Лекція 19

Стандартна бібліотека шаблонів

Стандартна бібліотека шаблонів (standard template library — STL) є ядром стандартної бібліотеки мови C++, що вплинуло на всю її архітектуру. STL — це бібліотека універсальних компонентів для керування колекціями даних за допомогою сучасних і ефективних алгоритмів. Вона дозволяє програмістам використовувати найсучасніші досягнення в області структур даних і алгоритмів, не вникаючи в те, як вони працюють.

З погляду програміста бібліотека STL містить сукупність класів колекцій для різних цілей і набір алгоритмів для роботи з ними. Усі компоненти STL є шаблонами, тому їх можна використовувати для довільних типів елементів. Однак це не все. Бібліотека STL утворює основу для створення інших класів колекцій і алгоритмів, що працюють разом з існуючими класами колекцій і алгоритмами. Завдяки цьому бібліотека STL піднімає мову C++ на новий рівень абстракції. Забудьте про програмування динамічних масивів, зв'язаних списків, бінарних дерев і хеш-таблиць, забудьте про програмування різних алгоритмів пошуку. Для використання необхідної колекції досить визначити відповідний контейнер і викликати відповідні функції-члени й алгоритм для обробки даних.

Однак за гнучкість бібліотеки STL приходиться розплачуватися її складністю. У цій лекції викладається загальна концепція бібліотеки STL і прийоми програмування, необхідні для роботи з нею. Перші приклади демонструють використання бібліотеки STL і головні моменти, на які варто звернути увагу. У главах 7–11 докладно обговорюються компоненти бібліотеки STL (контейнери, ітератори, функціональні об'єкти й алгоритми) і приводяться додаткові приклади.

19.1. Компоненти бібліотеки STL

В основі бібліотеки STL лежить взаємодія різних добре структурованих компонентів, головними серед який є контейнери, ітератори й алгоритми.

- **Контейнери** використовуються для керування колекціями об'єктів визначеного типу. Кожен вид контейнерів має свої переваги і недоліки, тому наявність різних контейнерів відображає різні вимоги до колекцій, що існують у програмах. Контейнери можуть бути реалізовані або як масиви, або як зв’язані списки і можуть мати спеціальний ключ для кожного елемента.
- **Ітератори** використовуються для обходу елементів у колекціях об'єктів. Цими колекціями можуть бути контейнери чи підмножини контейнерів. Основна перевага

ітераторів полягає в тім, що вони надають невеликий і в той же час універсальний інтерфейс для довільного типу контейнера. Наприклад, одна з операцій цього інтерфейсу переміщає ітератор на наступний елемент у колекції. Виконання цієї операції не залежить від внутрішньої структури колекції. Яка б ні була колекція — масив, дерево чи хеш-таблиця, — ця операція працює однаково, оскільки кожен контейнер визначає свій власний тип ітератора, що просто "правильно працює", оскільки знає внутрішню структуру свого контейнера.

Інтерфейс ітераторів майже збігається з інтерфейсом звичайних вказівників. Для переходу ітератора до наступного елементу виконується операція ++, а для доступу до значення, на яке посилається ітератор, — операція *. Отже, ітератор можна розглядати як різновид інтелектуального вказівника, що перетворює команду "перейти до наступного елементу" у придатну операцію.

- **Алгоритми** призначені для обробки елементів колекцій. Наприклад, алгоритми можуть шукати, сортувати, модифікувати і просто використовувати елементи для різних цілей. Алгоритми використовують ітератори. Таким чином, оскільки інтерфейс ітераторів є загальним для всіх типів контейнерів, алгоритм досить написати один раз, і він буде працювати з будь-яким контейнером

Для того щоб підвищити гнучкість алгоритмів, їх можна використовувати в сполученні з допоміжними функціями, що викликаються алгоритмами. Таким чином, універсальний алгоритм можна використовувати для розв'язання своєї задачі, навіть якщо ця задача є дуже специфічною чи складною. Наприклад, програміст може задати особливий критерій пошуку чи особливу операцію для об'єднання елементів. З появою стандарту C++11, що включає лямбда-функції, з'явилася можливість описувати практично будь-яку функціональність при обході елементів контейнера.

Концепція бібліотеки STL заснована на поділі даних і операцій. Дані керуються контейнерними класами, а операції визначаються алгоритмами. Сполучною ланкою між цими двома компонентами є ітератори. Вони дозволяють алгоритму взаємодіяти з будь-яким контейнером.

У деякому сенсі концепція бібліотеки STL суперечить вихідній ідеї об'єктно-орієнтованого програмування: STL *відокремлює* дані від алгоритмів, а не поєднує їх. Однак для цього є вагома причина. У принципі, програміст може поєднувати будь-який контейнер з будь-яким алгоритмом, так що результат буде дуже гнучким і в той же час компактним.

Однієї з основних особливостей бібліотеки STL є те, що усі компоненти працюють з довільними типами, як слідує з назви “стандартна бібліотека шаблонів”. Усі компоненти є шаблонами, що допускають використання будь-якого типу, за умови, що цей тип здатний виконувати необхідні операції. Таким чином, бібліотека STL — яскравий приклад концепції *узагальненого програмування* (generic programming). Контейнери й алгоритми є узагальненими стосовно довільних типів і класів відповідно.

У бібліотеці STL є ще більш універсальні компоненти. За допомогою деяких *адаптерів* і *функціональних об'єктів* (чи функторів) програміст може розширювати чи обмежувати алгоритми й інтерфейси для конкретних цілей. Спочатку розглянемо цю концепцію на прикладах. Це кращий спосіб зрозуміти, як працює бібліотека STL, і освоїти її.

19.2. Контейнери

Контейнерні класи, чи *контейнери*, керують колекціями елементів. Для різних цілей у бібліотеці STL передбачені різні контейнери. Існують три різновиди контейнерів.

1. **Послідовні контейнери** — це *упорядковані колекції*, у яких кожен елемент займає визначену позицію. Ця позиція залежить від часу і місця вставки, але не залежить від значення елемента. Наприклад, якщо вставити шість елементів в упорядковану колекцію, додаючи кожен елемент у кінець колекції, то ці елементи будуть слідувати в точному порядку їх вставки. Бібліотека STL містить п'ять стандартних контейнерних класів: `array`, `vector`, `deque`, `list` і `forward_list`.
2. **Асоціативні контейнери** — це *упорядковані колекції*, у яких позиція елемента залежить від його значення (чи ключа, якщо елемент являє собою пару “ключ-значення”) відповідно до визначеного критерію сортування. Якщо вставити шість елементів у таку колекцію, то порядок їх проходження буде визначений їхніми значеннями. У цьому випадку порядок вставки елементів не має значення. Бібліотека STL містить чотири стандартних асоціативних контейнерних класи: `set`, `multiset`, `map` і `multimap`.
3. **Неупорядковані (асоціативні) контейнери —** це *неупорядковані колекції*, у яких позиція елемента не має значення. У цьому випадку зміст має тільки одне питання: чи належить конкретний елемент такій колекції. Ані порядок вставки, ані значення вставленого елемента не впливає на його позицію. Його позиція згодом може змінюватися. Таким чином, якщо вставити в таку колекцію шість елементів, то їх порядок буде невизначеним і згодом може змінитися. Бібліотека STL містить чотири стандартні неупорядковані контейнерні класи: `unordered_set`, `unordered_multiset`, `unordered_map` і `unordered_multimap`.

Неупорядковані контейнери були введені в документі TR1 і викликали невелику плутанину в контейнерній термінології. Офіційно неупорядковані контейнери відносяться до категорії “неупорядковані асоціативні контейнери”. Не зовсім ясно, що в цій назві мається на увазі під асоціативним контейнером: термін, що поєднує упорядковані і неупорядковані асоціативні контейнери, чи аналог неупорядкованих контейнерів? Відповідь часто залежить від контексту. Ми під асоціативним контейнером маємо на увазі упорядкований асоціативний контейнер (у старому змісті), а термін “неупорядковані контейнери” уживатимемо без проміжного слова “асоціативні”.

Три категорії контейнерів, уведені вище, являють собою логічні групи, що залежать від визначення порядку проходження елементів. Відповідно до такої точки зору асоціативний контейнер можна вважати особливим різновидом послідовного контейнера, тому що упорядковані колекції мають додаткову можливість установлювати порядок проходження елементів у відповідності зі спеціальним критерієм сортування. Це цілком природно, особливо при використанні інших бібліотек класів колекцій, наприклад, бібліотеки мови Smalltalk чи бібліотеки NIHCL[1] , у яких відсортовані колекції є похідними від упорядкованих колекцій. Однак типи колекцій у бібліотеці STL різко відрізняються друг від друга і мають зовсім різні реалізації, що не є похідними друг від друга.

- Послідовні контейнери звичайно реалізуються як масиви чи зв'язані списки.
- Асоціативні контейнери, як правило, реалізуються як бінарні дерева.
- Неупорядковані контейнери звичайно реалізуються як хеш-таблиці.

Строго говорячи, у стандартній бібліотеці C++ не визначена конкретна реалізація ни контейнера. Однак поводження і складність, визначені стандартом, не залишають великого вибору варіантів. З цієї причини на практиці реалізації розрізняються лише незначними деталями.

При виборі придатного контейнера варто враховувати його можливості, а не порядок проходження елементів. Фактично автоматичне сортування елементів в асоціативному масиві *не означають*, що ці контейнери призначені спеціально для сортування елементів. Елементи можна сортувати й у послідовному контейнері.

Основна перевага автоматичного сортування — більш висока швидкодія при пошуку елементів. Зокрема, завжди можна використовувати бінарний пошук, що має логарифмічну, а не лінійну складність. Це означає, наприклад, що при пошуку елемента в колекції, що складається з 1000 елементів, у середньому приходиться виконувати тільки 10, а не 500

[1] Библиотека National Institutes of Health’s Class Library (NIHCL) — одна из первых библиотек классов на языке C++.

порівнянь (див. раздел 2.2). Таким чином, автоматичне сортування — це лише (корисний) "побічний ефект" реалізації асоціативного контейнера, розробленого з метою підвищення швидкодії.

У наступних підрозділах докладно розглядаються контейнерні класи: їх типові реалізації, а також переваги і недоліки. У главі 7 докладно описане точне поводження контейнерних класів, їх загальні й індивідуальні можливості, а також функції-члени. Використання кожного контейнера продемонстровано в розділі 7.12.

19.2.1. Послідовні контейнери

У бібліотеці STL визначені наступні стандартні послідовні контейнери.

- Масиви (клас `array`).
- Вектори.
- Деки.
- Списки (одно- і двозв'язні).

Почнемо з векторів, оскільки в стандарті мови C++ масиви з'явилися пізніше, у документі TR1. Крім того, вони мають особливості, що відрізняють їх від контейнерів бібліотеки STL у принципі.

Вектори

Вектор керує елементами, що зберігаються в динамічному масиві. Він забезпечує довільний доступ до елементів, тобто до кожного елемента можна звернутися прямо по відповідному індексі. Додавання і видалення елементів відбувається наприкінці масиву і виконується дуже швидко[2] . Однак вставка елемента в чи середину в початок масиву займає досить багато часу, тому що всі наступні елементи повинні переміститися, щоб звільнити місце для нового елемента, зберігаючи порядок проходження.

У наступному прикладі визначається вектор цілих чисел, виконуються вставка шести елементів і вивід елементів вектора на екран:

```
// stl/vector1.cpp

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

[2] Строго говоря, добавление элементов выполняется очень швидко *с учетом амортизации*. Добавление отдельного элемента может оказаться медленным, если вектор должен переместиться в новый участок памяти и скопировать туда все существующие элементы. Однако, поскольку перемещение вектора в новый участок памяти происходит относительно редко, в целом добавление очень происходит швидко. (Вопросы сложности обсуждались в разделе 2.2.)

```
int main()
{
   vector<int> coll;    // вектор для цілочисельних елементів

   // додаємо елементи зі значеннями від 1 до 6
   for (int i=1; i<=6; ++i) {
        coll.push_back(i);
   }

  // виводимо на екран всі елементи і пробіл
  for (int i=0; i<coll.size(); ++i) {
       cout << coll[i] << ' ';
  }
  cout << endl;
}
```

Заголовний файл для векторів включається за допомогою директиви

```
#include <vector>
```

Наступне оголошення створює вектор елементів типу `int`:

```
vector<int> coll;
```

Цей вектор не ініціалізований ніякими значеннями, тому конструктор, заданий за замовчуванням, створює його у виді порожньої колекції. Функція `push_back()` додає елемент у контейнер.

```
coll.push_back(i);
```

Ця функція-член є у всіх послідовних контейнерах, у яких можливе додавання елемента за досить швидкий час.

Функція-член `size()` повертає кількість елементів, що містяться в контейнері.

```
for (int i=0; i<coll.size(); ++i) {
   ...
}
```

Функція `size()` є в будь-якому контейнерному класі, за винятком однозв'язних списків (класу `forward_list`). Використовуючи операцію індексування `[]`, можна одержати доступ до окремого елемента вектора.

```
cout << coll[i] << ' ';
```

Тут елементи виводяться в стандартний вихідний потік данных, тому результат роботи програми виглядає в такий спосіб:

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

Деки

Термін *дек* (deque) є абревіатурою від “double-ended queue” (“двостороння черга”). Дек — це динамічний масив, реалізований таким чином, що він може зростати в обох напрямках. Таким чином, вставка елемента в кінець *і* в початок дека виконується швидко. Однак вставка елемента в середину дека може зажадати більше часу, тому що при этом необхідно переміщати елементи.

У наступному прикладі з'являється дек, що містить числа з десятковою точкою, виконується вставка елементів від 1.1 до 6.6 у початок контейнера і вивід всіх елементів на на екран:

```
// stl/deque1.cpp

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

 int main()
 {
   deque<float> coll; // дек для чисел із плаваючою точкою,
                      // вставляємо елементи від 1.1 до 6.6 у початок дека
   for (int i=1; i<=6; ++i) {
        coll.push_front(i*1.1); // insert at the front
   }

   // виводимо на екран всі елементи і пробіл
   for (int i=0; i<coll.size(); ++i) {
        cout << coll[i] << ' ';
   }
   cout << endl;
 }
```

У цьому прикладі заголовний файл для дека включається директивою

```
#include <deque>
```

Наступне оголошення створює порожню колекцію чисел із точкою, що плаває:

```
deque<float> coll;
```

Потім функція `push_front()` вставляє елементи:

```
coll.push_front(i*1.1);
```

Функція `push_front()` вставляє елементи в початок колекції. Цей вид вставки породжує зворотний порядок проходження елементів, тому що кожен елемент, вставлений у

початків дека, передує раніше вставленим елементам. Таким чином, результат роботи програми виглядає в такий спосіб:

```
6.6 5.5 4.4 3.3 2.2 1.1
```

Вставку елементів у дек можна також виконувати за допомогою функції-члена `push_back()`. Однак функція `push_front()` не передбачена для векторів через її низьку продуктивність при роботі з цим видом контейнерів (при вставці елемента в початок вектора необхідно перемістити все його елементи). Зазвичай контейнери з бібліотеки STL містять тільки ті функції-члени, що забезпечують гарну продуктивність, причому під словом “гарна” у даному випадку мається на увазі константна чи логарифмічна складність. Це не дозволяє програмісту викликати функцію, що може знизити швидкодію програми.

Масиви

Об'єкт класу `array` керує своїми елементами, що зберігаються в масиві фіксованого розміру (іноді такі масиви називають статичними чи масивами в стилі мови C). Таким чином, змінювати можна тільки значення елементів, але не їх кількість. Отже, розмір масиву необхідно вказувати в момент його створення. Масив також забезпечує довільний доступ, тобто до кожного елемента масиву можна звернутися за індексом.

Розглянемо приклад масиву, що містить рядки:

```
// stl/array1.cpp

 #include <array>
 #include <string>
 #include <iostream>
 using namespace std;

 int main()
 {
   // масив з 5 рядків:
   array<string,5> coll = { "hello", "world" };

   // виводимо на екран кожен елемент і його індекс
   for (int i=0; i<coll.size(); ++i) {
        cout << i << ": " << coll[i] << endl;
   }
 }
```

Заголовний файл для масивів включається директивою

```
#include <array>
```

Наступне оголошення створює масив з п'яти елементів типу `string`:

```
array<string,5> coll
```

За замовчуванням ці елементи ініціалізуються конструктором, передбаченим за замовчуванням для даного типу елементів. Це значить, що елементи фундаментальних типів мають невизначене початкове значення.

Однак у даній програмі використовується список ініціалізації (див. раздел 3.1.3), що дозволяє ініціалізувати об'єкти класу в момент їх створення значеннями, зазначеними в списку. У відповідності зі стандартом C++11 такий вид ініціалізації передбачений для кожного контейнера, у тому числі для векторів і деків. У нашому випадку для фундаментального типу використовується *ініціалізація нулем*, тобто початкове значення даних фундаментального типу завжди дорівнює нулю (див. раздел 3.2.1).

У даній програмі ми виводимо на екран всі елементи і їх індекси за допомогою функції `size()` і операції індексування `[ ]`. Результати роботи програми мають наступний вид:

```
0: hello
 1: world
 2:3:4

:
```

Як бачимо, програма виводить п'ять рядків, оскільки розмір масиву дорівнює п'яти. У відповідності зі списком ініціалізації перші два елементи рівні `"hello"` і `"world"`, а інші елементи мають значення, задані за замовчуванням, тобто залишаються порожніми рядками. Відзначимо, що кількість елементів є частиною типу масиву. Таким чином, `array<int,5>` і `array<int,10>` — це два різних типи, тому їх не можна привласнювати і порівнювати.

Списки

Історично в мові C++ був тільки один клас списків. Однак у відповідності зі стандартом C++11 бібліотека STL містить два різних типи списків: `list<>` і `forward_list<>`. Таким чином, термін *список* може стосуватися до конкретного класу чи позначати обидва класи списків. Але в деякому змісті односпрямований список (forward list) — це просто обмежений список (list), тому на практиці це розходження не так важливе. Відповідно, коли ми використовуємо термін *список*, то звичайно маємо на увазі клас `list<>`, хоча часто цей термін можна застосовувати і до класу `forward_list<>`. Для того щоб уточнити, який клас ми маємо на увазі, будемо називати клас `forward_list<>` односпрямованим списком. Отже, у цьому розділі ми розглядаємо звичайні списки, що споконвічно були частиною бібліотеки STL.

Клас `list<>` реалізований як двозв’язний список елементів. Це значить, що кожен елемент у списку займає свій власний сегмент пам'яті і посилається на попередній і наступний елементи.

Списки не підтримують довільний доступ до елементів. Наприклад, для доступу до десятого елемента списку необхідно пройти дев'ять попередніх елементів, слідуючи за ланцюжком їх посилань. Однак перехід до наступного чи попереднього елемента виконується за константний час. Таким чином, доступ до довільного елемента має лінійну тимчасову складність, тому що середня відстань переходу пропорційна кількості елементів. Це набагато гірше амортизованого константного часу доступу до елементів векторів і деків.

Перевага списку полягає в тім, що в будь-якій позиції вставка і видалення елемента здійснюються швидко. Для цього досить лише змінити посилання. Завдяки цьому переміщення елемента в середині списку виконується дуже швидко в порівнянні з переміщенням елемента в середині вектора чи дека.

У наступному прикладі ми створюємо порожній список символів, вставляємо в нього символи від `'a'` до `'z'` і виводимо всі елементи на екран:

```
// stl/list1.cpp

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

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

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

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

Зазвичай, для визначення колекції типу `list`, що містить символи, використовується заголовний файл для списків , `<list>`.

```
list<char> coll;
```

Для виводу всіх елементів використовується діапазонний цикл `for`, що з'явився в стандарті C++11, і що дозволяє виконувати оператори для кожного елемента. Прямий доступ до елементів за допомогою операції `[ ]` для списків не передбачений. Це пояснюється тим, що списки не забезпечують довільний доступ і операція `[]` мала би низьку швидкодію.

У циклі використовується ключове слово `auto`, за допомогою якого з'являється тип поточного елемента `coll`. Таким чином, тип об'єкта `elem` автоматично виводиться як `char`, тому що `coll` — це колекція елементів типу `char`. Замість цього можна було б явно оголосити тип об'єкта `elem`.

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

Зверніть увагу на те, що об'єкт `elem` завжди є копією поточного елемента. Таким чином, його можна модифікувати, і це впливало б тільки на оператори, виконувані з цим елементом. В об'єкті `coll` ніякі елементи при цьому не модифікуються. Для того щоб змінити елементи в переданій колекції, об'єкт `elem` необхідно оголосити як неконстантне посилання:

```
for (auto& elem : coll) {
   ... // будь-яка зміна об'єкта elem приводить
       // до зміни поточного елемента в об'єкті coll
 }
```

Для того щоб уникнути копіювання при передачі параметра функції, варто використовувати константне посилання. Таким чином, наступна шаблонна функція виводить на екран всі елементи переданого їй контейнера:

```
template <typename T>
 void printElements (const T& coll)
 {
   for (const auto& elem : coll) {
        std::cout << elem << std::endl;
   }
 }
```

Альтернативним способом виводу всіх елементів на екран до появи стандарту C++11 (без використання ітераторів) був вивід і видалення першого елемента в списку.

```
// stl/list2.cpp
```

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

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

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

  // вивід всіх елементів
  // - до повного вичерпання списку
  // - вивід і видалення першого елемента
  while (! coll.empty()) {
         cout << coll.front() << ' ';
         coll.pop_front();
  }
  cout << endl;
}
```

Функція-член `empty()` повертає результат перевірки наявності елементів контейнері. Цикл продовжується, поки ця функція повертає значення `false` (тобто контейнер містить елементи).

```
while (! coll.empty()) {
   ...
}
```

У циклі функція-член `front()` повертає перший елемент.

```
cout << coll.front() << ' ';
```

Функція `pop_front()` видаляє перший елемент.

```
coll.pop_front();
```

Відзначимо, що функція `pop_front()` не повертає вилучений елемент, тому два попередніх оператори об’єднати не можна.

Результат роботи програми залежить від використовуваного кодування символів. При використанні кодування ASCII результат має такий вигляд:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Односпрямовані списки

З появою стандарту C++11 у стандартну бібліотеку мови C++ включений додатковий контейнер: односпрямований список. Клас `forward_list<>` реалізований як *однозв'язний* список елементів. Як і в звичайному списку `list`, кожен елемент займає окремий сегмент пам'яті, але для економії пам'яті він посилається тільки на наступний елемент. В результаті односпрямований список являє собою обмежений список, у якому не підтримуються всі операції, що передбачають переміщення назад чи викликають зниження швидкодії. З цієї причини в цьому класі не реалізовані такі функції-члени, як `push_back()` і навіть `size()`.

На практиці це обмеження є ще більш незручним, чим здається. Одна з проблем полягає в тім, що елемент неможливо знайти і просто видалити чи вставити перед ним інший елемент. Справа в тім, що, для того щоб видалити елемент, необхідно знаходитися на позиції попереднього елемента, оскільки саме він посилається на наступний елемент. У результаті в односпрямованих списках передбачені спеціальні функції-члени.

Розглянемо невеликий приклад використання односпрямованого списку:

```
// stl/forwardlist1.cpp

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

 int main()
 {
   // створюємо односпрямований список для декількох простих чисел
   forward_list<long> coll = { 2, 3, 5, 7, 11, 13, 17 };

   // двічі змінюємо розмір
   // - примітка: низька продуктивність
   coll.resize(9);
   coll.resize(10,99);

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

Как зазвичай, для роботи з односпрямованими списками використовується заголовний файл <forward_list>. Це дозволяє нам визначити колекцію типу forward_list для довгих цілих чисел, що представляють собою декілька простих чисел.

```
forward_list<long> coll = { 2, 3, 5, 7, 11, 13, 17 };
```

Функція-член resize() змінює кількість елементів. Якщо розмір збільшується, можна передати додатковий параметр для вказівки значень нових елементів. В іншому випадку використовується значення, передбачене за замовчуванням (нуль для елементарних типів). Відзначимо, що виклик функції resize() зв'язаний з великими витратами. Він має лінійну складність, тому що, для того щоб досягти кінця списку, необхідно пройти всі елементи один за іншим. Однак цю операцію передбачають практично всі послідовні контейнери, незважаючи на її можливу низьку продуктивність (цієї функції немає тільки в класі array, тому що в цьому класі розмір є фіксованим).

Зазвичай при роботі зі списками, для виводу на екран всіх елементів використовується діапазонний цикл for. Результати роботи програми мають наступний вид:

```
2 3 5 7 11 13 17 0 0 99
```

19.2.2. Асоціативні контейнери

Асоціативні контейнери автоматично сортують свої елементи відповідно до визначеного критерію. Елементи можуть бути або значеннями будь-якого типу, або парою “ключ-значення”. Для пар “ключ-значення” кожен ключ, що може мати будь-який тип, відображається в асоційоване з ним значення, що також може мати будь-який тип. Критерій сортування може задаватися у вигляді функції, що порівнює значення чи ключі. За замовчуванням контейнери порівнюють елементи чи ключі за допомогою операції <. Однак програміст може створити власну функцію для порівняння, визначивши інший критерій сортування.

Звичайно асоціативні контейнери реалізуються у вигляді бінарних дерев. Отже, кожен елемент (вузол) має одного батька і двох нащадків. Усі предки ліворуч від вузла мають менші значення, а праворуч — великі. Асоціативні контейнери відрізняються видами елементів і способами роботи з дублікатами.

Основна перевага асоціативних контейнерів полягає в тім, що пошук елемента з заданим значенням виконується досить швидко, тому що має логарифмічну складність (у всіх послідовних контейнерах пошук має лінійну складність). Таким чином, при використанні асоціативного контейнера, що містить 1000 елементів, у середньому приходиться виконувати 10, а не 500 порівнянь. Однак асоціативні контейнери мають недолік — значення неможливо

змінювати безпосередньо, тому що при цьому може бути порушений автоматичний порядок сортування елементів.

У бібліотеці STL визначені наступні асоціативні контейнери.

- **Множина** — колекція, у якій елементи сортуються у відповідності зі своїми значеннями. Кожен елемент входить у колекцію тільки один раз, так що дублікати не допускаються.
- Мультимножина — це множина, у якій дозволені дублікати. Таким чином, мультимножина може містити декілька елементів, що мають однакові значення.
- **Відображення** містить пари "ключ-значення". У кожного елемента є ключ, що використовується для сортування, і значення. Кожен ключ може входити у відображення тільки один раз, так що дублікати ключів не дозволяються. Відображення може використовуватися як *асоціативний масив*, тобто масиву, що має індекс довільного типу (див. раздел 19.2.4).
- Мультивідображення — це відображення, у якому дозволені дублікати. Таким чином, мультивідображення може містити декілька елементів, що мають однакові ключі. Мультивідображення може також використовуватися як *словник*.

Усі перераховані асоціативні контейнери мають необов'язковий шаблонний аргумент для критерію сортування. За замовчуванням критерій сортування використовує операцію <. Критерій сортування використовується також для перевірки еквівалентності елементів; інакше кажучи, два елементи є дублікатами, якщо жодний з їхніх значень чи ключів не менше іншого.

Множину можна розглядати як особливий різновид відображення, у якому значення ідентичне ключу. Фактично всі ці асоціативні контейнерні типи звичайно реалізуються за допомогою бінарного дерева.

Приклади використання множина і мультимножин

Розглянемо спочатку приклад використання мультимножини:

```
// stl/multiset1.cpp

 #include <set>
 #include <string>
 #include <iostream>
 using namespace std;

 int main()
```

```
{
  multiset<string> cities {
     "Braunschweig", "Hanover", "Frankfurt", "New York",
     "Chicago", "Toronto", "Paris", "Frankfurt"
  };

  // вивід на екран кожного елемента:
  for (const auto& elem : cities) {
       cout << elem << " ";
  }
  cout << endl;

  // вставка додаткових значень:
  cities.insert( {"London", "Munich", "Hanover", "Braunschweig"} );

  // вивід на друк кожного елемента:
  for (const auto& elem : cities) {
       cout << elem << " ";
  }
  cout << endl;
}
```

Після оголошення типу множини в заголовному файлі `<set>` ми можемо оголосити контейнер `cities` як мультимножину рядків:

```
multiset<string> cities
```

У цьому оголошенні для ініціалізації передається набір елементів, що згодом уставляються за допомогою списку ініціалізації. Для виводу на друк усіх елементів використовується діапазонний цикл `for`. Відзначимо, що елементи з'являються з модифікаторами `const auto&`, тобто ми виводимо тип елементів з контейнера і не створюємо копію кожного елемента для обробки в циклі.

Всі елементи сортуються автоматично, так що перший вивід програми виглядає в такий спосіб:

```
Braunschweig Chicago Frankfurt Frankfurt Hanover New York Paris Toronto
```

Другий вивід має наступний вид:

```
Braunschweig Braunschweig Chicago Frankfurt Frankfurt Hanover Hanover London
Munich New York Paris Toronto
```

Оскільки ми використовуємо мультимножину, а не множина, дублікати дозволяються. Якби ми вирішили використовувати множину, а не мультимножину, то кожне значення виводилося б на друк тільки один раз. Якби ми використовували неупорядковану мультиножину, то порядок проходження елементів залишався б невизначеним.

Приклади використання відображень і мультивідображень

Наступний приклад демонструє використання відображень і мультивідображень:

```
// stl/multimap1.cpp

 #include <map>
 #include <string>
 #include <iostream>
 using namespace std;

 int main()
 {
   multimap<int,string> coll; // контейнер для пар int/string

   // вставляємо деякі елементи в довільному порядку
   // - значення з ключем 1 уставляється двічі
   coll = { {5,"tagged"},
            {2,"a"},
            {1,"this"},
            {4,"of"},
            {6,"strings"},
            {1,"is"},
            {3,"multimap"} };

   // виводимо на друк значення всіх елементів
   // - елемент second є значенням
   for (auto elem : coll) {
        cout << elem.second << ' ';
   }
   cout << endl;
 }
```

Після включення заголовного файлу <map> з'являється відображення з елементами, що мають ключ типу `int` і значення типу `string`.

```
multimap<int,string> coll;
```

Оскільки елементи відображень і мультивідображень являють собою пари “ключ-значення”, оголошення, вставка і доступ до елемента небагато відрізняються.

- По-перше, для ініціалізації (присвоєння чи вставки) елементів необхідно передавати пари “ключ-значення” за допомогою списків ініціалізації. Внутрішні списки визначають ключі і значення кожного елемента; зовнішні списки групують усі ці елементи. Таким чином, інструкція `{5,"tagged"}` описує вставку першого елемента.

- При обробці елементів ми знову працюємо з парами “ключ-значення”. Фактично типом елемента є клас `pair<const` *`ключ, значення`*`>`. Ключ є константою, тому що будь-яка модифікація його значення може порушити порядок проходження елементів, автоматично упорядкованих контейнером. Оскільки об'єкти структури `pair` не мають операції виводу, їх неможливо вивести на екран цілком. Замість цього необхідно одержати доступ до членів структури `pair`, що називаються `first` і `second`.

Таким чином, що наступний вираз створює другу частину пари “ключ-значення”, тобто значення елемента мультивідображення:

```
elem.second
```

Аналогично наступний вираз створює першу частину пари “ключ-значення”, тобто ключ елемента мультивідображення:

```
elem.first
```

У результаті програма виводить на екран такий рядок:

```
this is a multimap of tagged strings
```

До появи стандарту C++11 не було строгої гарантії збереження порядку еквівалентних елементів (тобто елементів, що мають однакові ключі). Отже, до прийняття стандарту C++11 порядок проходження рядків `"this"` і `"is"` міг бути довільним. Стандарт C++11 гарантує, що нові елементи вставляються після еквівалентних елементів, що уже містяться в мультимножинах і мультивідображеннях. Крім того, при виклику функцій `insert()`, `emplace()` чи `erase()` порядок еквівалентних елементів зберігається.

19.2.3. Неупорядковані контейнери

У неупорядкованому контейнері елементи не мають визначеного порядку. Отже, якщо вставити три елементи, то при обході контейнера вони можуть слідувати в будь-якому порядку. Якщо вставити четвертий елемент, то порядок проходження раніше вставлених елементів може змінитися. Єдиним фактом, що має значення, є те, що конкретний елемент знаходиться *десь* у контейнері. Навіть якщо два контейнери містять однакові елементи, їхній порядок може бути різним. Уявіть собі, що це просто мішок.

Неупорядковані контейнери звичайно реалізуються у виді хеш-таблиці. Таким чином, власне кажучи, контейнер — це масив зв'язаних списків. Позиція елемента в масиві обчислюється за допомогою хеш-функції. Мета полягає в тім, щоб кожен елемент мав свою позицію, що забезпечує до нього швидкий доступ, за умови швидкого обчислення хеш-функції. Однак, оскільки швидкі ідеальні хеш-функції не завжди можливі чи можуть зажадати величезний об'єм пам'яті для масиву, дозволяється зберігати в одній і тій же позиції

декілька елементів. З цієї причини елементами масиву є зв'язані списки, що дозволяють зберігати в одній позиції масиву декілька елементів контейнера.

Основна перевага неупорядкованих контейнерів полягає в тім, що пошук елемента, що має конкретне значення, виконується ще швидше, ніж в асоціативному масиві. Фактично використання неупорядкованих контейнерів забезпечує амортизовану константну складність, за умови, що хеш-функція є досить гарною. Однак створити гарну хеш-функцію нелегко, так що може знадобитися багато пам'яті для осередків. Аналогічно асоціативним контейнерам, у бібліотеці STL передбачені наступні такі контейнери.

- **Неупорядкована множина** — колекція неупорядкованих елементів, кожний з який може зберігатися тільки в одному екземплярі. Таким чином, дублікати заборонені.
- **Неупорядкована мультимножина** — це неупорядкована множина, у якій дозволені дублікати. Таким чином, неупорядкована мультимножина може містити декілька елементів, що мають однакові значення.
- **Неупорядковане відображення** — контейнер, що містить пари "ключ-значення". Кожен ключ може зберігатися тільки в одному екземплярі, так що дублікати ключів заборонені. Неупорядковане відображення можна використовувати як *асоціативний масив*, тобто масив з індексом довільного типу.
- **Неупорядковане мультивідображення** — це неупорядковане відображення, у якому дозволені дублікати. Таким чином, неупорядкована **мультимножина** може містити декілька елементів з однаковими ключами. Неупорядковану **мультимножина** можна використовувати як *словник*.

Усі ці класи неупорядкованих контейнерів мають декілька необов'язкових шаблонних аргументів, що задають хеш-функцію і критерій еквівалентності. Критерій еквівалентності використовується для пошуку конкретних значень та ідентифікації дублікатів. За замовчуванням критерієм еквівалентності є операція ==.

Неупорядковану множину можна розглядати як різновид неупорядкованого відображення, у якому значення ідентичне ключу. Зазвичай всі ці типи неупорядкованих контейнерів створюються на основі однієї і тієї ж базової реалізації хеш-таблиці.

Приклади використання неупорядкованих множин і мультимножин

У першому прикладі демонструється використання неупорядкованої мультимножини рядків.

```
// stl/unordmultiset1.cpp

 #include <unordered_set>
 #include <string>
```

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

int main()
{
  unordered_multiset<string> cities {
     "Braunschweig", "Hanover", "Frankfurt", "New York",
     "Chicago", "Toronto", "Paris", "Frankfurt"
  };

  // виводимо на друк кожен елемент:
  for (const auto& elem : cities) {
       cout << elem << " ";
  }
  cout << endl;

  // вставляємо додаткові значення:
  cities.insert( {"London", "Munich", "Hanover", "Braunschweig"} );

  // виводимо на друк кожен елемент:
  for (const auto& elem : cities) {
       cout << elem << " ";
  }
  cout << endl;
}
```

Після включення необхідного заголовного файлу

```
#include <unordered_set>
```

можна оголосити і ініціалізувати неупорядковану множину рядків:

```
unordered_multiset<string> cities { ... };
```

Якщо тепер вивести на друк всі елементи, то порядок може виявитися іншим, тому що він не визначений. Гарантується лише, що дублікати, що дозволяються в мультимножині, будуть згруповані в порядку їх вставки. Таким чином, один з можливих варіантів виводу може виглядати в такий спосіб:

```
Paris Toronto Chicago New York Frankfurt Frankfurt Hanover Braunschweig
```

Будь-яка вставка може змінити цей порядок. Насправді цей порядок може змінити будь-яка операція, зв'язана з повторним хешуванням. Таким чином, після вставки декількох додаткових значень результат може бути таким:

```
London Hanover Hanover Frankfurt Frankfurt New York Chicago Munich Braunschweig
Braunschweig Toronto Paris
```

Наслідки залежать від стратегії повторного хешування, на яку частково може впливати програміст. Наприклад, можна зарезервувати досить багато пам'яті, щоб для її вставки визначеної кількості елементів повторное хешування стало непотрібним. Крім того, для того щоб мати можливість видаляти елементи при їхній обробці, стандарт гарантує, що видалення не вимагає повторного хешування. Однак вставка після видалення може привести до повторного хешування.

У принципі асоціативні і неупорядковані контейнери мають однакові інтерфейси, відрізнятися можуть лише оголошення. При цьому неупорядковані контейнери надають спеціальні функції-члени, що дозволяють впливати на їх поводження чи інспектувати поточний стан. Таким чином, у представленому прикладі лише заголовні файли і типи відрізняються від відповідного приклада, що демонструє використання звичайної мультимножини.

До появи стандарту C++11 для доступу до елементів були потрібні ітератори.

Приклади використання неупорядкованих відображень і мультивідображень

Приклад, що демонструє використання мультивідображень, можна поширити і на неупорядковане мультивідображення, якщо замінити назву `map` на `unordered_map` у директиві `include` і назву `multimap` на `unordered_multimap` в оголошенні контейнера.

```
#include <unordered_map>
...
 unordered_multimap<int,string> coll;
 ...
```

Єдина відмінність полягає в тім, що порядок проходження елементів не визначений. Однак на більшості платформ елементи залишаються упорядкованими, оскільки як хеш-функції за замовчуванням використовується операція ділення за модулем. Таким чином, невизначеність порядку не означає його відсутність. У той же час цей властивість не гарантується, і якщо додати декілька нових елементів, то порядок може змінитися.

Розглянемо інший приклад використання неупорядкованого відображення. У даному випадку ми використовуємо неупорядковане відображення, у якому ключами є рядки, а значеннями — числа типу `double`.

```
// stl/unordmap1.cpp

 #include <unordered_map>
 #include <string>
 #include <iostream>
 using namespace std;
```

```
int main()
{
  unordered_map<string,double> coll { { "tim", 9.9 },
                                      { "struppi", 11.77 }
                                    };

  // підномсимо значення кожного елемента в квадрат:
  for (pair<const string,double>& elem : coll) {
       elem.second *= elem.second;
  }

  // виводимо на друк кожен елемент (ключ і значення):
  for (const auto& elem : coll) {
       cout << elem.first << ": " << elem.second << endl;
  }
}
```

Після звичайних включень заголовних файлів для відображень, рядків і потоків вводу-виводу з'являється неупорядковане відображення, що ініціалізується двома елементами. Тут використовуються вкладені списки ініціалізації, так що відображення ініціалізується двома елементами:

```
{ "tim", 9.9 }
```

і

```
{ "struppi", 11.77 }
```

Потім ми зводимо значення кожного елемента в квадрат.

```
for (pair<const string,double>& elem : coll) {
     elem.second *= elem.second;
}
```

Тут можна побачити внутрішній тип елементів — об'єктів типу `pair<>`, що складаються з константного рядка і чисел типу `double`. Таким чином, модифікувати ключ `first` в елементі неможливо:

```
for (pair<const string,double>& elem : coll) {
     elem.first = ...; // ПОМИЛКА: ключі відображення є константами
}
```

У відповідності зі стандартом C++11 явно вказувати тип елементів необов'язково, оскільки в діапазонному циклі `for` він виводиться з типу контейнера. З цієї причини в другому циклі, що виводить всі елементи на друк, використовується ключове слово `auto`. Фактично, повідомляючи `elem` як `const auto&`, ми уникаємо утворення копій:

```
for (const auto& elem : coll) {
     cout << elem.first << ": " << elem.second << endl;
 }
```

В результаті один з *можливих* результатів роботи програми може виглядати в такий спосіб:

```
struppi: 138.533
tim: 98.01
```

Цей порядок не гарантований, оскільки реальний порядок не визначений. Якби використовувався звичайний контейнер `map`, то порядок елементів був би гарантованим і елемент із ключем `"struppi"` виводився б на екран до елемента з ключем `"tim"`, тому що відображення сортує елементи за ключами, а рядок `"struppi"` передує рядку `"tim"`.

19.2.4. Асоціативні масиви

Відображення і неупорядковані відображення є колекціями пар "ключ-значення" з унікальними ключами. Такі колекції можна інтерпретувати як *асоціативний масив*, тобто масив, індекс якого не є цілим числом. У результаті обидва контейнери допускають використання операції індексування `[ ]`.

Розглянемо наступний приклад:

```
// stl/assoarray1.cpp

 #include <unordered_map>
 #include <string>
 #include <iostream>
 using namespace std;

 int main()
 {
   // тип контейнера:
   // - unordered_map: елементами є пари ключ/значення
   // - string: ключі мають тип string
   // - float: значення мають тип float
   unordered_map<string,float> coll;

   // вставляємо елементи в колекцію
   // - використовуючи синтаксис асоціативного масиву
   coll["VAT1"] = 0.16;
   coll["VAT2"] = 0.07;
   coll["Pi"] = 3.1415;
   coll["an arbitrary number"] = 4983.223;
```

```
    coll["Null"] = 0;

    // змінюємо значення
    coll["VAT1"] += 0.03;

    // виводимо на друк різниці між значеннями VAT
    cout << "VAT difference: " << coll["VAT1"] - coll["VAT2"] << endl;
}
```

Оголошення контейнерного типу повинне задавати тип ключа і значення:

```
unordered_map<string,float> coll;
```

Це оголошення означає, що ключами є рядки, а зв'язані з ними значення є числами з точкою, що плаває.

Відповідно до концепції асоціативних масивів доступ до елементів забезпечує операція індексування `[ ]`. Відзначимо, однак, що ця операція індексування не схожа на звичайне індексування масивів: якщо заданому індексу не відповідає жодний елемент, то це *не* помилка. Новий індекс (чи ключ) ініціює створення і вставку нового елемента відображення, у якого даний індекс є ключем. Таким чином, індекс ніколи не буває неправильним.

Отже, у нашому прикладі оператор

```
coll["VAT1"] = 0.16;
```

створює новий елемент, що має ключ `"VAT1"` і значення `0.16`.

Наступнт вираз створює новий елемент, що має ключ `"VAT1"`, і ініціалізує його типом, заданим за замовчуванням (використовуючи конструктор за замовчуванням чи нуль для фундаментальних типів даних):

```
coll["VAT1"]
```

Вираз в цілому забезпечує доступ до значення нового елемента, так що оператор присвоювання привласнює йому `0.16`.

У відповідності зі стандартом C++11 як альтернативу можна використовувати функцію `at()` для доступу до значень елементів по переданому ключі. Якщо ключ не буде знайдений, буде сгенеровано винятки `out_of_range`.

```
coll.at("VAT1") = 0.16; // винятки out_of_range при відсутності елемента
```

Такі вираз, як

```
coll["VAT1"] += 0.03;
```

чи

```
coll["VAT1"] - coll["VAT2"]
```

забезпечують читання і запис значень зазначених елементів. Таким чином, результат програми буде мати наступний вид:

```
VAT difference: 0.12
```

Зазвичай, різниця між неупорядкованим і традиційним відображенням полягає в тім, що елементи в неупорядкованому відображенні слідують у довільному порядку, у той час як елементи у відображенні упорядковані. Однак доступ до елемента в неупорядкованому відображенні має амортизовану константну складність, а у відображенні — логарифмічну складність. З цієї причини звичайно варто віддавати перевагу неупорядкованим відображенням, якщо тільки вам не потрібне сортування чи ви не можете використовувати неупорядковане відображення через те, що оточення не підтримує властивості стандарту C++11. У цьому випадку варто просто змінити тип контейнера: видаліть префікс "`unordered_`" у директиві `include` і оголошенні контейнера.

19.2.5. Інші контейнери

Рядки

Рядки також можна використовувати в якості STL контейнерів. Під *рядками* маються на увазі об'єкти класів рядків у мові C++ (`basic_string<>`, `string`, і `wstring`). Рядки схожі на вектори, але як елементи містять символи.

Звичайні масиви в стилі мови C

Інший вид контейнерів стосується скоріше до ядра мов C і C++, а не до класів. Це звичайний масив ("масив у стилі C"), що з'являється з фіксованим чи динамічним розміром, керованим функціями `malloc()` і `realloc()`. Однак звичайні масиви не належать до STL контейнерів, оскільки вони не мають функції-члени, такі як `size()` і `empty()`. Проте архітектура бібліотеки STL дозволяє застосовувати до них алгоритми.

Використання звичайних масивів не являє собою нічого нового. Новим є застосування до них алгоритмів. У мові C++ більше не обов'язково програмувати масиви в стилі мови C. Вектори й об'єкти класу `array` забезпечують усі властивості звичайних масивів, але є більш безпечними і зручними.

Користувальницькі контейнери

У принципі програміст може створити будь-який об'єкт із властивостями контейнера інтерфейсом, що відповідають вимогам бібліотеки STL, що дозволить обходити елементи чи виконувати стандартні операції над їх вмістом. Наприклад, можна написати клас для каталогу, у якому можна обходити файли і маніпулювати ними. Найкращими кандидатами на включення в STL-подібні інтерфейси є звичайні операції над контейнерами.

Однак деякі об'єкти, що нагадують контейнери, не відповідають концепції бібліотеки STL. Наприклад, той факт, що кожен контейнер у бібліотеці STL має початок і кінець, утрудняє реалізацію циклічних контейнерів, таких як кільцевий буфер, у стилі бібліотеки STL.

19.2.6. Адаптери контейнерів

Крім основних контейнерних класів, у стандартній бібліотеці C++ передбачені *адаптери контейнерів*, що являють собою стандартні контейнери, що надають обмежений інтерфейс для спеціальних задач. Ці адаптери контейнерів реалізуються на основі фундаментальних контейнерних класів. Стандартні адаптери контейнерів перераховані в наступному списку.

- **Стек** керує своїми елементами за принципом LIFO (останнім увійшов — першим вийшов).
- **Черга** керує своїми елементами за принципом FIFO (першим увійшов — першим вийшов). Інакше кажучи, це звичайний буфер.
- **Черга з пріоритетами —** це контейнер, у якому елементи мають різні пріоритети. Пріоритет залежить від критерію сортування, що може задавати програміст (за замовчуванням використовується операція <). Черга з пріоритетами власне кажучи являє собою буфер, у якому черговий елемент завжди має найвищий пріоритет у черзі. Якщо найвищий пріоритет мають кілька елементів, то порядок проходження цих елементів не визначений.

Адаптери контейнерів історично є частиною бібліотеки STL. Однак з погляду програмування вони являють собою усього лише спеціальну категорію контейнерних класів, що використовують загальну архітектуру контейнерів, ітераторів і алгоритмів бібліотеки STL.

Література

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