement par défaut de `sort`. Et la seconde reproduit le comportement du foncteur `greater`, elle n'est pas très utile non plus. Vous verrez dans les exercices et les chapitres suivants des exemples d'utilisation plus intéressants des fonctions lambdas avec les prédicats.

Exercices

- trier selon la valeur absolue

Chapitre précédent	Sommaire principal	Chapitre suivant

predicats.txt · Dernière modification: 2019/02/21 23:04 par alavida