Лекція 22

Функтори, або функціональні об'єкти

Функціональні аргументи алгоритмів не зобов'язані бути функціями. На прикладі лямбда-виражень видно, що функціональними аргументами можуть бути об'єкти, що ведуть себе як функції. Такі об'єкти називаються *функціональними об'єктами*, чи *функторами*. Замість використання лямбда-функції можна визначити функціональний об'єкт як об'єкт класу, що містить *операцію виклику функції*. Це було можливе ще до появи стандарту C++11.

22.1. Визначення функціональних об'єктів

Функціональні об’єкти — це ще один приклад прояву мощи узагальненого програмування і концепції чистої абстракції. Усе, що *поводиться* як функція, є *функцією*. Отже, якщо визначити об'єкт, що веде себе як функція, те його можна використовувати як функцію.

Яке ж поводження функції? Функціональне поводження — це щось, що можна викликати за допомогою пари дужок і передачі аргументів. Наприклад:

```
function(arg1,arg2); // виклик функції
```

Якщо потрібно, щоб об'єкти поводилися подібним чином, необхідно зробити можливим “виклик” цих об'єктів за допомогою пари дужок і передачі аргументів. Да, це возможно (у мові C++ можливо майже усіх). Для цього досить оголосити операцію `()` з відповідними типами параметрів.

```
class X {
   public:
      // визначення операції "виклик функції":
      return-value operator() (arguments) const;
      ...
};
```

Тепер об'єкти цього класу можна використовувати як виклик функції.

```
X fo;
...
 fo(arg1,arg2); // виклик operator() з об'єкта-функції fo
```

Цей виклик еквівалентний конструкції.

```
fo.operator()(arg1,arg2); // виклик operator() з функціонального об'єкта fo
```

Розглянемо закінчений приклад. Це варіант функціонального об'єкта з попереднього приклада, що робив тієї ж саме за допомогою звичайної функції:

```
// stl/foreach2.cpp
```

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// простий функціональний об'єкт, що виводить на екран
// передані аргументи
class PrintInt {
  public:
     void operator() (int elem) const {
        cout << elem << ' ';
     }
};

int main()
{
  vector<int> coll;

  // вставляємо елементи від 1 до 9
  for (int i=1; i<=9; ++i) {
       coll.push_back(i);
  }

  // виводимо на екран всі елементи
  for_each (coll.cbegin(), coll.cend(), // діапазон
            PrintInt());                // операція
  cout << endl;
}
```

Клас `PrintInt` визначає об'єкти, до яких можна застосовувати операцію `()` з аргументом типу `int`. Вираз `PrintInt()` в операторі

```
for_each (coll.cbegin(), coll.cend(), PrintInt());
```

створює тимчасовий об'єкт цього класу, що передається алгоритму `for_each()` як аргумент.

Алгоритм `for_each()` може бути записаний у такий спосіб:

```
namespace std {
   template <typename Iterator, typename Operation>
   Operation for_each (Iterator act, Iterator end, Operation op)
   {
      while (act != end) { // пока не достигнут кінець
         op(*act); // - викликаємо op() з поточним елементом
         ++act;    // - переміщаємо ітератор на наступний елемент
```

```
    }
    return op;
  }
}
```

Алгоритм `for_each()` використовує тимчасовий функціональний об'єкт `op` при виклику `op(*act)` для кожного елемента `act`. Якби `op` була звичайною функцією, то алгоритм `for_each()` просто викликав би її з аргументом `*act`. Якщо ж `op` — функціональний об'єкт, то алгоритм `for_each()` викликає її операцію `()` з аргументом `*act`. Таким чином, у цьому прикладі алгоритм `for_each()` виконує наступний виклик:

```
PrintInt::operator()(*act)
```

Може виникнути питання: навіщо все це потрібно? Функціональні об'єкти можуть навіть показатися дивними, незграбними і безглуздими. Дійсно, вони ускладнюють код. Однак функціональні об'єкти — це щось більше, чем просто функції, і завдяки цьому вони мають певні переваги.

1. **Функціональний об’єкт — це “функція зі станом”.** Об’єкти, що поводять себе як вказівники, називаються інтелектуальними вказівниками. Це ж стосується об’єктів, що поводять себе як функції: вони можуть розглядатися як “інтелектуальні функції”, тому що мають додаткові можливості крім операції `()`. Функціональні об'єкти можуть містити інші функції-члени й атрибути. Це означає, що функціональні об'єкти мають стан. Фактично та сама функціональна можливість, виражена двома різними функціональними об'єктами того самого типу, у той самий час може мати різні стани. Для звичайних функцій це неможливо. Іншою перевагою функціональних об'єктів є можливість їх ініціалізації в ході виконання програми до виклику.
2. **Кожен функціональний об’єкт має свій тип.** Звичайні функції мають різні типи, тільки якщо їх сигнатури відрізняються друг від друга. Однак функціональні об'єкти можуть мати різні типи, навіть якщо їхньої сигнатури збігаються. Фактично кожне функціональне поводження, визначене за допомогою функціонального об'єкта, має свій власний тип. Це важливе поліпшення для узагальненого програмування за допомогою шаблонів, тому що тепер з’являється можливість передавати функціональне поводження як шаблонний параметр. У результаті контейнери різних типів можуть використовувати той самий вид функціональних об'єктів як критерій сортування, запобігаючи присвоювання, об'єднання і порівняння колекцій, що мають різні критерії сортування. Можна навіть розробляти ієрархії функціональних об'єктів, щоб, наприклад, створювати конкретні варіанти загального критерію.

3. **Функціональні об'єкти, як правило, працюють швидше, ніж звичайні функції.** Концепція шаблонів звичайно дозволяє поліпшити оптимізацію, оскільки більше деталей визначається на етапі компіляції. Таким чином, передача функціональних об'єктів замість звичайних функцій часто підвищує продуктивність програми.

У частині, що залишилася, рассматриваются розглядаються приклади, що демонструють перевагу функціональних об'єктів над звичайними функціями. У главі 10, присвяченої винятково функціональним об'єктам, приведено більше прикладів і подробиць. Зокрема, показано, як покористуватися з можливості передавати функціональне поводження як шаблонний параметр.

Допустимо, що ми хочемо додати визначене значення до всіх елементів колекції. Якщо це значення відоме на етапі компіляції, то можна використовувати звичайну функцію.

```
void add10 (int& elem)
{
   elem += 10;
}

void f1()
{
   vector<int> coll;
   ...

   for_each (coll.begin(), coll.end(), // діапазон
             add10);                   // операція
 }
```

Якщо необхідно додавати різні значення, відомі на етапі компіляції, то можна використовувати шаблон.

```
template <int theValue>
void add (int& elem)
{
   elem += theValue;
}

void f1()
{
   vector<int> coll;
   ...

   for_each (coll.begin(), coll.end(), // діапазон
```

```
            add<10>);                  // операція
 }
```

Якщо значення додається на етапі виконання програми, ситуація ускладнюється. У цьому випадку необхідно передавати значення у функцію до того, як вона буде викликана. Як правило, для цього використовується глобальна змінна, котра використовується як функція, що викликає алгоритм, так і як функція, яка викликається алгоритмом, що додає значення. Це поганий стиль програмування.

Якщо така функція буде потрібна двічі, щоб додати два різних значення, і обоє цих значення повинні додаватися на етапі виконання програми, однією звичайною функцією вже не обійтися. Доведеться або передати дескриптор, або написати дві різні функції. Ви копіювали коли-небудь функцію, що містить статичну перемінну для збереження її стану, просто тому, що вам знадобилася точно така ж функція з іншим станом у той же момент часу? Ця проблема відноситься до тієї ж категорії.

Працюючи з функціональними об'єктами, можна написати більш інтелектуальну функцію, що веде себе бажаним образом. Оскільки вона може мати стан, об'єкт може бути ініціалізований правильним значенням. Розглянемо закінчений приклад:

```
// stl/add1.cpp

 #include <list>
 #include <algorithm>
 #include <iostream>
 #include "print.hpp"
 using namespace std;

 // функціональний об'єкт, що додає значення, отримане при ініціалізації
 class AddValue {
   private:
      int theValue; // значення, що
   додається, public
: // конструктор ініціалізує значення, що
      додається, AddValue(int v) : theValue(v)
   {

   } // "виклик функції" для елемента, до якого додається
   значення void operator() (int& elem) const
      { elem += theValue
;
 };
```

```
int main()
{
  list<int> coll;

  // вставляємо елементи від 1 до 9
  for (int i=1; i<=9; ++i) {
       coll.push_back(i);
  }

  PRINT_ELEMENTS(coll,"ініціалізований: ");

  // додаємо значення 10 до кожного елемента
  for_each (coll.begin(), coll.end(), // діапазон
            AddValue(10));            // операція

  PRINT_ELEMENTS(coll,"після додавання 10: ");

  // додаємо значення першого елемента до кожного елемента
  for_each (coll.begin(), coll.end(), // діапазон
            AddValue(*coll.begin())); // операція

  PRINT_ELEMENTS(coll,"після додавання першого елемента: ");
}
```

Після ініціалізації колекція містить значення від `1` до `9`:

```
ініціалізований: 1 2 3 4 5 6 7 8 9
```

Перший виклик алгоритму `for_each()` додає `10` до кожного значення:

```
for_each (coll.begin(), coll.end(), // діапазон
          AddValue(10));            // операція
```

Тут вираз `AddValue(10)` створює об'єкт типу `AddValue`, ініціалізований значенням `10`. Конструктор `AddValue` зберігає це значення як член `theValue`. В алгоритмі `for_each()` викликається операція `()` для кожного елемента колекції `coll`. Цей виклик операції `()` відноситься до переданого тимчасового функціонального об'єкта типу `AddValue`. Поточний елемент передається як аргумент.

Функціональний об'єкт додає своє значення `10` до кожного елемента. Після цього елементи приймають наступні значення:

```
після додавання 10:11 12 13 14 15 16 17 18 21
```

Другий виклик алгоритму `for_each()` використовує ту ж саму функціональну можливість для додавання значення першого елемента до всіх елементів колекції. Цей

виклик ініціалізує тимчасовий функціональний об'єкт типу `AddValue` першим елементом колекції.

```
AddValue(*coll.begin())
```

Результат роботи програми набуває такого вигляду:

```
після додавання першого елемента: 22 23 24 25 26 27 28 29 30
```

Використовуючи цей прийом, можна вирішити проблему за допомогою двох різних функціональних об'єктів, що мають різні стани в той самий момент часу. Наприклад, можна просто оголосити функціональні об'єкти і використовувати їх незалежно один від одного.

```
AddValue addx(x); // функціональний об'єкт, що додає значення x
AddValue addy(y); // функціональний об'єкт, що додає значення y
for_each (coll.begin(),coll.end(), // додаємо значення x до кожного елемента
          addx);
...
for_each (coll.begin(),coll.end(), // додаємо значення y до кожного елемента
          addy);
...
for_each (coll.begin(),coll.end(), // додаємо значення x до кожного елемента
          addx);
```

Аналогічно можна передбачити додаткову функція-член для чи запиту зміни стану функціонального об'єкта протягом терміну його існування.

Відзначимо, що для деяких алгоритмів стандартна бібліотека мови не регламентує, як часто функціональні об'єкти можуть викликатися для кожного елемента, і може так статися, що елементам будуть передаватися різні копії функціонального об'єкта. Якщо функціональні об'єкти використовуються як предикати, це може викликати неприємні наслідки.

22.2. Стандартні функціональні об'єкти

Стандартна бібліотека мови C++ містить декілька стандартних функціональних об'єктів, що виконують основні операції. Завдяки їм у визначених ситуаціях можна обійтися без створення власних функціональних об'єктів. Типовим прикладом є функціональний об'єкт, використовуваний як критерій сортування. Критерій сортування за допомогою операції < являє собою стандартний функціональний об'єкт `less<>`. Таким чином, оголошення

```
set<int> coll;
```

розвгортається в оголошення

```
set<int,less<int>> coll; // сортуємо елементи за допомогою операції <
```

Тепер легко упорядкувати елементи в зворотному порядку.

```
set<int,greater<int>> coll; // сортуємо елементи за допомогою операції >
```

Іншим прикладом застосування стандартних функціональних об'єктів є алгоритми. Розглянемо наступну програму:

```
// stl/fo1.cpp

 #include <deque>
 #include <algorithm>
 #include <functional>
 #include <iostream>
 #include "print.hpp"
 using namespace std;

 int main()
 {
   deque<int> coll = { 1, 2, 3, 5, 7, 11, 13, 17, 21 };

   PRINT_ELEMENTS(coll,"ініціалізація: ");

   // змінюємо знаки всіх значень у колекції coll на протилежний
   transform (coll.cbegin(),coll.cend(), // джерело
              coll.begin(),              // призначення
              negate<int>());            // операція
   PRINT_ELEMENTS(coll,"зміна знака: ");

   // зводимо всі значення в колекції coll у квадрат
   transform (coll.cbegin(),coll.cend(), // перше джерело
              coll.cbegin(),             // друге джерело
              coll.begin(),              // призначення
              multiplies<int>());        // операція
   PRINT_ELEMENTS(coll,"у квадраті: ");
 }
```

Спочатку в програму включається заголовок для стандартних об'єктів-функцій: `<functional>`

```
#include <functional>
```

Потім два стандартних функціональних об'єкти використовуються для заміни знака і зведення в квадрат елементів колекції `coll`. В фрагменті

```
transform (coll.cbegin(), coll.cend(), // джерело
           coll.begin(),               // призначення
           negate<int>());             // операція
```

вираз

```
negate<int>()
```

створює об'єкт-функцію стандартного шаблонного класу `negate<>`, що просто повертає елемент типу `int`, що має протилежний знак стосовно того елемента, для якого він був викликаний. Алгоритм `transform()` використовує цю операцію для перетворення всіх елементів першої колекції в другу. Якщо діапазони джерела і призначення збігаються, як у даному випадку, що повертаються елементи з протилежним знаком заміняють вихідні елементи. Таким чином, даний оператор змінює на протилежні знаки всіх елементів колекції. Аналогічно функціональний об'єкт `multiplies` використовується для зведення в квадрат всіх елементів колекції `coll`.

```
transform (coll.cbegin(), coll.cend(), // перше джерело
           coll.cbegin(),              // друге джерело
           coll.begin(),               // призначення
           multiplies<int>());         // операція
```

Тут інша форма алгоритму `transform()` поєднує елементи двох колекцій за допомогою заданої операції і записує результат у третю колекцію. Як і раніше, усі колекції збігаються, тому кожен елемент збільшується на себе, а результати заміняють старі значення. Отже, програма виводить на екран наступні рядки:

```
ініціалізація: 1 2 3 5 7 11 13 17 21
зміна знака:  -1 -2 -3 -5 -7 -11 -13 -17 -21
у квадраті:    1 4 9 25 49 121 169 289 361
```

22.3. Зв’язувачі

Для об'єднання функціональних об'єктів з іншими значеннями і виконання інших специфічних задач використовуються *функціональні адаптери*, чи зв’чзувачі (binders). Розглянемо закінчений приклад.

```
// stl/bind1.cpp

 #include <set>
 #include <deque>
 #include <algorithm>
 #include <iterator>
 #include <functional>
 #include <iostream>
 #include "print.hpp"
 using namespace std;
 using namespace std::placeholders;

 int main()
 {
   set<int,greater<int>> coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
   deque<int> coll2;

   // Примітка: завдяки критерію сортування greater<>()
   // елементи слідують у зворотному порядку:
   PRINT_ELEMENTS(coll1,"ініціалізація: ");

   // перетворюємо всі елементи в колекцію coll2,
   // множачи їх на 10
   transform (coll1.cbegin(),coll1.cend(),    // джерело
              back_inserter(coll2),           // призначення
              bind(multiplies<int>(),_1,10)); // операція
   PRINT_ELEMENTS(coll2,"перетворення: ");

   // заменяем значення, рівне 70, значенням, рівним 42
   replace_if (coll2.begin(),coll2.end(),     // діапазон
               bind(equal_to<int>(),_1,70),   // діапазон заміни
               42);                           // нове значення
   PRINT_ELEMENTS(coll2,"заміна: ");

   // видаляємо всі елементи зі значеннями від 50 до 80
   coll2.erase(remove_if(coll2.begin(),coll2.end(),
               bind(logical_and<bool>(),
               bind(greater_equal<int>(),_1,50),
               bind(less_equal<int>(),_1,80))),
               coll2.end());
   PRINT_ELEMENTS(coll2,"видалення: ");
 }
```

Тут оператор

```
transform (coll1.cbegin(),coll1.cend(),        // джерело
           back_inserter(coll2)                // призначення
           bind(multiplies<int>(),_1,10));     // операція
```

перетворює всі елементи колекції `coll1` у колекцію `coll2` (вставляючи їх), множачи кожен елемент на `10`. Для визначення відповідної операції використовується функціональний адаптер `bind()`, що дозволяє створювати високорівневі функціональні об'єкти з низькорівневих функціональних об'єктів і шаблонних заготовок, що представляють собою числові ідентифікатори, що починаються із символу підкреслення. Оператор

```
bind(multiplies<int>(),_1,10)
```

визначає функціональний об'єкт, що множить перший переданий аргумент на `10`.

Аналогічний функціональний об'єкт можна було б використовувати для множення на 10 всіх елементів. Наприклад, наступні інструкції записують у стандартний потік виводу число 990:

```
auto f = bind(multiplies<int>(),_1,10);
 cout << f(99) << endl;
```

Цей функціональний об'єкт передається алгоритму transform(), що очікує як четвертий аргумент операцію, що одержує один аргумент, а саме поточний елемент. Згодом алгоритм transform() виконує операцію "помножити на 10" для кожного елемента і вставляє результат у колекцію coll2, тобто після всіх елементів колекції coll2 слідують усі значення колекції coll1, помножені на 10.

Аналогічно у виклику

```
replace_if (coll2.begin(),coll2.end(),    // діапазон
            bind(equal_to<int>(),_1,70),  // критерій заміни
            42);                          // нове значення
```

як критерій для вибору елемента, що повинний бути замінений числом 42, використовується наступний функціональний адаптер.

```
bind(equal_to<int>(),_1,70)
```

Тут функціональний адаптер bind() викликає бінарний предикат equal_to, якому як перший параметр передається перший аргумент, а як другий параметр — число 70. Таким чином, функціональний об'єкт, визначений за допомогою зв'язувача bind(), повертає true, якщо переданий аргумент (елемент колекції coll2) дорівнює 70. У результаті оператор заміняє всі значення, що дорівнюють 70, значенням 42.

Останній приклад використовує комбінацію зв'язувачів, де оператор

```
bind(logical_and<bool>(),
     bind(greater_equal<int>(),_1,50),
     bind(less_equal<int>(),_1,80))
```

задає для параметра x унарний предикат "x>=50&&x<=80." Цей приклад демонструє можливість використання вкладених зв'язувачів bind() для опису більш складних предикатів і функціональних об'єктів. У даному випадку алгоритм remove_if() використовує функціональний об'єкт для видалення з колекції всіх значень, що лежать у діапазоні від 50 до 80. Фактично алгоритм remove_if() лише змінює порядок і повертає новий кінець діапазону, а функція-член coll2.erase() видаляє "вилучені" елементи з колекції coll2.

Результат роботи програми виглядає так:

```
ініціалізація:  9 8 7 6 5 4 3 2 1
перетворення: 90 80 70 60 50 40 30 20 10
заміна:          90 80 42 60 50 40 30 20 10
видалення:        90 42 40 30 20 10
```

Відзначимо, що заповнювачі мають свій простір імен: `std::placeholders`. З цієї причини в початок програми уставлена відповідна директива, що дозволяє задавати в якості першого чи другого параметра зв'язувача заповнювачі `_1` чи `_2`. Без використання цієї директиви зв'язувачі довелося б визначати в такий спосіб:

```
std::bind(std::logical_and<bool>(),
          std::bind(std::greater_equal<int>(),std::placeholders::_1,50),
          std::bind(std::less_equal<int>(),std::placeholders::_1,80))
```

Цей вид програмування називається *функціональною композицією* (functional composition). Цікаво, що всі ці функціональні об'єкти звичайно з'являються що вбудовуються. Таким чином, ми використовуємо функціональну абстракцію і при цьому одержуємо високу продуктивність.

Існують і інші способи визначення функціональних об'єктів. Наприклад, для виклику функції-члена для кожного елемента колекції можна використовувати виклик

```
for_each (coll.cbegin(), coll.cend(), // діапазон
          bind(&Person::save,_1));    // операція: Person::save(elem)
```

Функціональний об'єкт `bind` зв'язує зазначену функція-член, щоб вона викликалася для кожного елемента, що передається за допомогою заповнювача `_1`. Таким чином, для кожного елемента колекції `coll` викликається функція-член `save()` із класу `Person`. Зрозуміло, усе це працює тільки за умови, що елементи мають тип `Person` чи тип, похідний від типу `Person`.

До появи документа TR1 для функціональної композиції використовувалися інші зв'язувачі й адаптери: `bind1st()`, `bind2nd()`, `ptr_fun()`, `mem_fun()` і `mem_fun_ref()`, що у стандарті C++11 оголошені застарілими.

23.4. Функціональні об'єкти і зв'язувачі проти лямбда-функції

Лямбда-функції — це різновид неявно визначеного функціонального об'єкта. Лямбда-функції забезпечують більш інтуїтивний підхід до визначення функціонального поводження алгоритмів з бібліотеки STL. Крім того, лямбда-функції працюють швидше функціональних об'єктів.

Проте в лямбда-функцій є кілька недоліків.

- У такого функціонального об’єкта немає схованого внутрішнього стану. Замість цього всі дані, що визначають його стан, визначаються функцією, що викликається, і передаються у виді захоплення.
- Зникає перевага часткового опису функціонального поводження, що потрібно в різних місцях. Ви можете визначити лямбда-функцію, а потім привласнити її об'єкту `auto`, але читабельність такого визначення досить сумнівна.

Література

Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. — М.: Вильямс, 2005. — глава 6.