Лекція 21

Алгоритми

21.1. Поняття про алгоритм

У бібліотеці STL передбачено кілька стандартних алгоритмів для обробки елементів колекцій. Ці алгоритми забезпечують основні операції, такі як пошук, сортування, копіювання, переупорядкування, модифікація і чисельні розрахунки.

Алгоритми не є членами контейнерних класів. Вони являють собою глобальні функції, що працюють з ітераторами. Ця обставина забезпечує важливу перевагу: можна розробляти алгоритми для всіх контейнерних типів відразу, а не для кожного окремо. Алгоритм може навіть працювати з елементами контейнерів різних типів. Крім того, можна використовувати алгоритми для контейнерних типів, визначених користувачем. У результаті ця концепція дозволяє зменшити розмір коду і збільшити міць і гнучкість бібліотеки.

Відзначимо, що ця концепція належить не до об’єктно-орієнтованого, а до функціонального програмування. Замість об'єднання даних і операцій, як це прийнято в об'єктно-орієнтованому програмуванні, вони розділяються на окремі частини, що взаємодіють за допомогою визначеного інтерфейсу. Утім, ця концепція має і недоліки: по-перше, її використання не є інтуїтивним, по-друге, деякі сполучення структур даних і алгоритмів можуть виявитися непрацездатними. Можлива ще більш погана ситуація, коли деяке сполучення контейнерного типу й алгоритму може виявитися можливим, але шкідливим (наприклад, знижувати швидкодію програми). Таким чином, для того щоб витягти максимальну користь, необхідно добре вивчити бібліотеку STL із усіма її достоїнствами і недоліками. Розглянемо приклади і подробиці, що відносяться до алгоритмів. Почнемо з простого прикладу використання алгоритмів бібліотеки STL.

```
// stl/algo1.cpp

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

int main()
{
  // створюємо вектор елементів від 1 до 6 у довільному порядку
  vector<int> coll = { 2, 5, 4, 1, 6, 3 };
  // знаходимо і виводимо на екран мінімальний і максимальний елементи
```

```
   auto minpos = min_element(coll.cbegin(),coll.cend());
   cout << "min: " << *minpos << endl;
   auto maxpos = max_element(coll.cbegin(),coll.cend());
   cout << "max: " << *maxpos << endl;

   // сортуємо всі елементи
   sort (coll.begin(), coll.end());

   // знаходимо перший елемент зі значенням 3
   // - не використовуємо cbegin()/cend(), тому що пізніше ми
   // модифікуємо елементи, на которые буде посилатися pos3
   auto pos3 = find (coll.begin(), coll.end(), // діапазон
                     3);                       // значення

   // змінюємо на зворотний порядок елементів, починаючи з рівного 3
   // і до кінця
   reverse (pos3, coll.end());

   // виводимо на екран всі елементи
   for (auto elem : coll) {
      cout << elem << ' ';
   }
   cout << endl;
}
```

Щоб мати можливість викликати алгоритми, у програму слід включити заголовний файл <algorithm> (для деяких алгоритмів необхідні спеціальні заголовні файли, див. раздел 11.1).

```
#include <algorithm>
```

Перші два алгоритми, `min_element()` і `max_element()`, викликаються з двома параметрами, що визначають діапазон оброблюваних елементів. Для обробки всіх елементів контейнера використовуються функції `cbegin()` і `cend()` чи `begin()` і `end()` відповідно. Обидва алгоритми повертають ітератор, що посилається на позицію першого знайденого елемента. Таким чином, в операторі

```
auto minpos = min_element(coll.cbegin(),coll.cend());
```

алгоритм `min_element()` повертає позицію мінімального елемента. (Якщо мінімальних елементів несколько, алгоритм повертає перший з них.) Наступний оператор виводить на екран елемент, на який посилається ітератор:

```
cout << "min: " << *minpos << endl;
```

Зрозуміло, ці операції можна сполучити:

```
cout << *min_element(coll.cbegin(),coll.cend()) << endl;
```

Наступний алгоритм, `sort()`, як слідує з його назви, сортує діапазон, визначений двома аргументами. як зазвичай, йому можна передати необов'язковий критерій сортування. За замовчуванням як критерій сортування використовується операція <. Таким чином, у даному прикладі всі елементи контейнера сортуються в зростаючому порядку.

```
sort (coll.begin(), coll.end());
```

У результаті вектор містить елементи, що слідують у зазначеному нижче порядку.

```
1 2 3 4 5 6
```

Зверніть увагу на те, що тут ми не можемо використовувати функції `cbegin()` і `cend()`, тому що алгоритм `sort()` змінює значення елементів, що неможливо для ітераторів типу `const_iterator`.

Алгоритм `find()` шукає значення в заданому діапазоні. У нашому прикладі цей алгоритм шукає перший елемент, рівний `3` у всьому контейнері.

```
auto pos3 = find (coll.begin(), coll.end(), // діапазон
                  3);                       // значення
```

Якщо алгоритм `find()` успішно виконаний, він повертає позицію ітератора, установленого на знайдений елемент. Якщо пошук завершився невдало, алгоритм повертає кінець діапазону, переданий як другий аргумент. У нашому випадку це позамежний ітератор контейнера `coll`. Значення `3` знайдене як третій елемент, тому ітератор `pos3` посилається на третій елемент контейнера `coll`.

Останній алгоритм — `reverse()`, викликаний у прикладі, змінює на зворотний порядок проходження елементів у заданому діапазоні. У даному випадку як аргументи в алгоритм `find()` передається ітератор, установлений на третій елемент, і позамежний ітератор:

```
reverse (pos3, coll.end());
```

Цей виклик змінює на зворотний порядок проходження елементів, розташованих у діапазоні від третього до останнього. Оскільки ця операція є модифікацією, ми повинні використовувати неконстантний ітератор. Саме тому ми викликали алгоритм `find()` з функціями `begin()` і `end()`, а не `cbegin()` і `cend()`. У протидежному випадку ітератор `pos3` був би константним, і це привело б до помилки при його передачі алгоритму `reverse()`.

Результати роботи програми виглядають у такий спосіб:

```
min: 1
 max: 6
 1 2 6 5 4 3
```

Відзначимо, що в цьому прикладі використано декілька засобів зі стандарту C++11. Якщо платформа не підтримує стандарт C++11, програма може виглядати в такий спосіб:

```
// stl/algo1old.cpp

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

 int main()
 {
   // створюємо вектор елементів від 1 до 6 у довільному порядку
   vector<int> coll;
   coll.push_back(2);
   coll.push_back(5);
   coll.push_back(4);
   coll.push_back(1);
   coll.push_back(6);
   coll.push_back(3);

   // знаходимо і виводимо на екран мінімальний і максимальний елементи
   vector<int>::const_iterator minpos = min_element(coll.begin(),
                                                    coll.end());
   cout << "min: " << *minpos << endl;

   vector<int>::const_iterator maxpos = max_element(coll.begin(),
                                                    coll.end());
   cout << "max: " << *maxpos << endl;

   // сортуємо всі елементи
   sort (coll.begin(), coll.end());

   // знаходимо перший елемент, рівний 3
   vector<int>::iterator pos3;
   pos3 = find (coll.begin(), coll.end(), // діапазон
               3);                        // значення

   // змінюємо на зворотний порядок елементів, починаючи з рівного 3
   // і до кінця
   reverse (pos3, coll.end());
```

```
    // виводимо на екран всі елементи
    vector<int>::const_iterator pos;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
         cout << *pos << ' ';
    }
    cout << endl;
}
```

Відмінності полягають у наступному.

- Для ініціалізації вектора неможливо використовувати список ініціалізації.
- Функції-члени `cbegin()` і `cend()` не передбачені, тому замість них приходиться використовувати функції `begin()` і `end()`. Тем можна використовувати константні ітератори.
- Замість ключового слова `auto` необходимо завжди явно оголошувати ітератори.
- Замість діапазонних циклів `for` для виводу на екран елементів колекції необхідно використовувати ітератори.

21.2. Діапазони

Всі алгоритми обробляють один чи декілька *діапазонів*. Ці діапазони можуть, але не зобов'язані містити всі елементи контейнера. Отже, для того щоб обробляти підмножину елементів контейнера, необхідно передавати початок і кінець діапазону в якості двох різних аргументів, а не всю колекцію як один аргумент.

Цей інтерфейс є гнучким і в той же час небезпечним. Функція, що викликається, зобов'язана гарантувати, що перший і другий аргументи визначають *коректний* діапазон. Діапазон вважається коректним, якщо кінець діапазону *досяжний* з його початку шляхом перебору елементів. Отже, програміст повинний самостійно гарантувати, що обоє ітератори належать тому самому контейнеру і що початок діапазону передує його кінцю. У протилежному випадку наслідку будуть непередбаченими, зокрема, можуть виникнути нескінченні цикли і звертання до заборонених областей пам’яті. У цьому відношенні ітератори так само небезпечні, як звичайні вказівники. Однак невизначене поводження також означає, що реалізація бібліотеки STL може вільно розпізнавати такі види помилок і відповідним чином обробляти їх. Нижче буде показано, що гарантувати коректність *діапазонів* не так просто, як здається.

Кожен алгоритм обробляє *напіввідчинений* діапазон. Таким чином, діапазон визначений так, що він містить початок, але не містить кінець. Цю концепцію часто описують за допомогою традиційних математичних позначень:

[*begin,end*)

чи

[*begin,end*[

Ми використовуємо перший вид позначень.

Перевага концепції напіввідчиненого діапазону полягає в тім, що вона проста і запобігає роботі з порожніми колекціями. Однак у неї є і недоліки. Розглянемо наступний приклад:

```
// stl/find1.cpp

 #include <algorithm>
 #include <list>
 #include <iostream>
 using namespace std;

 int main()
 {
   list<int> coll;

   // вставляємо елементи від 20 до 40
   for (int i=20; i<=40; ++i) {
        coll.push_back(i);
   }

   // знаходимо позицію елемента, рівного 3
   // - його тут ні, тому ітератор pos3 дорівнює coll.end()
   auto pos3 = find (coll.begin(), coll.end(), // діапазон
                     3);                       // значення

   // змінюємо на зворотний порядок проходження елементів
   // від знайденого до кінця контейнера
   // - оскільки pos3 дорівнює coll.end(), зворотний порядок
   // встановлюється в порожньому контейнері
   reverse (pos3, coll.end());

   // знаходимо позиції значень 25 і 35
   list<int>::iterator pos25, pos35;
   pos25 = find (coll.begin(), coll.end(), // діапазон
                 25);                      // значення
   pos35 = find (coll.begin(), coll.end(), // діапазон
                 35);                      // значення
```

```
   // виводимо на екран максимальний елемент заданого діапазону
   // - примітка: включаючи pos25, але крім pos35
   cout << "max: " << *max_element (pos25, pos35) << endl;

   // обробляємо елементи, включаючи останню позицію
    cout << "max: " << *max_element (pos25, ++pos35) << endl;
}
```

У цьому прикладі колекція ініціалізується цілими числами від `20` до `40`. Коли пошук елемента, рівного `3`, закінчується невдачею, алгоритм `find()` повертає кінець обробленого діапазону (у нашому прикладі — `coll.end()`) і привласнюємо його ітератору `pos3`. Використання цього значення в якості початку діапазону в наступному виклику алгоритму `reverse()` не створює ніяких проблем, тому що він еквівалентний наступному виклику:

```
reverse (coll.end(), coll.end());
```

Це просте звертання порядку елементів порожнього діапазону. Таким чином, ця операція нічого не робить (так звана “порожня операція”).

Однак якщо алгоритм `find()` використовується для пошуку першого й останнього елементів підмножини, необхідно передати ці ітератори у вигляді діапазону, що не містить останній елемент. Отже, перший виклик

```
max_element (pos25, pos35)
```

знайде число `34`, але не `35`:

```
max: 34
```

Для роботи з останнім елементом необхідно передати позицію, що слідує за останнім елементом:

```
max_element (pos25, ++pos35)
```

Це приведе до правильного результату:

```
max: 35
```

Відзначимо, що в цьому прикладі як контейнер використовується список. Таким чином, для того щоб одержати позицію, що слідує за ітератором `pos35`, необхідно використовувати операцію `++`. При роботі з ітератором довільного доступу, наприклад ітераторами вектора чи дека, можна також використовувати вираз `pos35 + 1`, оскільки ітератори довільного доступу допускають *арифметику* ітераторів.

Зрозуміло, для пошуку елементів у підінтервалі можна використовувати ітератори `pos25` і `pos35`. І знову, щоб включити `pos35` у діапазон пошуку, необхідно передати алгоритму позицію, що слідує за этим ітератором. Розглянемо приклад:

```
// збільшуємо pos35 для пошуку з урахуванням його значення
 ++pos35;
 pos30 = find(pos25,pos35, // діапазон
             30);          // значення
 if (pos30 == pos35) {
    cout << "30 не належить підінтервалу" << endl;
 }
 else {
    cout << "30 належить підінтервалу" << endl;
 }
```

Усі приклади в цьому розділі працюють тільки тому, що нам відомо, що ітератор `pos25` передує ітератору `pos35`. У протилежному випадку діапазон [`pos25`,`pos35`) не був би коректним. Якщо заздалегідь невідомо, який елемент передує іншому, ситуація ускладнюється і може виникнути невизначене поводження.

Допустимо, що нам невідомо, ли передує елемент, рівний `25`, елементу, рівному `35`. Може навіть виявитися, що одне чи обоє цих значення в колекції відсутні. Використовуючи ітератор довільного доступу, для перевірки цієї умови можна викликати операцію <.

```
if (pos25 < pos35) {
   // тільки [pos25,pos35) є коректним діапазоном
   ...
}
 else if (pos35 < pos25) {
   // тільки [pos35,pos25) є коректним діапазоном
   ...
}
 else {
   // обоє ітератора рівні один одному, значить обох повинні бути рівні end()
   ...
}
```

Однак при роботі з ітераторами, що не є ітераторами довільного доступу, не існує простого і швидкого способу виявлення попереднього ітератора. Можна лише виконувати пошук першого ітератора в діапазоні від початку контейнера до другого ітератора чи в діапазоні від другого ітератора до кінця контейнера. У цьому випадку алгоритм можна змінити в такий спосіб: замість пошуку обох значень у всьому контейнері варто спробувати з'ясувати, яке значення виявляється першим:

```
pos25 = find (coll.begin(), coll.end(), // діапазон
              25);                      // значення
 pos35 = find (coll.begin(), pos25,      // діапазон
              35);                      // значення
```

```
 if (pos25 != coll.end() && pos35 != pos25) {
 // ітератор pos35 розташований перед ітератором pos25
 // тому коректним є тільки діапазон [pos35,pos25)
...
}
 else {
   pos35 = find (pos25, coll.end(), // діапазон
                 35);               // значення
   if (pos35 != coll.end()) {
       // ітератор pos25 передує ітератору pos35
       // тому коректним є тільки діапазон [pos25,pos35)
       ...
   }
   else {
       // 25 і/чи 35 не знайдені
       ...
   }
 }
```

На противагу попередньої версії ми не шукаємо `35` у всьому контейнері `coll`. Замість цього ми спочатку шукаємо його в діапазоні від початку до ітератора `pos25`. Потім, якщо він не знайдений, шукаємо його серед елементів, що слідують за ітератором `pos25`. У результаті нам відомо, яка з позицій ітератора є першої і який діапазон вважається коректним.

Ця реалізація не дуже ефективна. Більш ефективний спосіб знайти перший елемент, рівний `25` чи `35`, — шукати їх непосредственно. Це можна зробити за допомогою алгоритму `find_if()` чи лямбда-виразу, визначивши критерій, що застосовується до кожного елемента контейнера `coll`.

```
pos = find_if (coll.begin(), coll.end(), // діапазон
               [] (int i) {               // критерій
                   return i == 25 || i == 35;
               });
if (pos == coll.end()) {
   // елемент, рівний 25 чи 35, не виявлений
   ...
}
else if (*pos == 25) {
   // елемент, рівний 25, є першим
   pos25 = pos;
   pos35 = find (++pos, coll.end(), // діапазон
                 35);               // значення
```

```
   ...
}
else {
   // елемент, рівний 35, є першим
   pos35 = pos;
   pos25 = find (++pos, coll.end(), // діапазон
                 25);               // значення
   ...
}
```

Тут спеціальний лямбда-вираз

```
[] (int i) {
 return i == 25 || i == 35;
 }
```

використовується як критерій, що дозволяє виконувати пошук першого елемента, що дорівнює `25` чи `35`.

21.3. Обробка декількох діапазонів

Деякі алгоритми працюють з декількома діапазонами. У цьому випадку зазвичай необхідно визначити початок і кінець тільки першого діапазону. Для всіх інших *діапазонів* необхідно передати лише їх початок. Кінці інших *діапазонів* визначаються кількістю елементів у першому діапазоні. Наприклад, такий виклик алгоритму `equal()` порівнює всі елементи колекції `coll1` з елементами колекції `coll2`, починаючи з першого елемента:

```
if (equal (coll1.begin(), coll1.end(), // перший діапазон
           coll2.begin())) {           // другий діапазон
   ...
}
```

Таким чином, кількість елементів колекції `coll2`, порівнюваних з елементами колекції `coll1`, задається побічно за допомогою кількості елементів колекції `coll1`. Це приводить до важливого наслідку: **при виклику алгоритмів для декількох діапазонів другий і наступний діапазони повинні мати не менше елементів, чим перший діапазон**. Зокрема, діапазони призначення повинні бути досить великими, щоб алгоритми могли виконати запис.

Розглянемо наступну програму:

```
// stl/copybug.cpp

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

```
int main()
{
  list<int> coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  vector<int> coll2;

  // ПОМИЛКА ПІД ЧАС ВИКОНАННЯ ПРОГРАМИ:
  // - перезапис неіснуючих елементів у діапазоні призначення
  copy (coll1.cbegin(), coll1.cend(), // джерело
        coll2.begin());               // призначення
  ...
}
```

Тут викликається алгоритм `copy()`, що просто копіює всі елементи першого діапазону в діапазон призначення. Як зазвичай, визначаються початок і кінець першого діапазону і лише початок другого. Однак алгоритм виконує перезапис, а не вставку. З цієї причини алгоритм *вимагає*, щоб у діапазоні призначення містилася достатня кількість елементів для перезапису. Якщо місця недостатньо, як у даному випадку, виникає невизначене поводження. На практиці це часто означає, що відбувається перезапис інформації, розташованої за ітератором `coll2.end()`. Якщо пощастить, справа обмежиться збоєм програми, і ви зрозумієте, що зробили щось не так. Утім, можна підстрахуватися за допомогою безпечної версії бібліотеки STL, у якій невизначене поводження приводить до виклику процедури обробки помилок.

Для того щоб уникнути цих помилок, можна 1) зробити так, щоб діапазон призначення мав досить місця для запису, чи 2) використовувати *ітератори вставки*. Спочатку подивимося, як можна змінити діапазон призначення, щоб він мав досить місця для запису.

Для того щоб діапазон призначення став досить великим, можна або відразу задати правильний розмір, або змінити його явно. Обидві ці альтернативи застосовні тільки до деяких послідовних контейнерів (`vector`, `deque`, `list` і `forward_list`). Однак для інших контейнерів це взагалі не проблема, оскільки асоціативні і неупорядковані контейнери не можна використовувати як діапазони призначення в алгоритмах перезапису. Наступна програма демонструє спосіб збільшення розміру контейнера:

```
// stl/copy1.cpp

#include <algorithm>
#include <list>
#include <vector>
```

```
#include <deque>
using namespace std;

int main()
{
  list<int> coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  vector<int> coll2;

  // змінюємо розмір діапазону призначення, щоб він мав
  // достаточно місця для алгоритму перезапису
  coll2.resize (coll1.size());

  // копіюємо елементи з першої колекції в другу
  // - перезаписываем існуючі елементи в діапазон призначення
  copy (coll1.cbegin(), coll1.cend(), // джерело
        coll2.begin());               // призначення

  // створюємо третю колекцію, що має достатній об'єм
  // - початковий розмір передається як параметр
  deque<int> coll3(coll1.size());

  // копіюємо елементи з першої в третю колекцію
  copy (coll1.cbegin(), coll1.cend(), // джерело
        coll3.begin());               // призначення
}
```

Тут функція `resize()` використовується для зміни кількості елементів в існуючому контейнері `coll2`.

```
coll2.resize (coll1.size());
```

Потім колекція `coll3` ініціалізується спеціальним початковим розміром, що забезпечує достатній об'єм для збереження всіх елементів колекції `coll1`.

```
deque<int> coll3(coll1.size());
```

Відзначимо, що зміна й ініціалізація розміру приводять до створення нових елементівів, що ініціалізуються своїм конструктором, заданим за замовчуванням, оскільки їм не передаються ніякі аргументи. Крім того, конструктору і функції `resize()` можна передати додатковий аргумент для ініціалізації нових елементів.

21.6. Користувальницькі узагальнені функції

Бібліотека STL допускає розширення. Це значить, що користувач може писати свої функції й алгоритми для роботи з елементами колекції. Зрозуміло, ці операції можуть бути узагальненими. Однак для оголошення коректного ітератора в цих операціях необхідно

використовувати тип контейнера, що у кожного контейнера свій. Для того щоб полегшити створення узагальнених функцій, кожен контейнерний тип передбачає внутрішні визначення типів. Розглянемо наступним приклад:

```
// stl/print.hpp

 #include <iostream>
 #include <string>

 // PRINT_ELEMENTS()
 // - виводить необов'язковий рядок optstr, за якого слідують
 // - всі елементи колекції coll
 // - в одному рядку, розділені пробілами
 template <typename T>
 inline void PRINT_ELEMENTS (const T& coll,
                             const std::string& optstr="")
 {
   std::cout << optstr;
   for (const auto& elem : coll) {
        std::cout << elem << ' ';
   }
   std::cout << std::endl;
 }
```

У цьому прикладі визначена узагальнена функція, що виводить на екран необов'язковий рядок, за якого слідують всі елементи переданого контейнера.

До появи стандарту C++11 цикл по елементах виглядав у такий спосіб:

```
typename T::const_iterator pos;
 for (pos=coll.begin(); pos!=coll.end(); ++pos) {
     std::cout << *pos << ' ';
 }
```

Тут змінна `pos` оголошена як ітератор переданого контейнера. Відзначимо, що ключове слово `typename` тут є обов'язковим, щоб указати. що `const_iterator` — це тип, а не статичний член типу `T`.

На додаток до типів `iterator` і `const_iterator` контейнери містять інші типи для створення узагальнених функцій. Наприклад, вони надають тип елементів для створення тимчасових копий елементів.

Необов'язковий другий аргумент `PRINT_ELEMENTS` являє собою рядок, використовуваний як префікс перед записуваними елементами. Таким чином, функція `PRINT_ELEMENTS()` дозволяє коментувати результати роботи програми:

```
PRINT_ELEMENTS (coll, "всі елементи: ");
```

Ця функція описана тут тому, що вона буде широко використовуватися в іншій частині книги при виводу елементів контейнера.

21.7. Алгоритми, що модифікують

Дотепер ми розглядали концепцію бібліотеки STL у цілому. Контейнери представляють різні способи керування колекціями даних, Алгоритми виконують операції читання і запису елементів цих колекцій. Ітератори служать посередниками між контейнерами й алгоритмами. Ітератори, надані контейнерами, дозволяють перебирати всі елементи в різному порядку й у різних режимах, наприклад у режимі вставки.

На практиці робота з бібліотекою STL зв'язана з певними обмеженнями, які варто знати. Багато з них стосуються модифікацій. Декілька алгоритмів модифікують цільові діапазони. Зокрема, ці алгоритми можуть видаляти елементи. При цьому виникають визначені аспекти, про які мова йтиме в цьому розділі. Ці аспекти викликають подив і демонструють зворотну сторону концепції STL, що має на увазі поділ контейнерів і алгоритмів для досягнення високої гнучкості.

21.7.1. "Видалення" елементів

Алгоритм `remove()` видаляє елементи з діапазону. Однак застосування цього алгоритму до всіх елементів контейнера приводить до дивних результатів. Розглянемо наступним приклад:

```
// stl/remove1.cpp

 #include <algorithm>
 #include <iterator>
 #include <list>
 #include <iostream>
 using namespace std;

 int main()
 {
   list<int> coll;

   // вставляємо елементи від 6 до 1 і від 1 до 6
   for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
   }
```

```
   // виводимо на екран всі елементи колекції
   cout << "pre:  ";
   copy (coll.cbegin(), coll.cend(),       // джерело
         ostream_iterator<int>(cout," ")); // призначення
   cout << endl;

   // видаляємо всі елементи, рівні 3
   remove (coll.begin(), coll.end(), // діапазон
           3);                       // значення

   // виводимо на екран всі елементи колекції
   cout << "post: ";
   copy (coll.cbegin(), coll.cend(),       // джерело
         ostream_iterator<int>(cout," ")); // призначення
   cout << endl;
}
```

Людина, що поверхово розбирається в бібліотеці STL, при читанні цієї програми могла б очікувати, що з колекції будуть вилучені всі елементи, рівні `3`. Однак результат програми виглядає в такий спосіб:

```
pre:  6 5 4 3 2 1 1 2 3 4 5 6
post: 6 5 4 2 1 1 2 4 5 6 5 6
```

Таким чином, алгоритм `remove()` не змінює кількість елементів у колекції, до якої він застосовується. Функція-член `cend()` повертає старий кінець — як і функція `end()`, — а функція `size()` повертає стару кількість елементів. Однак дещо змінилося: елементи змінили порядок проходження, як якби елементи были вилучені. Кожен елемент, рівний `3`, заміняється наступними елементами. Наприкінці колекції старі елементи не заміняються алгоритмом і залишаються незмінними. Логічно ці елементи більше не належать колекції.

Однак цей алгоритм повертає новий логічний кінець. За допомогою цього алгоритму можна одержати доступ до результуючого діапазону, зменшити розмір чи колекції підрахувати кількість вилучених елементів. Розглянемо наступну модифіковану версію цього приклада:

```
// stl/remove2.cpp

#include <algorithm>
#include <iterator>
#include <list>
#include <iostream>
using namespace std;
```

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

  // вставляємо елементи від 6 до 1 і від 1 до 6
  for (int i=1; i<=6; ++i) {
       coll.push_front(i);
       coll.push_back(i);
  }

  // виводимо на екран всі елементи колекції
  copy (coll.cbegin(), coll.cend(),
        ostream_iterator<int>(cout," "));
  cout << endl;

  // видаляємо всі елементи, рівні 3
  // - зберігаємо новий кінець
  list<int>::iterator end = remove (coll.begin(), coll.end(),
                                    3);

  // виводимо на екран результуючі елементи колекції
  copy (coll.begin(), end,
        ostream_iterator<int>(cout," "));
  cout << endl;

  // виводимо на екран кількість вилучених елементів
  cout << "number of removed elements "
       << distance(end,coll.end()) << endl;

  // удаляем "вилучені" елементи
  coll.erase (end, coll.end());

  // виводимо на екран всі елементи модифікованої колекції
  copy (coll.cbegin(), coll.cend(),
        ostream_iterator<int>(cout," "));
  cout << endl;

}
```

У цій версії значення, що повертається алгоритмом `remove()`, привласнюється ітератору `end`:

```
list<int>::iterator end = remove (coll.begin(), coll.end(), 3);
```

Це новий логічний кінець модифікованої колекції після “видалення” елементів. Значення, що це повертається, можна використовувати як новий кінець для наступних операцій:

```
copy (coll.begin(), end, ostream_iterator<int>(cout," "));
```

Відзначимо, що ми повинні використовувати функцію `begin()`, а не `cbegin()`, тому що ітератор `end` визначено як неконстантний, а початок і кінець діапазону визначені як ітератори однакового типу.

Інша можливість — обчислити кількість “вилучених” елементів, визначивши відстань між “логічним” і реальним кінцем колекції.

```
cout << "number of removed elements: "
     << distance(end,coll.end()) << endl;
```

Тут використовується спеціальна допоміжна функція `distance()` для ітераторів, що повертає відстань між ітераторами. Якби ітератори были ітераторами довільного доступу, можна було б обчислити різниця безпосередньо за допомогою операції `-`. Однак у даному випадку контейнер є списком, тому він передбачає двоспрямовані ітератори. Докладний опис функції `distance()` приведено в розділі 9.3.3.

Якщо дійсно необхідно видалити “вилучені” елементи, тоді потрібно викликати відповідну функція-член контейнера. Для цієї мети контейнери містять функція-член `erase()`, що видаляє весь діапазон, заданий за допомогою аргументів.

```
coll.erase (end, coll.end());
```

Результати роботи всієї програми виглядають у такий спосіб:

```
6 5 4 3 2 1 1 2 3 4 5 6
6 5 4 2 1 1 2 4 5 6
number of removed elements: 2
6 5 4 2 1 1 2 4 5 6
```

Якщо необхідно видалити елементи за допомогою одного оператора, можна виконати інструкцію

```
coll.erase (remove(coll.begin(),coll.end(),3), coll.end());
```

Чому алгоритми самі не викликають функцію `erase()`? Це питання підкреслює ціну, которую приходиться платити за гнучкість бібліотеки STL. Бібліотека STL відокремлює структури даних від алгоритмів, використовуючи ітератори як інтерфейс. Однак ітератори — це абстракція позиції в контейнері. У принципі, ітератори *нічого не знають* про свої контейнери. Таким чином, алгоритми, що використовують ітератори для доступу до елементів контейнера, не можуть викликати його функції-члени.

Ця схема має важливі наслідки, оскільки дозволяє алгоритмам працювати з діапазонами, що відрізняються від “всіх елементів контейнера”. Наприклад, діапазон може

бути підмножиною всіх елементів колекції. Він навіть може бути контейнером, що не має функції-члена `erase()` (прикладом такого контейнера є масив). Отже, для того щоб алгоритм був як можна більш гнучким, бажано не вимагати, щоб ітератор знав про контейнер.

Відзначимо, що фізично видаляти "вилучені" елементи часто не потрібно. У багатьох ситуаціях немає нічого страшного в тім, що замість реального кінця контейнера повертається його логічний кінець. Зокрема, всі алгоритми можна застосовувати, указуючи цей новий логічний кінець.

21.7.2. Робота з асоціативними і неупорядкованими контейнерами

Алгоритми, що модифікують, тобто алгоритми, що видаляють елементи, що змінюють їхній чи порядок їх значення, породжують іншу проблему при їхньому використанні для асоціативних чи неупорядкованих контейнерів: такі контейнери не можна використовувати як цільовий діапазон. Причина проста: при роботі з асоціативними і неупорядкованими контейнерами алгоритми, що модифікують, можуть змінювати значення чи позиції елементів, тим самим порушуючи порядок проходження елементів у контейнері (упорядкований в асоціативних чи контейнерах визначений хеш-функцією у неупорядкованих контейнерах). Для того щоб уникнути порушення внутрішнього порядку, кожен ітератор для асоціативного і неупорядкованого контейнера з'являється як ітератор з константним чи значенням ключем. У результаті маніпуляція елементами в асоціативних чи неупорядкованих контейнерах розпізнається на етапі компіляції як помилка.

Через цю проблему також не можна виконувати алгоритми видалення для асоціативних контейнерів, тому що ці алгоритми неявно маніпулюють елементами. Значення "вилучених" елементів перезаписуються наступними елементами, що не відлучаються.

Виникає питання: як же видалити елементи з асоціативних контейнерів? Відповідь простій: викликати їхні функції-члени! Кожен асоціативний і неупорядкований контейнер містить функцію-член для видалення елементів. Наприклад, для видалення елементів можна викликати функцію-член `erase()`.

```
// stl/remove3.cpp

 #include <set>
 #include <algorithm>
 #include <iterator>
 #include <iostream>
 using namespace std;
```

```
int main()
{
  // неупорядкована множина з елементами від 1 до 9
  set<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  // виводимо на екран всі елементи колекції
  copy (coll.cbegin(), coll.cend(),
        ostream_iterator<int>(cout," "));
  cout << endl;

  // Видаляємо всі елементи, значення яких дорівнює 3,
  // - алгоритм remove() не працює
  // - замість нього застосовується функція-член erase()
  int num = coll.erase(3);

  // виводимо на екран кількість вилучених елементів
  cout << "кількість вилучених елементів: " << num << endl;

  // виводимо на екран всі елементи модифікованої колекції
  copy (coll.cbegin(), coll.cend(),
        ostream_iterator<int>(cout," "));
  cout << endl;
}
```

Відзначимо, що контейнери містять різні функції-члени `erase()`. Кількість вилучених елементів повертає тільки варіант, що видаляє елемент за значенням, заданому як єдиний аргумент. Зрозуміло, якщо дублікати заборонені, те значення, що повертається, може бути рівним `0` 0 чи 1. Зокрема, це стосується контейнерів `set`, `map`, `unordered_set` і `unordered_map`.

Результати роботи програми виглядають у такий спосіб:

```
1 2 3 4 5 6 7 8 9
кількість вилучених елементів: 1
1 2 4 5 6 7 8 9
```

21.7.3. Алгоритми і функції-члени

Навіть якщо алгоритм формально можна застосувати, це може виявитися невдалою ідеєю. Контейнер може мати функції-члени, що забезпечують більш високу продуктивність. Яскравим прикладом є використання функції `remove()` для видалення елементів зі списку. Якщо застосувати функцію `remove()` до елементів списку, алгоритм не буде знати, що він

працює зі списком, і буде працювати з ним як зі звичайним контейнером: переупорядкує елементи, змінивши їх значення. Наприклад, якщо алгоритм видалить перший елемент, те всі інші елементи будуть привласнені їхнім попередникам. Це поводження суперечить основній перевазі списків: можливості вставляти, переміщати і видаляти елементи, модифікуючи зв'язок, а не значення.

Для того щоб не знижувати продуктивність, списки мають спеціальні функції-члени для всіх алгоритмів, що модифікують. Перевагу завжди варто віддавати функціям-членам. Більш того, ці функції-члени дійсно видаляють “вилучені” елементи, як показано в наступному прикладі:

```
// stl/remove4.cpp

 #include <list>
 #include <algorithm>
 using namespace std;

 int main()
 {
   list<int> coll;

   // вставляємо елементи від 6 до 1 і від 1 до 6
   for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
   }

   // видаляємо всі елементи зі значенням 3 (низька продуктивність)
   coll.erase (remove(coll.begin(),coll.end(),
               3),
               coll.end());

   // видаляємо всі елементи зі значенням 4 (висока продуктивність)
   coll.remove (4);
 }
```

Якщо метою є висока швидкодія програми, то завжди варто віддавати перевагу функціям-членам, а не алгоритмам. Проблема лише в тім, щоб знати про існування функції-члена, що забезпечує набагато більш високу продуктивність для конкретного контейнера. Якщо застосувати алгоритм `remove()` до списку, ви не одержите ніякого попередження чи повідомлення про помилку. Однак, якщо в цій ситуації вибрати функцію-член, то при зміні

типу контейнера доведеться змінювати весь код. У главі 11 зазначено, чи існує функція-член, що забезпечує більш високу продуктивність, чим алгоритм.

21.2. Алгоритми і функції-члени

Навіть якщо алгоритм формально можна застосувати, це може виявитися невдалою ідеєю. Контейнер може мати функції-члени, що забезпечують більш високу продуктивність. Яскравим прикладом є використання функції `remove()` для видалення елементів зі списку. Якщо застосувати функцію `remove()` до елементів списку, алгоритм не буде знати, що він працює зі списком, і буде працювати з ним як зі звичайним контейнером: переупорядкує елементи, змінивши їх значення. Наприклад, якщо алгоритм видалить перший елемент, те всі інші елементи будуть привласнені їхнім попередникам. Це поводження суперечить основній перевазі списків: можливості вставляти, переміщати і видаляти елементи, модифікуючи зв'язок, а не значення.

Для того щоб не знижувати продуктивність, списки мають спеціальні функції-члени для всіх алгоритмів, що модифікують. Перевагу завжди варто віддавати функціям-членам. Більш того, ці функції-члени дійсно видаляють “вилучені” елементи, як показано в наступному прикладі:

```
// stl/remove4.cpp

 #include <list>
 #include <algorithm>
 using namespace std;

 int main()
 {
   list<int> coll;

   // вставляємо елементи від 6 до 1 і від 1 до 6
   for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
   }

   // видаляємо всі елементи зі значенням 3 (низька продуктивність)
   coll.erase (remove(coll.begin(),coll.end(),
               3),
               coll.end());

   // видаляємо всі елементи зі значенням 4 (висока продуктивність)
```

```
  coll.remove (4);
}
```

Якщо метою є висока швидкодія програми, то завжди варто віддавати перевагу функціям-членам, а не алгоритмам. Проблема лише в тім, щоб знати про існування функції-члена, що забезпечує набагато більш високу продуктивність для конкретного контейнера. Якщо застосувати алгоритм `remove()` до списку, ви не одержите ніякого попередження чи повідомлення про помилку. Однак, якщо в цій ситуації вибрати функцію-член, то при зміні типу контейнера доведеться змінювати весь код. У главі 11 зазначено, чи існує функція-член, що забезпечує більш високу продуктивність, чим алгоритм.

21.2.1. Використання функцій як аргументів алгоритмів

Найпростіший приклад — алгоритм `for_each()`, що застосовує користувальницьку функцію до кожного елемента в заданому діапазоні. Розглянемо наступний приклад:

```
// stl/foreach1.cpp

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

// функція, що виводить на екран переданий аргумент
void print (int elem)
{
  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
            print);                     // операція
  cout << endl;
}
```

Алгоритм `for_each()` застосовує передану функцію `print()` до кожного елемента в діапазоні [`coll.cbegin()`,`coll.cend()`). Таким чином, одержуємо наступний результат роботи програми:

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

Алгоритми використовують допоміжні функції в різних варіантах: іноді необов'язково, іноді обязательно. Зокрема, допоміжні функції можна використовувати для того, щоб задати критерій пошуку чи сортування для визначення операції при передачі елементів з однієї колекції в іншу.

Розглянемо ще один приклад:

```
// stl/transform1.cpp

 #include <set>
 #include <vector>
 #include <algorithm>
 #include <iterator>
 #include <iostream>
 #include "print.hpp"

 int square (int value)
 {
   return value*value;
 }

 int main()
 {
   std::set<int> coll1;
   std::vector<int> coll2;

   // вставляємо елементи від 1 до 9 у колекцію coll1
   for (int i=1; i<=9; ++i) {
        coll1.insert(i);
   }
   PRINT_ELEMENTS(coll1,"ініціалізовані: ");

   // переносимо кожен елемент із колекції coll1 у колекцію coll2
   // - зводимо в квадрат кожне передане значення
   std::transform (coll1.cbegin(),coll1.cend(), // джерело
                   std::back_inserter(coll2),   // призначення
                   square); // операція
```

```
  PRINT_ELEMENTS(coll2,"у квадраті: ");
}
```

У даному прикладі для піднесення в квадрат кожного елемента колекції `coll1`, переданого в колекцію `colls2`, використовується функція `square()`. Програма виводить на екран наступні рядки:

```
ініціалізовані: 1 2 3 4 5 6 7 8 9
у квадраті:         1 4 9 16 25 36 49 64 81
```

21.2.2. Предикати

Предикат — це особливий різновид допоміжної функції. Предикати повертають булеве значення і часте використовуються для завдання критерію чи сортування пошуку. У залежності від поставленої мети, предикати бувають унарними і бінарними.

Не кожна унарна чи бінарна функція, що повертає булеве значення, є коректним предикатом. Крім того, бібліотека STL вимагає, щоб предикати не мали стану, тобто вони завжди повинні повертати той самий результат для того самого значення. Це виключає застосування функцій, що модифікують свій внутрішній стан при виклику.

Унарні предикати

Унарні предикати перевіряють конкретну властивість єдиного аргументу. Типовим прикладом є функція, використовувана як критерій пошуку для виявлення першого простого числа.

```
// stl/prime1.cpp

#include <list>
#include <algorithm>
#include <iostream>
#include <cstdlib> // для abs()
using namespace std;

// предикат, що повертає результат перевірки, ли є ціле число простим
bool isPrime (int number)
{
  // ігноруємо негативні числа
  number = abs(number);

  // 0 і 1 — не прості числа
  if (number == 0 || number == 1) {
      return false;
  }
```

```
   // знаходимо дільник, ділення на який відбувається без залишку
   int divisor;
   for (divisor = number/2; number%divisor != 0; --divisor) {
        ;
   }

   // якщо не виявлені дільники більше одиниці, значить число просте
   return divisor == 1;
}

int main()
{
   list<int> coll;

   // вставляємо елементи від 24 до 30
   for (int i=24; i<=30; ++i) {
        coll.push_back(i);
   }

   // шукаємо просте число
   auto pos = find_if (coll.cbegin(), coll.cend(), // діапазон
                       isPrime);                   // предикат
   if (pos != coll.end()) {
      // знайдений
      cout << *pos << " — простої число" << endl;
   }
   else {
      // не знайдене
      cout << "простое число не знайдене" << endl;
   }
}
```

У цьому прикладі для пошуку першого елемента в заданому діапазоні, для якого переданий предикат повертає `true`, використовується алгоритм `find_if()`. Тут предикатом є функція `isPrime()`, що перевіряє, чи є число простим. Використовуючи цей предикат, алгоритм повертає перше просте число в заданому діапазоні. Якщо в діапазоні не знайдений жодний елемент, що відповідає предикату, алгоритм повертає кінець діапазону (другий аргумент). Ця умова перевіряється після виконання виклику. Колекція в даному прикладі містить просте число в діапазоні від 24 до 30, тому результат роботи програми виглядає в такий спосіб:

```
29 — просте число
```

Бінарні предикати

Бінарні предикати звичайно порівнюють конкретну властивість двох аргументів. Наприклад, для сортування елементів відповідно до заданого користувача критерієм можна написати просту предикатну функцію. Це може виявитися необхідним у ситуаціях, у яких операція < неприйнятна чи коли потрібний інший критерій.

У наступному прикладі сортуються елементи дека на прізвище й ім'я людини:

```
// stl/sort1.cpp

#include <algorithm>
#include <deque>
#include <string>
#include <iostream>
using namespace std;

class Person {
   public:
      string firstname() const;
      string lastname() const;
      ...
   };

 // бінарний предикат:
 // - повертає результат перевірки, ли передує одна людина іншому
 bool personSortCriterion (const Person& p1, const Person& p2)
 {
   // людина передує іншій людині
   // - якщо його прізвище передує прізвищу іншої людини
   // - якщо прізвища збігаються, а ім'я першої людини
   //   передує імені другого
   return p1.lastname()<p2.lastname() ||
          (p1.lastname()==p2.lastname() &&
          p1.firstname()<p2.firstname());
   }

int main()
{
   deque<Person> coll;
   ...
   sort(coll.begin(),coll.end(), // діапазон
        personSortCriterion);    // критерій сортування
```

```
   ...
}
```

Відзначимо, що критерій сортування можна реалізувати як функціональний об'єкт. Перевагою цього виду реалізації є те, що критерієм є тип, якому можна використовувати, наприклад, для оголошення множина, що використовує цей критерій для сортування елементів.

21.3. Використання виразів-лямбда-виразів

Лямбда-вирази, введені в стандарті C++11, описують функціональне поводження у вигляді вираз чи оператора. Завдяки цьому можна визначати об'єкти, що виражають функціональне поводження, і передавати їх як аргументи в алгоритми для використання як предикати і для інших цілей.

Наприклад, розглянемо такий код:

```
// зводимо всі елементи в куб
 std::transform (coll.begin(), coll.end(), // джерело
                 coll.begin(),             // призначення
                 [](double d) {            // лямбда як функтор
                    return d*d*d;
                 });
```

Вираз

```
[](double d) { return d*d*d; }
```

визначає лямбда-вираз, що представляє функціональний об'єкт, який повертає число типу `double`, піднесене в куб. Як бачимо, це дає можливість задати функціональне поводження, непосредственно передане алгоритму `transform()` при його виклику.

Переваги лямбда-функцій

Використання лямбда-функцій для визначення поводження в рамках бібліотеки STL усуває багато недоліків минулих варіантів реалізації такої функціональності. Допустимо, потрібно знайти перший елемент у колекції, що має значення більше `x` і менше `y`:

```
// stl/lambda1.cpp

 #include <algorithm>
 #include <deque>
 #include <iostream>
 using namespace std;

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

```
   int x = 5;
   int y = 12;
   auto pos = find_if (coll.cbegin(), coll.cend(), // діапазон
                       [=](int i) {                // критерій пошуку
                           return i > x && i < y;
                       });
   cout << "перший елемент >5 і <12: " << *pos << endl;
}
```

Викликаючи алгоритм `find_if()`, ми передаємо відповідний предикат як третій аргумент.

```
auto pos = find_if (coll.cbegin(), coll.cend(),
                    [=](int i) {
                        return i > x && i < y;
                    });
```

Лямбда-функція — це просто функціональний об'єкт, що одержує ціле число `i` і повертає результат перевірки, чи більше число `i` числа `x` і чи менше воно числа `y`.

```
[=](int i) {
    return i > x && i < y;
}
```

Указуючи = як *захоплення* усередині конструкції `[=]`, ми передаємо в тіло лямбда-функції *за значенням* символи, що були коректними в момент оголошення лямбда-функції. Таким чином, у лямбда-функції ми можемо читати змінні `x` і `y`, оголошені у функції `main()`. Якщо використовується конструкція `[&]`, то значення можна передавати по посиланню, так що в тілі лямбда-функції ці значення можна модифікувати.

Порівняємо тепер цей спосіб пошуку першого елемента, що задовольняє умові “`>5` і `<12`”, з іншими підходами, що використовувалися в мові C++ до появи лямбда-функцій.

- На відміну від циклів, написаних вручну,

```
// перший елемент, що задовольняє умові "> x і < y"
 vector<int>::iterator pos;
 for (pos = coll.begin() ; pos != coll.end(); ++pos) {
     if (*pos > x && *pos < y) {
         break; // the loop
     }
 }
```

лямбда-функції дозволяють використовувати стандартні алгоритми й уникати незграбної конструкції `break`.

- На відміну від предикатів, написаних вручну,

```
bool pred (int i)
 {
     return i > x && i < y;
 }
 ...
pos = find_if (coll.begin(), coll.end(), // діапазон
               pred);                    // критерій пошуку
```

лямбда-вирази не створюють проблем, що виникають, коли деталі поводження визначені десь в іншім місці і приходиться з'ясовувати, що саме шукає алгоритм `find_if()`, якщо в програмі немає точних коментарів. Крім того, компілятори мови C++ краще оптимізують лямбда-функції, ніж звичайні функції.

Ще важливіше те, що доступ до перемінних `x` і `y` у цьому сценарії дійсно вкрай незручний. До появи стандарту C++11 звичайно в таких ситуаціях використовувалися функціональні об'єкти, що яскраво демонструють незграбність такого підходу:

```
class Pred
{
   private:
      int x;
      int y;
   public:
      Pred (int xx, int yy) : x(xx), y(yy) {
      }
   bool operator() (int i) const {
      return i > x && i < y;
   }
};
...
 pos = find_if (coll.begin(), coll.end(), // діапазон
               Pred(x,y));               // критерій пошуку
```

- На відміну від застосування зв'язувачів,

```
pos = find_if (coll.begin(), coll.end(), // діапазон
               bind(logical_and<bool>(), // критерій пошуку
                    bind(greater<int>(),_1,x),
                    bind(less<int>(),_1,y)));
```

вираз стає набагато зрозумілішим.

Отже, лямбда-функції вперше забезпечують зручний, зрозумілий, швидкий і технологічний підхід до використання алгоритмів STL.

Використання лямбда-функцій як критерій сортування

Як інший приклад продемонструємо використання лямбда-виразів для визначення критерію сортування вектора елементів типу `Person`.

```
// stl/sort2.cpp

#include <algorithm>
#include <deque>
#include <string>
#include <iostream>
using namespace std;
class Person {
   public:
      string firstname() const;
      string lastname() const;
      ...
};

int main()
{
   deque<Person> coll;
   ...

   // сортуємо вектор типу Persons на прізвище (чи імені):
   sort(coll.begin(),coll.end(),                    // діапазон
        [] (const Person& p1, const Person& p2) { // критерій сортування
            return p1.lastname()<p2.lastname() ||
                   (p1.lastname()==p2.lastname() &&
                    p1.firstname()<p2.firstname());
        });
   ...
 }
```

Обмеження лямбда-функцій

Тем лямбда-функції не завжди є найкращим засобом. Розглянемо, наприклад, використання лямбда-функції для визначення критерію сортування для асоціативних масивів:

```
auto cmp = [] (const Person& p1, const Person& p2) {
               return p1.lastname()<p2.lastname() ||
                      (p1.lastname()==p2.lastname()&&
                       p1.firstname()<p2.firstname());
           };
```

```
...
 std::set<Person,decltype(cmp)> coll(cmp);
```

Оскільки в оголошенні об'єкта класу `set` необхідно визначити тип лямбда-функції, приходиться використовувати операцію `decltype`, що повертає тип лямбда-об'єкта, наприклад `cmp`. Відзначимо, що необходимо також передати лямбда-об'єкт у конструктор `coll`; у протилежному випадку клас `coll` буде змушений викликати конструктор за замовчуванням для переданого критерію сортування, у той час як лямбда-функції не можуть мати конструкторів за замовчуванням і операціями присвоювання. Отже, для критерію сортування клас, що визначає функціональні об'єкти, може виявитися більш інтуїтивно зрозумілим.

Інша проблема, зв'язана з лямбда-функціями, полягає в тім, що лямбда-функція не може мати внутрішнього стану, що могло б зберігатися при багаторазових викликах. Якщо потрібно такий стан, то необхідно оголосити чи об'єкт перемінну в зовнішній області видимості і передати її по посиланню в захоплення лямбда-функції. По контрасту функціональні об'єкти дозволяють інкапсулювати внутрішній стан. Тем лямбда-функції можна використовувати для визначення хеш-функцій і/чи критерію еквівалентності для неупорядкованих контейнерів.

Література

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