Лекція 20

Ітератори

20.1. Поняття про ітератор

У відповідності зі стандартом C++11 всі елементи контейнера можна обійти за допомогою діапазонної операції `for`. Однак, для того щоб знайти елемент, обходити всі елементи не обов'язково. Замість цього варто перебирати всі елементи, поки не буде знайдений шуканий. Крім того, його позицію, возможно, бажано десь зберегти, наприклад, щоб продовжити обхід згодом. Таким чином, необхідна концепція об'єкта, що представляє позицію елемента в контейнері. Така концепція існує. Об'єкти, що її реалізують, називаються ітераторами. Як буде показано далі, діапазонний цикл `for` надає зручний інтерфейс для реалізації цієї концепції. Інакше кажучи, він використовує ітератори для обходу всіх елементів контейнера.

Ітератор — це об'єкт, що перебирає всі елементи (переходить від одного елемента до іншого). Він може обійти всі елементи контейнера STL чи його підмножини. Ітератор представляє визначену позицію в контейнері. Для ітератора визначені наступні фундаментальні операції.

- **Операція *** повертає елемент, що стоїть в поточній позиції. Якщо цей елемент має члени, то за допомогою операції `->` можна одержати доступ до них безпосередньо з ітератора.
- **Операція ++** переміщає ітератор уперед на наступний елемент. Більшість ітераторів також дозволяють повернення до попереднього елементу за допомогою операції `--`.
- **Операції ==** і **!=** повертають результат перевірки, чи посилаються два ітератора на одну позицію.
- **Операція =** привласнює ітератор (позицію елемента, на яку він посилається).

Ці операції використовують інтерфейс, що точно збігається з інтерфейсом звичайних вказівників у мовах C і C++, за допомогою яких можна обійти елементи в звичайному масиві. Різниця полягає в тім, що ітератор є *інтелектуальним вказівником,* тобто може обходити більш складні структури даних. Внутрішнє поводження ітераторів залежить від структури даних, по якій вони переміщаються. З цієї причини кожен контейнерний тип передбачає свій власний вид ітераторів. У результаті ітератори мають загальний інтерфейс, але різні типи. Це безпосередньо приводить до концепції узагальненого програмування: операції використовують однаковий інтерфейс, але мають різні типи, тому можна

використовувати шаблони для формулювання узагальнених операцій, що застосовуються до довільних типів, що задовольняють зазначеному інтерфейсу.

Усі контейнерні класи мають однакові основні функції-члени, що дозволяють переміщати ітератори по елементах контейнера. Найбільш важливими з них є такі.

- Функція **`begin()`** повертає ітератор, що представляє початок контейнера, тобто позицію першого елемента, якщо такий мається в контейнері.
- Функція **`end()`** повертає ітератор, що представляє кінець контейнера, тобто позицію, що *слідує за* останнім елементом. Такий ітератор називається *позамежним* (past-the-end iterator).

Таким чином, функції-члени `begin()` і `end()` визначають *напіввідчинений діапазон* (half-open range), що включає перший елемент і не містить останній. Напіввідчинений *діапазон* має дві переваги.

1. Існує простий критерій зупинки циклу при обході всіх елементів: цикл продовжується, поки не буде досягнута позиція `end()`.
2. Він дозволяє уникнути спеціальної обробки порожніх діапазонів. Для порожніх діапазонів позиція `begin()` збігається з позицією `end()`.

Наступний приклад демонструє використання ітераторів для виводу на екран всіх елементів списку:

```
// stl/list1old.cpp

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

 int main()
 {
   list<char> coll; // список символів

   // додаємо елементи від 'a' до 'z'
   for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
   }

   // виводимо на друк всі елементи:
   // - обходимо всі елементи
   list<char>::const_iterator pos;
   for (pos = coll.begin(); pos != coll.end(); ++pos) {
```

```
        cout << *pos << ' ';
   }
   cout << endl;
 }
```

Знову після створення списку і заповнення його символами від `'a'` до `'z'`, ми виводимо на екран всі елементи. Однак замість діапазонного циклу `for`:

```
for (auto elem : coll) {
    cout << elem << ' ';
 }
```

тепер всі елементи виводяться в звичайному циклі за допомогою ітераторів, що обходить елементи контейнера:

```
list<char>::const_iterator pos;
 for (pos = coll.begin(); pos != coll.end(); ++pos) {
     cout << *pos << ' ';
 }
```

Ітератор `pos` з'являється прямо перед циклом. Він має тип ітератора для доступу до константних елементів контейнерного класу.

```
list<char>::const_iterator pos;
```

У кожнім контейнері з'являються два типи ітераторів.

1. Ітератор *контейнер*`::iterator` переміщається по елементах у режимі читання/запису.
2. Ітератор *контейнер*`::const_iterator` переміщається по елементах тільки в режимі читання.

Наприклад, у класі `list` визначення можуть мати наступний вид:

```
namespace std {
   template <typename T>
   class list {
      public:
         typedef ... iterator;
         typedef ... const_iterator;
         ...
   };
 }
```

Точний тип `iterator` і `const_iterator` визначається реалізацією.

У циклі `for` ітератор `pos` ініціалізується позицією першого елемента:

```
pos = coll.begin()
```

Цикл продовжується, поки ітератор `pos` не досягне кінця контейнера:

```
pos != coll.end()
```

Тут ітератор `pos` порівнюється з так називаним позамежної ітератором, що представляє позицію, що слідує за останнім елементом. Коли цикл виконує операцію інкремента `++pos`, ітератор `pos` переміщається на наступний елемент. Ітератор `pos` переміщається від першого елемента до останнього. Якщо контейнер не містить елементів, то цикл не виконується, тому що `coll.begin()` дорівнює `coll.end()`.

У тілі циклу вираз `*pos` представляє поточний елемент. У даному прикладі він записується в стандартний потік виводу `cout`, а за ним слідує символ пробілу. Елементи модифікувати не можна, оскільки використовується тип ітератора `const_iterator`. Таким чином, з погляду ітератора елемент є константою. Однак, якщо використовувати неконстантний ітератор і неконстантний тип елементів, то можна змінювати значення елементів. Розглянемо приклад:

```
// переводимо всі символи в списку у верхній регістр
 list<char>::iterator pos;
 for (pos = coll.begin(); pos != coll.end(); ++pos) {
 *pos = toupper(*pos);
 }
```

Якщо ітератори використовуються для обходу елементів (неупорядкованих) відображень і мультивідображень, то ітератор `pos` посилається на пари “значення^-ключ-значення”. Таким чином, вираз

```
pos->second
```

повертає другу частину пари “значення^-ключ-значення”, тобто значення елемента, а вираз

```
pos->first
```

повертає (константний) ключ.

Порівняння операцій ++pos і pos++

Відзначимо, що для переміщення ітератора на наступний елемент тут використовується операція префіксного інкремента ++. Причина полягає в тім, що ця операція теоретично забезпечує більш високу швидкодію в порівнянні з постфиксною операцією інкремента. Постфиксна операція неявно використовує тимчасовий об'єкт, тому що вона повинна повертати стару позицію ітератора. З цієї причини, узагалі говорячи, `++pos` кращий за `pos++`. Таким чином, що наступної версії варто уникати:

```
for (pos = coll.begin(); pos != coll.end(); pos++) {
                                            ^^^^^ // OK,
                                  // але небагато повільніше
   ...
}
```

Реально такі способи підвищення продуктивності програми майже ніколи не досягають мети, тому не слід інтерпретувати наші рекомендації занадто буквально для досягнення мікроскопічних переваг. Читабельність і зручність супроводу програм набагато важливіше, ніж оптимізація їхньої швидкодії. Варто також підкреслити, що в даному випадку вибір префіксної форми інкремента і відмова від постфіксної форми не позв'язана з жодними додатковими витратами. Таким чином, рекомендація використовувати префіксні форми інкремента і декремента — просто добра порада.

Функції cbegin() і cend()

У відповідності зі стандартом C++11 ключове слово `auto` дозволяє вказати точний тип ітератора (за умови, що ітератор був ініціалізований під час оголошення, так що його тип можна вивести з його початкового значення). Таким чином, безпосередня ініціалізація ітератора за допомогою функції `begin()` дозволяє використовувати ключове слово `auto` для оголошення його типу:

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
     cout << *pos << ' ';
 }
```

Легко бачити, що використання ключового слова `auto` робить код більш компактним. Без ключового слова `auto` оголошення ітератора в циклі виглядало б у такий спосіб:

```
for (list<char>::const_iterator pos = coll.begin();
     pos != coll.end(); ++pos) {
     cout << *pos << ' ';
 }
```

Інша перевага застосування `auto` полягає в тім, що цикл є стійким до змін коду, таким як модифікація типу контейнера. Однак у такої конструкції є недолік — ітератор втрачає свою константність, тобто з'являється ризик ненавмисного присвоювання. Вираз

```
auto pos = coll.begin()
```

робить ітератор `pos` неконстантним, тому що функція `begin()` повертає об'єкт типу *контейнер*`::iterator`. Для того щоб зберегти константність ітератора, у стандарті C++11 передбачені функції `cbegin()` і `cend()`. Вони повертають об'єкт типу *контейнер*`::const_iterator`.

Резюмуючи, відзначимо, що в стандарті C++11 цикл, що дозволяє обходити всі елементи контейнера без використання діапазонного циклу `for`, може мати наступний вид:

```
for (auto pos = coll.cbegin(); pos != coll.cend(); ++pos) {
     ...
 }
```

Діапазонний цикл for і ітератори

Увівши поняття ітератора, ми можемо пояснити точне поводження діапазонного циклу `for` (range-based for). Для контейнерів діапазонний цикл `for` просто надає зручний інтерфейс, що дозволяє обійти всі елементи переданого діапазону чи колекції, і нічого більше. У кожнім циклі реальний елемент ініціалізується значенням, на яке посилається поточний ітератор.

Таким чином, конструкція

```
for (type elem : coll) {
   ...
 }
```

інтерпретується як

```
for (auto pos=coll.begin(), end=coll.end(); pos!=end; ++pos) {
     type elem = *pos;
     ...
 }
```

Тепер стає зрозумілим, чому об'єкт `elem` повинний з'являтися як константне посилання, щоб уникнути непотрібного копіювання. У протилежному випадку об'єкт `elem` іеніціалізувавя б копією значення `*pos`.

20.2. Додаткові приклади використання асоціативних і неупорядкованих контейнерів

Тепер, одержавши представлення про ітератори, можна привести декілька прикладів програм, що використовують асоціативні контейнери без використання таких мовних конструкцій стандарту C++11, як діапазонний цикл `for`, ключове слово `auto` і списки ініціалізації. Крім того, використовувані тут конструкції в деяких ситуаціях можуть виявитися корисними й у сполученні зі стандартом C++11.

Використання множини до появи стандарту C++11

Перший приклад демонструє вставку елементів у множина і застосування ітераторів без використання мовних засобів зі стандарту C++11.

```
// stl/set1.cpp

 #include <set>
 #include <iostream>

 int main()
 {
   // тип колекції
```

```
   typedef std::set<int> IntSet;

   IntSet coll; // оголошуємо множину цілих чисел

   // вставляємо елементи від 1 до 6 у довільному порядку
   // - зверніть увагу на два виклики функції insert() зі значенням 1
   coll.insert(3);
   coll.insert(1);
   coll.insert(5);
   coll.insert(4);
   coll.insert(1);
   coll.insert(6);
   coll.insert(2);

   // виводимо на друк всі елементи
   // - обходимо всі елементи
   IntSet::const_iterator pos;
   for (pos = coll.begin(); pos != coll.end(); ++pos) {
        std::cout << *pos << ' ';
   }
   std::cout << std::endl;
}
```

Как зазвичай, директива `include` визначає всі необхідні типи й операції над множинами.

```
#include <set>
```

Тип контейнера використовується в декількох місцях, тому спочатку визначаємо його скорочене ім'я.

```
typedef set<int> IntSet;
```

Ця інструкція визначає тип `IntSet` як множину елементів типу `int`. Цей тип використовує критерій сортування, використовуваний за замовчуванням, що упорядковує елементи за допомогою операції <, так що елементи слідують у зростаючому порядку. Для того щоб упорядкувати елементи в зворотньому порядку чи використати зовсім інший критерій сортування, програміст може передати його в якості другого шаблонового параметра. Наприклад, що слідує інструкція визначає тип множинаі, у якому елементи упорядковані по убуванню:

```
typedef set<int,greater<int>> IntSet;
```

Об'єкт `greater<>` — це стандартний функціональний об'єкт.

Всі асоціативні контейнери передбачають функція-член `insert()` для вставки нового елемента.

```
coll.insert(3);
coll.insert(1);
...
```

У відповідності зі стандартом C++11 можна написати просто:

```
coll.insert ( { 3, 1, 5, 4, 1, 6, 2 } );
```

Кожен вставлений елемент автоматично займає правильну позицію відповідно до критерію сортування. Функції `push_back()` чи `push_front()` використовувати не можна, тому що вони призначені для послідовних контейнерів. У даному контексті вони не мають змісту, оскільки неможливо вказати позицію нового елемента.

Для того щоб вивести на екран всі елементи контейнера, використовуємо цикл із попереднього прикладу, присвяченого списку. Ітератор обходить всі елементи і виводить їх на друк.

```
IntSet::const_iterator pos;
 for (pos = coll.begin(); pos != coll.end(); ++pos) {
     cout << *pos << ' ';
 }
```

Оскільки ітератор визначається контейнером, цикл працює правильно, навіть якщо внутрішня структура контейнера є більш складною. Наприклад, якщо ітератор посилається на третій елемент, то операція ++ переміщає його до четвертого елемента, що знаходиться на вершині. Після наступного виконання операції ++ ітератор посилається на п'ятий елемент, що знаходиться внизу. Результат роботи програми виглядає в такий спосіб:

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

Для того щоб використовувати мультимножину, а не множину, досить змінити тип контейнера; заголовний файл залишається колишнім.

```
typedef multiset<int> IntSet;
```

Мультимножина допускає дублікати, тому в ньому будуть зберігатися два елементи зі значенням `1`. Таким чином, результат роботи програми зміниться.

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

Подробиці використання неупорядкованої мультимножини

В іншому прикладі демонструється, що відбудеться, якщо виконати обхід всіх елементів неупорядкованої мультимножини

```
// stl/unordmultiset2.cpp

 #include <unordered_set>
 #include <iostream>

 int main()
```

```
{
   // неупорядкована мультимножина з цілими значеннями
   std::unordered_multiset<int> coll;

   // вставляємо декілька елементів
   coll.insert({1,3,5,7,11,13,17,20,23,27,1});

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

   // вставляємо ще один елемент
   coll.insert(25);

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

Порядок проходження елементів не визначений. Він залежить від внутрішньої структури хеш-таблиці і її хеш-функції. Навіть якщо елементи вставлялися один за одним, у контейнері вони розташовуються в довільному порядку. Додавання ще одного елемента може змінити порядок усіх наявних у множини елементів.

Таким чином, один з можливих варіантів результату роботи програми може виглядати в такий спосіб:

```
11 23 1 1 13 3 27 5 17 7 20
23 1 1 25 3 27 5 7 11 13 17 20
```

Як бачимо, порядок проходження елементів дійсно не визначений, так що на іншій платформі цей порядок може виявитися іншим. Додавання одного елемента може змінити порядок проходження всіх елементів. Однак гарантується, що елементи з однаковими значеннями будуть розташовані поруч.

Обхід елементів і вивід їх на екран

```
for (auto elem : coll) {
    std::cout << elem << ' ';
 }
```

еквівалентний наступному коду:

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
     auto elem = *pos;
     std::cout << elem << ' ';
 }
```

Тип внутрішнього ітератора `pos`, використовуваного в циклі `for`, дається контейнером, так що ітератор "знає", як обходити всі елементи.

Якщо переключитися на неупорядковану множина і тим самим заборонити дублікати

```
std::unordered_set<int> coll;
```

то результат роботи програми може стать таким:

```
11 23 1 13 3 27 5 17 7 20
11 23 1 13 25 3 27 5 17 7 20
```

20.3. Категорії ітераторів

Крім основних операцій, ітератори можуть мати функціональні можливості, що залежать від внутрішньої структури контейнера. як зазвичай, бібліотека STL передбачає лише ті операції, що забезпечують високу продуктивність. Наприклад, якщо контейнери надають довільний доступ (наприклад, вектори і деки), їх ітератори також можуть виконувати операції довільного доступу, такі як установка ітератора на конкретний елемент.

Ітератори підрозділяються на *категорії* по їхніх загальних можливостях. Ітератори стандартних контейнерних класів відносяться до однієї з трьох категорій.

1. **Односпрямований ітератор** може переміщатися тільки вперед, використовуючи операцію инкремента. Ітератори класу `forward_list` є прямими. Ітератори контейнерних класів `unordered_set`, `unordered_multiset`, `unordered_map` і `unordered_multimap` є "принаймні" прямими (бібліотекам дозволяється заміняти їх двоспрямованими ітераторами).
2. **Двоспрямовані ітератори** можуть обходити контейнери в двох напрямках: уперед за допомогою операції інкремента і назад за допомогою операції декремента. Ітератори контейнерних класів `list`, `set`, `multiset`, `map` і `multimap` є двоспрямованими.
3. **Ітератори довільного доступу** має усі властивості двоспрямованих ітераторів. Крім того, вони можуть забезпечити довільний доступ. Зокрема, вони виконують операції *арифметики ітераторів* (за аналогією з арифметикою звичайних вказівників). Можна складати і віднімати, обчислювати різниці і порівнювати ітератори за допомогою операцій порівняння, наприклад < і >. Ітератори контейнерних класів `vector`, `deque`, `array` і `string` є ітераторами довільного доступу.

Крім того, існують ще дві категорії ітераторів.

- **Ітератори введення** можуть зчитувати й обробляти значення, переміщаючи вперед. До цієї категорії, зокрема, відносяться потокові ітератори.
- **Ітератори виводу** можуть записувати значення, переміщаючи вперед. Прикладами таких ітераторів є ітератори вставки і потокові ітератори **виводу**.

Для створення узагальненого коду, якомога менш залежного від типу контейнера, не слід використовувати спеціальні операції для ітераторів довільного доступу. Наприклад, такий цикл буде працювати з усіма контейнерами:

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
     ...
 }
```

Однак наступний код *не працює* з усіма контейнерами:

```
for (auto pos = coll.begin(); pos < coll.end(); ++pos) {
     ...
 }
```

Єдина різниця між ними полягає у використанні операції < замість операції `!=` в умові виходу з циклу. Операція < передбачена тільки для ітераторів довільного доступу, тому цей цикл не буде працювати зі списками, множинами і відображеннями. Для того щоб написати узагальнений код для довільних контейнерів, варто використовувати операцію `!=`, а не <. Однак у цьому випадку код стає менш безпечним, тому що можна не розпізнати вихід ітератора `pos` за позицію `end()`. Вибір версії залежить від програміста; він може залежати від контексту чи особистих вподобань.

Для того щоб уникнути непорозумінь, варто підкреслити, що мова йде про *категорії*, а не про *класи* ітераторів. Категорія визначає лише можливості ітераторів. Тип не має значення. Узагальнена концепція бібліотеки STL працює з *чистою абстракцією*: усе, що *поводиться* як двоспрямований ітератор, *є* двоспрямованим ітератором.

20.4. Адаптери ітераторів

Ітератори є *чистою абстракцією*: усе, що *поводиться* як ітератор, *є* ітератором. З цієї причини можна написати класи, що мають інтерфейс ітераторів, але роблять щось зовсім інше. Стандартна бібліотека C++ містить декілька стандартних ітераторів особливого виду: *адаптери* ітераторів. Це щось більше, ніж просто допоміжні класи; вони додають концепції ітераторів багато більше потужності.

У наступних підрозділах розглядаються перераховані нижче адаптери ітераторів.

1. Ітератори вставки.
2. Потокові ітератори.

3. Зворотні ітератори.
4. Ітератори переміщення (починаючи зі стандарту C++11).

20.5. Ітератори вставки

Ітератори вставки використовуються для того, щоб дати алгоритмам можливість працювати в режимі вставки, а не заміни. Зокрема, ітератори вставки вирішують проблему, що виникає, коли в діапазоні призначення немає достатнього місця: вони дозволяють збільшувати розмір діапазону вставки.

Ітератори вставки перевизначають свій інтерфейс у такий спосіб.

- Якщо ви привласнюєте значення елементу ітератора вставки, він вставляє це значення в колекцію, якій належать ітератор. Три різних ітератори вставки мають різні можливості для вставки елементів — у початок, у кінець і в задану позицію.
- Виклик для переміщення вперед є порожньою операцією.

Ітератори з таким інтерфейсом відносяться до категорії ітераторів виводу, здатних записувати і привласнювати значення тільки при переміщенні вперед.

Розглянемо приклад:

```
// stl/copy2.cpp

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

 int main()
 {
   list<int> coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   // копіюємо елементи колекції coll1 у колекцію coll2,
   // додаючи їх до існуючих

   vector<int> coll2;
   copy (coll1.cbegin(), coll1.cend(), // джерело
         back_inserter(coll2));        // призначення

   // копіюємо елементи з колекції coll1 у колекцію coll3,
```

```
   // уставляючи їх у початок
   // - змінюємо порядок на зворотний
   deque<int> coll3;
   copy (coll1.cbegin(), coll1.cend(), // джерело
         front_inserter(coll3));       // призначення

   // копіюємо елементи колекції coll1 до колекцію coll4
   // - ітератор, що працює тільки з асоціативними колекціями
   set<int> coll4;
   copy (coll1.cbegin(), coll1.cend(),   // джерело
         inserter(coll4,coll4.begin())); // призначення
}
```

Цей приклад демонструє всі три стандартних ітератора вставки.

1. **Ітератори вставки в кінець** вставляє елементи в кінець свого контейнера (додає їх), викликаючи функцію `push_back()`. Наприклад, у наступному прикладі всі елементи колекції `coll1` додаються в колекцію `coll2`:

   ```
   copy (coll1.cbegin(), coll1.cend(), // джерело
         back_inserter(coll2));        // призначення
   ```

   Зрозуміло, ітератори вставки в кінець можна використовувати тільки для контейнерів, що мають функцію-член `push_back()`. У стандартній бібліотеці C++ до таких контейнерів належать `vector`, `deque`, `list` і рядка.

2. Ітератори **вставки в початок** вставляють елементи в початок свого контейнера, викликаючи функцію `push_front()`. Наприклад, наступний оператор вставляє всі елементи колекції `coll1` у колекцію `coll3`:

   ```
   copy (coll1.cbegin(), coll1.cend(), // джерело
         front_inserter(coll3));       // призначення
   ```

   Відзначимо, що цей вид ітераторів вставки змінює порядок вставлених елементів на протилежний. Якщо вставити в початок `1`, а потім `2`, то `1` буде слідувати за `2`.

   Ітератори вставки в початок можна використовувати тільки в контейнерах, що мають функцію-член `push_front()`. У стандартній бібліотеці C++ до таких контейнерів відносяться `deque`, `list` і `forward_list`.

3. **Узагальнені ітератори вставки** вставляють елементи безпосередньо перед позицією, переданої як другий аргумент під час ініціалізації. Узагальнений ітератор вставки викликає функція-член `insert()` з новим значенням і новою позицією, що задаються аргументами. Відзначимо, що функція-член `insert()` містять усі стандартні контейнери, за винятком `array` і `forward_list`. Таким чином, цей ітератор є

єдиним ітератором для роботи зі стандартними асоціативними і неупорядкованими контейнерами.

Однак передавати позицію для вставки елемента в асоціативний чи неупорядкований масив не занадто корисно, чи не так? В асоціативних контейнерах позиції залежать від значень *елементів*, а в неупорядкованих контейнерах позиція елемента не визначена. Рішення просте: для асоціативних і неупорядкованих контейнерів передана позиція вважається *підказкою* для початку пошуку правильної позиції. Однак контейнер може неї ігнорувати.

Функціональні можливості ітераторів вставки перераховані в табл. 21.1.

Таблиця 21.1. Стандартні ітератори **вставки**

Вираз	***Що робить ітератор вставки***
`back_inserter(`***контейнер***`)`	Додає елементи в тім же порядку, використовуючи функцію `push_back(`***значення***`)`
`front_inserter(`***контейнер***`)`	Вставляє елемент у початок у зворотному порядку, використовуючи функцію `push_front(`***значення***`)`
`inserter(`***контейнер,позиція***`)`	Вставляє елементи в зазначену *позицію* (у тім же порядку), використовуючи функцію `insert(`***позиція,значення***`)`

20.6. Потокові ітератори

Потокові ітератори зчитують дані з чи потоку записують дані в потік. Таким чином, вони створюють абстракцію, що дозволяє вводити дані з клавіатури, що розгля дається як колекція, призначена для читання. Аналогічно за допомогою потокового ітератора можна перенаправляти вивід алгоритму безпосередньо в файл чина екран.

Наступний приклад демонструє потужність усієї бібліотеки STL. У порівнянні зі звичайними програмами мовою C чи C++, цей приклад виконує дуже складну роботу за допомогою декількох операцій.

```
// stl/ioiter1.cpp
#include <iterator>
 #include <algorithm>
 #include <vector>
 #include <string>
 #include <iostream>
```

```
using namespace std;

int main()
{
  vector<string> coll;
  // зчитуємо всі слова зі стандартного потоку введення
  // - джерело: усі рядки аж до кінця файлу (чи помилки)
  // - призначення: coll (вставка)
  copy (istream_iterator<string>(cin), // початок джерела
        istream_iterator<string>(),    // кінець джерела
        back_inserter(coll));          // призначення

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

  // виводимо всі елементи без дублікатів
  // - джерело: coll
  // - призначення: стандартний потік виводу
  // (із символом переходу на новий рядок між елементами)
  unique_copy (coll.cbegin(), coll.cend(),           // джерело
               ostream_iterator<string>(cout,"\n")); // призначення
}
```

Ця програма містить тільки три операторів, що зчитують усі слова зі стандартного потоку введення і виводять їхній упорядкований список. Розглянемо ці три оператори по черзі. В операторі

```
copy (istream_iterator<string>(cin),
      istream_iterator<string>(),
      back_inserter(coll));
```

використовуються два потокових ітератора введення.

1. Вираз

   `istream_iterator<string>(cin)`

   створює потоковий ітератор, що зчитує дані зі стандартного потоку введення `cin`. Шаблонний аргумент `string` указує, що потоковий ітератор зчитує елементи даного типу. Ці елементи зчитуються за допомогою звичайної операції введення >>. Таким чином, щораз, коли алгоритм хоче обробити новий елемент, потоковий ітератор уведення трансформує це бажання у виклик `cin >>` *рядок*.

   Операція введення рядків звичайно зчитує одне слово, відділене роздільниками, так що алгоритм зчитує слово за словом.

2. Вираз

`istream_iterator<string>()`

викликає конструктор потокових ітераторів, заданий за замовчуванням, що створює так називаний *ітератор кінця потоку* (end-of-stream iterator). Він представляє потік, читання з який більше неможливо.

Зазвичай, алгоритм `copy()` працює, тільки якщо перший аргумент (інкрементований) відрізняється від другого. Ітератор кінця потоку використовується як *кінець діапазону,* тому алгоритм зчитує всі рядки з потоку `cin`, поки може (тобто пока не досягне кінця чи потоку не відбудеться помилка). Підводячи підсумки, відзначимо, що джерелом алгоритму є "усі слова, зчитані з потоку `cin`". Ці слова копіюються шляхом вставки в колекцію `coll` за допомогою зворотного ітератора.

Алгоритм `sort()` сортує всі елементи:

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

І нарешті, оператор

```
unique_copy (coll.cbegin(), coll.cend(),
ostream_iterator<string>(cout,"\n"));
```

копіює всі елементи з колекції у вихідний потік `cout`. Протягом цього процесу алгоритм `unique_copy()` видаляє дублікати. Вираз

```
ostream_iterator<string>(cout,"\n")
```

створює ітератор потоку виводу, що записує об'єкти класу `string` у потік `cout`, виконуючи операцію << для кожного елемента.

Другий аргумент, що слідує за `cout`, є необов'язковим і служить як роздільник між елементами. У нашому прикладі цим роздільником є символ переходу на новий рядок, тому кожен елемент буде записаний в окремому рядку.

Усі компоненти нашої програми є шаблононними, тому її легко настроїти на сортування будь-яких інших типів, таких як цілі чи числа більш складні об'єкти. У нашому прикладі для сортування всіх слів, уведених зі стандартного потоку введення, використовувалося одне оголошення і три інструкції. Однак те ж саме можна зробити за допомогою одного оголошення й однієї інструкції. Відповідний приклад приведений у розділі 1.

20.7. Зворотні ітератори

Зворотні ітератори дозволяють алгоритмам перебирати елементи контейнера в зворотному напрямку, заміняючи операцію інкремента на операцію декремента, і навпаки. Усі контейнери з двоспрямованими ітераторами чи операціями довільного доступу (тобто всі послідовні контейнери, за винятком `forward_list` і всіх асоціативних контейнерів)

можуть створювати зворотні ітератори за допомогою функцій-членів `rbegin()` і `rend()`. У відповідності зі стандартом C++11 у контейнерах також передбачені функції-члени, що повертають ітератори, призначені тільки для читання, `crbegin()` і `crend()`.

У контейнерах `forward_list` і неупорядкованих контейнерах інтерфейс для зворотного обходу (`rbegin()`, `rend()` і т.д.) не передбачений. Причина полягає в тім, що їх реалізація для обходу елементів використовує тільки однозв'язні списки.

Розглянемо наступний приклад:

```
// stl/reviter1.cpp

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

 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 в зворотному порядку
   copy (coll.crbegin(), coll.crend(),       // джерело
         ostream_iterator<int>(cout," "));   // призначення
   cout << endl;
 }
```

Наступний вираз повертає зворотний ітератор, призначений тільки для читання елементів контейнера `coll`.

```
coll.crbegin()
```

Цей ітератор можна використовувати в якості початку зворотного обходу елементів колекції. Він установлений на останній елемент колекції. Таким чином, що слідує вираз повертає значення останнього елемента:

```
*coll.crbegin()
```

Відповідно, що слідує вираз повертає зворотний ітератор для колекції `coll`, якому можна використовувати як кінець для зворотного обходу:

```
coll.crend()
```

Як зазвичай при роботі з діапазонами, позиція ітератора слідує за останнім елементом, але в зворотному напрямку; інакше кажучи, він розташована *перед* першим елементом колекції.

Ніколи не слід застосовувати операцію `*` (чи операцію `->`) до позиції, що не містить коректного елемента. Таким чином, вираз

```
*coll.crend()
```

не визначений, так само як і вираз `*coll.end()` і `*coll.cend()`.

Перевага використання зворотних ітераторів полягає в тім, що всі алгоритми здатні обходити елементи колекції в зворотному напрямку без спеціального коду. Перехід до наступного елементу за допомогою операції `++` перевизначається за допомогою операції `--`. Наприклад, у нашому випадку алгоритм `copy()` обходить елементи колекції `coll` від останнього до першого елемента. Таким чином, одержуємо наступні результати роботи програми:

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

Звичайні ітератори також можна перетворювати в зворотні, і навпаки. Однак розіменоване значення ітератора при цьому змінюється.

20.8. Ітератори переміщення

Ітератори переміщення введені в стандарті C++11. Вони перетворюють будь-який доступ до елемента в операцію переміщення. Це дозволяє переміщати елементи з одного контейнера в інший або в конструкторах, або за допомогою алгоритмів.

Література

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