Глава 7

Отсортированные ассоциативные контейнеры

Теперь мы перейдем к отсортированным ассоциативным контейнерам, которые представляют собой еще одно семейство абстракций, отличающееся от уже рассмотренного в главе 6, “Последовательные контейнеры”. В то время как последовательные контейнеры хранят свои данные линейно, с сохранением относительных позиций, в которые были вставлены элементы, отсортированные ассоциативные контейнеры обходятся без этого порядка, а вместо этого сосредотачиваются на максимальной скорости выборки на основе *ключей*, которые хранятся в элементе (или, в некоторых случаях, представляют собой сам элемент).

Один общий подход к ассоциативной выборке состоит в хранении ключей в отсортированном состоянии в соответствии с некоторым глобальным упорядочением, таким как числовой порядок или лексикографический, если ключи являются строками, и использовании бинарного алгоритма поиска. Еще одним подходом является хеширование: разделение пространства ключей на несколько подмножеств и вставка каждого ключа в предназначенное ему подмножество; после этого поиск выполняется только в одном подмножестве. Каждый ключ связан со своим подмножеством при помощи так называемой хеш-функции. Первый подход — *отсортированные ассоциативные контейнеры* — можно реализовать в том числе с использованием сбалансированных бинарных деревьев поиска, а последний — *хешированные ассоциативные контейнеры* — при помощи любого из множества представлений хеш-таблиц.

Основным преимуществом хеширования является константное среднее время вставки и выборки, а подхода с сортировкой — в гарантированности производительности. (Производительность операций с хеш-таблицами в худшем случае может быть очень низкой — линейно зависящей от размера N, но для сбалансированных бинарных деревьев она всегда составляет $O(\log N)$.)

В идеале в стандартной библиотеке C++ должны были бы быть и отсортированные, и хешированные ассоциативные контейнеры, но в нее оказались включены только отсортированные ассоциативные контейнеры (имеются неофициальные спецификации хеш-таблиц и их реализации; см. первую сноску в главе 6, “Последовательные контейнеры”, на с. 138).

Отсортированными ассоциативными контейнерами STL являются классы `set`, `multiset`, `map` и `multimap`. В случае множеств и мультимножеств элементами данных являются сами ключи, причем мультимножество допускает наличие одинаковых ключей, а множество — нет. В отображениях и мультиотображениях элементы данных представляют собой пары, состоящие из ключей и собственно данных некоторого другого типа, причем мультиотображение допускает наличие одинаковых ключей, а отображение — нет. Мы рас-

сматриваем множества и мультимножества в разделе 7.1, а отображения и мультиотображения — в разделе 7.2.

Хотя их фундаментальная природа и различна, отсортированные ассоциативные контейнеры обладают многими свойствами, присущими последовательным контейнерам, поскольку они поддерживают обход элементов данных в виде линейной последовательности, с применением таких же аксессоров контейнеров, что и у последовательных контейнеров. Они предоставляют двунаправленные итераторы, обход с использованием которых дает отсортированный порядок элементов. Фактически в некоторых случаях (например, когда элементы данных представляют собой большие структуры) сортировка последовательности элементов может быть выполнена более эффективно путем вставки их в мультимножество и обхода мультимножества, чем обобщенным алгоритмом сортировки или соответствующей функцией-членом списка.

7.1. Множества и мультимножества

7.1.1. Типы

Параметры шаблонов `set` и `multiset` задаются как

```
template <typename Key, typename Compare = less<Key>,
          class Allocator = allocator<Key> >
```

Первый параметр представляет собой тип хранимых ключей, второй — тип функции сравнения, используемой для определения порядка, а третий — тип используемого аллокатора.

Оба класса предоставляют определения тех же типов, что и в последовательных контейнерах — `value_type` (определен как тип `Key`), `size_type`, `difference_type`, `reference`, `pointer`, `iterator`, `reverse_iterator`, `const_reference`, `const_pointer`, `const_iterator` и `const_reverse_iterator`, — а также:

- `key_type` — определен как тип `Key`;
- `key_compare` — определен как тип `Compare`;
- `value_compare` — определен как тип `Compare`.

Эти классы определяют как `key_compare`, так и `value_compare` для совместимости с `map` и `multimap`, где они представляют собой разные типы.

Функция сравнения должна определять отношение упорядочения для ключей, которое используется для определения их порядка при линейном обходе контейнера при помощи поддерживаемых итераторов. Требования к данному отношению те же, что и уже рассматривались в разделе 5.4 для отношений, используемых алгоритмами STL, связанными с сортировкой. Отношение упорядочения используется также при выяснении эквивалентности двух ключей. Два ключа, `k1` и `k2`, рассматриваются как эквивалентные, если

```
key_compare(k1, k2) == false && key_compare(k2, k1) == false
```

В большинстве простых случаев такое определение эквивалентности соответствует оператору `==`. Например, если ключи представляют собой любые встроенные числовые типы и `key_compare(k1, k2) == (k1 < k2)`, то определение эквивалентности сводится к выражению

```
!(k1 < k2) && !(k2 < k1)
```

которое является тем же отношением, что и вычисляемое как `k1 == k2`. Однако, конечно же, возможны и несогласованности этих определений. В качестве тривиального примера предположим, что ключи представляют собой контейнеры `vector<int>`, и мы определяем

key_compare как сравнение только их первых элементов. Тогда любые два вектора, у которых первые элементы одинаковы, рассматриваются key_compare как эквивалентные, но с точки зрения оператора == эти векторы различны, если у них имеются несовпадения в некоторой другой позиции.

Важно помнить об этом (потенциальном) отличии, поскольку, как будет показано позже, понятие эквивалентности ключей в отсортированных ассоциативных контейнерах используется по-разному, в частности, в зависимости от того, должен элемент быть вставлен в множество, исходя из наличия в нем дубликатов (т.е. эквивалентных элементов), или выполняется ли поиск в мультимножестве всех ключей, эквивалентных заданному. В любом случае, когда мы говорим об эквивалентности ключей, мы имеем в виду приведенное здесь определение, а не оператор ==.

Итераторы множества и мультимножества представляют собой двунаправленные итераторы, а не итераторы с произвольным доступом, так что некоторые обобщенные алгоритмы, такие как алгоритмы сортировки, к множествам и мультимножествам не применимы. Однако алгоритмы сортировки к ним и не требуется применять, поскольку эти контейнеры постоянно поддерживают свое содержимое отсортированным.

7.1.2. Конструкторы

Конструкторами множества являются

```
set(const Compare& comp = Compare());

template <typename InputIterator>
set(InputIterator first, InputIterator last,
    const Compare& comp = Compare());

set(const set<Key, Compare, Allocator>& otherSet);
```

Первый из них создает пустое множество, второй — множество с копиями элементов из диапазона [first;last) (с удаленными дублями), а третий (копирующий конструктор) — множество с теми же элементами, что и у otherSet. Первый и второй конструкторы получают необязательный второй параметр типа Compare, по умолчанию равный Compare(). Конструкторы мультимножества имеют тот же вид и тот же смысл, за исключением того, что они не отбрасывают дублирующиеся элементы.

7.1.3. Вставка

Простейшая функция-член insert классов set и multiset получает единственный аргумент типа value_type, который представляет собой тип Key, и вставляет копию аргумента.

Пример 7.1. Демонстрация создания множества и вставки в него

```
"ex07-01.cpp" 165 ≡
  #include <iostream>
  #include <cassert>
  #include <list>
  #include <string>
  #include <set>
  using namespace std;

  <Функция make (создание контейнера символов) 53б>
  int main()
  {
      cout << "Демонстрация создания множества "
```

```
        << "и вставки в него." << endl;
    list<char> list1 =
    make< list<char> >("There is no distinctly native "
                       "American criminal class");

    // Размещение символов из list1 в set1:
    set<char> set1;
    list<char>::iterator i;
    for (i = list1.begin(); i != list1.end(); ++i)
        set1.insert(*i);

    // Размещение символов из set1 в list2:
    list<char> list2;
    set<char>::iterator k;
    for (k = set1.begin(); k != set1.end(); ++k)
        list2.push_back(*k);

    assert (list2 ==
            make< list<char> >(" ATacdehilmnorstvy"));
    return 0;
}
```

После вставки символов `list1` в `set1` мы изучаем содержимое множества, копируя его в `list2` с помощью цикла с переменной итераторного типа `set<char>::iterator` и функциями-членами `begin` и `end`, которые отсортированные ассоциативные контейнеры предоставляют точно так же, как и последовательные контейнеры. Вызов `assert` показывает, что последовательность символов в `list2`, а значит, и в `set1`, является отсортированной и не содержит дублей.

Вот программа, которая выполняет те же действия с мультимножествами вместо множеств.

Пример 7.2. Демонстрация создания мультимножества и вставки в него

```
"ex07-02.cpp" 166 ≡
  #include <iostream>
  #include <cassert>
  #include <list>
  #include <string>
  #include <set>
  using namespace std;

  <Функция make (создание контейнера символов) 53б>

  int main()
  {
      cout << "Демонстрация создания мультимножества "
           << "и вставки в него." << endl;
      list<char> list1 =
          make<list<char> >("There is no distinctly native "
                            "American criminal class");

      // Размещение символов из list1 в multiset1:
      multiset<char> multiset1;
      list<char>::iterator i;
      for (i = list1.begin(); i != list1.end(); ++i)
          multiset1.insert(*i);

      // Размещение символов из multiset1 в list2:
      list<char> list2;
```

```
    multiset<char>::iterator k;
    for (k = multiset1.begin(); k != multiset1.end(); ++k)
        list2.push_back(*k);

    assert (list2 ==
        make< list<char> >(" ATaaaaccccdeeeehiiiiiii"
                           "lllmmnnnnnorrrssssttttvy"));
    cout << " --- Ok." << endl;
    return 0;
}
```

В этом случае вызов `assert` показывает наличие дубликатов.

Функция-член `insert`, использовавшаяся в этих примерах, имеет разные возвращаемые типы в классах `set` и `multiset`. В классе `set` ее интерфейс представляет собой

```
pair<iterator, bool> insert(const value_type& x);
```

в то время как в классе `multiset` он имеет вид

```
iterator insert(const value_type& x);
```

В версии `set` возвращаемый итератор указывает на позицию `x` во множестве, независимо от того, был такой элемент во множестве или нет, а значение `bool` равно `true`, если элемент был вставлен, и `false`, если он уже имелся во множестве. В версии `multiset` элемент всегда вставляется, так что значение типа `bool` излишнее.

Время работы `insert` равно $O(\log N)$, где N — количество элементов, хранящихся во множестве или мультимножестве.

Функция-член `insert` отличается от функций-членов `insert` последовательных контейнеров тем, что ей не передается итератор, указывающий, куда следует вставить новый элемент. Здесь позиция нового элемента вычисляется так, чтобы поддерживалась упорядоченность элементов. Имеется, однако, еще одна функция-член `insert` множества и мультимножества, которая получает аргумент-итератор и таким образом имеет тот же интерфейс, что и в последовательных контейнерах:

```
iterator insert(iterator position, const value_type& x);
```

Эта функция все равно помещает `x` в позицию, диктуемую требованием упорядоченности содержимого, рассматривая `position` исключительно как "подсказку", где именно начинать поиск. Она имеет константное амортизированное время работы, если `x` вставляется или уже присутствует в позиции `position`. Простая ситуация, когда используется эта возможность, — это копирование некоторого уже отсортированного контейнера во множество или мультимножество, например:

```
vector<int>::iterator i;
for (i = vector1.begin(); i != vector1.end(); ++i)
    set1.insert(set1.end(), *i);
```

Этот код работает независимо от того, отсортированы ли элементы `vector1`, требуя $O(N \log N)$ времени (где N есть `vector1.size()`), если они не отсортированы, и только $O(N)$ в противном случае.

Поскольку итератор вставки, получающийся в результате вызова `inserter`, работает посредством функции `insert`, предыдущий код можно переписать в виде

```
copy(vector1.begin(), vector1.end(),
     inserter(set1, set1.end()));
```

В общем случае версия с "подсказкой" функции-члена `insert` для множеств и мультимножеств полезна при работе в связке с обобщенными алгоритмами для теоретико-

множественных операций над отсортированными структурами (includes, set_union, set_intersection, set_difference и set_symmetric_difference). Например, код

```
set_union(set1.begin(), set1.end(),
          set2.begin(), set2.end(),
          inserter(set3, set3.end()));
```

помещает объединение set1 и set2 в set3 за время $O(N_1 + N_2)$, где N_1 и N_2 — размеры set1 и set2.

У классов множеств и мультимножеств имеется еще одна функция-член insert, предназначенная для вставки элементов из диапазона:

```
template <typename InputIterator>
void insert(InputIterator first, InputIterator last);
```

Время ее работы — $O(M \log(N + M))$, где M — размер диапазона, а N — размер множества или мультимножества.

7.1.4. Удаление

Элементы из множества или мультимножества могут быть удалены либо по ключу, либо по позиции: для удаления всех элементов с ключом k из множества или мультимножества set1 используется выражение

```
set1.erase(k);
```

Если set1 является множеством, то в нем может быть не более одного элемента с ключом k; если такого элемента нет, этот вызов функции не выполняет никаких действий. Если i — корректный разыменуемый итератор set1, то выражение

```
set1.erase(i);
```

удаляет элемент, на который указывает i. В случае, когда set1 представляет собой мультимножество и в нем имеются другие копии *i, будет удалена только та копия, на которую указывает i. Для иллюстрации сказанного в приведенной далее программе определим новую функцию make_string, преобразующую любой контейнер символов STL в строку.

```
<Функция make_string (создание строки из контейнера) 168а> ≡
  #include <functional>
  template <typename Container>
  string make_string(const Container& c)
  {
      string s;
      copy(c.begin(), c.end(), inserter(s, s.end()));
      return s;
  }
```

```
Используется в частях 168б, 170.
```

В данном примере для поиска элемента по ключу используется один из аксессоров мультимножества, find.

Пример 7.3. Демонстрация функций erase мультимножества

```
"ex07-03.cpp" 168б ≡
  #include <iostream>
  #include <cassert>
  #include <list>
  #include <string>
```

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

<Функция make (создание контейнера символов) 53б>
<Функция make_string (создание строки из контейнера) 168а>

int main()
{
    cout << "Демонстрация функций erase мультимножества"
         << endl;
    list<char> list1 =
    make< list<char> >("There is no distinctly native "
                       "American criminal class");

    // Размещение символов list1 в multiset1:
    multiset<char> multiset1;
    copy(list1.begin(), list1.end(),
         inserter(multiset1, multiset1.end()));
    assert (make_string(multiset1) ==
            "       ATaaaaccccdeeeehiiiiiiilll"
            "mmnnnnnorrrssssttttvy");
    multiset1.erase('a');
    assert (make_string(multiset1) ==
            "       ATccccdeeeehiiiiiiilll"
            "mmnnnnnorrrssssttttvy");

    multiset<char>::iterator i = multiset1.find('e');

    multiset1.erase(i);
    assert (make_string(multiset1) ==
            "       ATccccdeeehiiiiiiilll"
            "mmnnnnnorrrssssttttvy");
    cout << " --- Ok." << endl;
    return 0;
}
```

Первый вызов `erase` получает ключ `a` и из `multiset1` удаляются все элементы с этим ключом. Второй вызов `erase` получает итератор `i`, который возвращает вызов `find` и который указывает на элемент, содержащий один из символов `e`, что приводит к тому, что удаляется только один этот элемент. Кстати, можно объединить вызовы `find` и `erase` в одной строке:

```
multiset1.erase(multiset1.find('e'));
```

Так можно избежать объявления переменной-итератора.

Классы `set` и `multiset` предоставляют еще одну функцию-член `erase`, которая удаляет все элементы из диапазона. Например, к концу приведенной программы можно добавить следующие строки:

```
i = multiset1.find('T');
multiset<char>::iterator j = multiset1.find('v');

multiset1.erase(i, j);
assert (make_string(multiset1) == " Avy");
```

Все функции-члены `erase` имеют время работы $O(\log N + E)$, где N — размер множества или мультимножества, а E — количество удаляемых элементов.

7.1.5. Аксессоры

Множества и мультимножества в основном имеют те же аксессоры, что и последовательные контейнеры, использующиеся для получения информации об элементах линейной отсортированной последовательности: `begin`, `end`, `rbegin`, `rend`, `empty`, `size` и `max_size`. Как обычно, аксессоры, возвращающие итераторы, имеют версии, возвращающие константные итераторы при применении их к константным множествам или мультимножествам. Однако эти контейнеры имеют ряд функций-членов для получения информации по ключу: `find`, `lower_bound`, `upper_bound`, `equal_range` и `count`.

Примеры использования `find` были приведены в разделе 7.1.4. Важно понимать отличие между этой функцией-членом и обобщенным алгоритмом `find`. Как указывалось в разделе 4.12, основное отличие заключается в эффективности: функция-член `find` множества или мультимножества выполняется за время $O(\log N)$, где N — размер контейнера, а обобщенный алгоритм `find` — за время $O(N)$. Функция-член `find` гораздо эффективнее потому, что использует тот факт, что ключи находятся в отсортированном порядке, так что можно выполнить бинарный поиск, в то время как обобщенный алгоритм `find` выполняет линейный поиск.

Еще одно отличие между функцией-членом `find` и обобщенным алгоритмом `find` в случае мультимножеств заключается в элементе, на который указывает возвращаемый итератор. Обобщенный алгоритм `find` возвращает позицию первого члена с данным ключом, но в случае функции-члена `find` не указано, какой именно из имеющихся в мультимножестве одинаковых ключей возвращается. Если важно получить первую позицию, то следует использовать функцию-член `lower_bound`, а функция-член `upper_bound` возвращает итератор, указывающий на позицию, следующую за концом диапазона позиций, содержащих интересующий нас ключ.

Пример 7.4. Демонстрация функций-членов мультимножества для поиска

```
"ex07-04.cpp" 170 ≡
  #include <iostream>
  #include <cassert>
  #include <list>
  #include <string>
  #include <set>
  using namespace std;

  <Функция make (создание контейнера символов) 53б>
  <Функция make_string (создание строки из контейнера) 168a>

  int main()
  {
      cout << "Демонстрация функций-членов "
           << "мультимножества для поиска." << endl;
      list<char> list1 =
          make<list<char> >("There is no distinctly native "
                            "American criminal class"),
      list2 =
          make<list<char> >("except Congress. - Mark Twain");

      // Размещение символов list1 в multiset1:
      multiset<char> multiset1;
      copy(list1.begin(), list1.end(),
           inserter(multiset1, multiset1.end()));

      assert (make_string(multiset1) ==
```

```
                "       ATaaaaccccdeeeehiiiiiiilll"
                "mmnnnnnorrrssssttttvy");

    multiset<char>::iterator
            i = multiset1.lower_bound('c'),
            j = multiset1.upper_bound('r');
    multiset1.erase(i, j);
    assert (make_string(multiset1) ==
                "       ATaaaasssstttvy");

    list<char> found, not_found;
    list<char>::iterator k;
    for (k = list2.begin(); k != list2.end(); ++k)
        if (multiset1.find(*k) != multiset1.end())
            found.push_back(*k);
        else
            not_found.push_back(*k);

    assert (found == make< list<char> >("t ss  a Ta"));
    assert (not_found ==
            make< list<char> >("excepCongre.-Mrkwin"));

    cout << " --- Ok." << endl;
    return 0;
}
```

Можно рассматривать `lower_bound` как функцию, возвращающую первую позицию, куда может быть вставлен ключ так, чтобы сохранилась отсортированность последовательности, а `upper_bound` — как возвращающую последнюю такую позицию. Эти утверждения истинны независимо от наличия ключа в контейнере. (Это тот же смысл, который имеют обобщенные алгоритмы `lower_bound` и `upper_bound` при работе с отсортированными последовательностями.)

Последняя часть программы выполняет поиск в `multiset1` каждого из символов `list2` и помещает найденные символы в список `found`, а не найденные — в `not_found`.

Если нам нужны результаты `lower_bound` и `upper_bound` для одного и того же ключа, их можно получить одновременно, вызвав функцию `equal_range`, которая возвращает пару итераторов. Свое имя данная функция получила потому, что возвращаемые ею итераторы определяют диапазон элементов, эквивалентных данному ключу. Этот диапазон пустой (итераторы равны), если искомого элемента в контейнере нет.

Наконец, `count` возвращает расстояние от позиции `lower_bound` до позиции `upper_bound`, которое равно количеству элементов, эквивалентных данному ключу. В качестве примера использования `equal_range` и `count` можно добавить в конец предыдущей программы следующие строки:

```
assert (make_string(multiset1) ==
            "       ATaaaasssstttvy");
i = multiset1.lower_bound('s');
j = multiset1.upper_bound('s');

pair<multiset<char>::iterator,
     multiset<char>::iterator>
        p = multiset1.equal_range('s');

assert (p.first == i && p.second == j);
assert (multiset1.count('s') == 4);

multiset1.erase(p.first, p.second);
```

```
assert (multiset1.count('s') == 0);
assert (make_string(multiset1) ==
            "       ATaaaatttvy");
```

Каждая функция поиска (за исключением `count`) имеет версию, возвращающую константный итератор (или, в случае `equal_range`, пару константных итераторов) при применении к константному контейнеру. Заметим, что хотя мы иллюстрировали применение функций только к мультимножествам, каждая из них определена и для множеств. В случае множеств функция-член `count` всегда возвращает либо `0`, либо `1`.

При работе с функциями поиска следует не забывать, что их смысл зависит от смысла эквивалентности, которая определяется в терминах функции `key_compare`, а не оператором `==`.

Все функции-члены для поиска выполняются за время $O(\log N)$, где N — размер множества или мультимножества, за исключением `count`, которой требуется время $O(\log N + E)$, где E — количество элементов с заданным ключом.

7.1.6. Отношения эквивалентности и “меньше, чем”

Вспомним определение эквивалентности контейнеров, которое применимо ко всем контейнерам STL (см. раздел 6.1.6):

- содержащиеся в них последовательности должны иметь одинаковую длину;
- элементы в соответствующих позициях должны быть эквивалентны, что определяется оператором `==` типа элементов.

При применении ко множествам и мультимножествам STL с их свойством содержания элементов в отсортированном порядке это определение отвечает нашим ожиданиям, основанным на понятиях “множества” и “мультимножества” из математики. То есть, два множества эквивалентны, если содержат одни и те же элементы — не важно, в каком именно порядке. Два мультимножества эквивалентны, если содержат одинаковое количество каждого элемента, опять же независимо от их порядка. Вместе с обобщенными алгоритмами STL для выполнения теоретико-множественных операций над отсортированными структурами (`includes`, `set_union`, `set_intersection`, `set_difference` и `set_symmetric_difference`) эти свойства обеспечивают возможность выполнения всех основных и наиболее полезных вычислений со множествами и мультимножествами.

В действительности все не так просто, поскольку понятие “одинаковости” в данном случае представляет собой не что иное, как отношение эквивалентности, определяемое функцией `key_compare`: два ключа `k1` и `k2` рассматриваются как эквивалентные, если

```
key_compare(k1, k2) == false && key_compare(k2, k1) == false
```

Как упоминалось в разделе 7.1.1, это определение не всегда идентично отношению, вычисляемому оператором `==` элементов. Если эти два отношения не согласованы, можно получить несколько неожиданные результаты при проверке эквивалентности множеств и мультимножеств.

Аналогичная проблема имеется и для оператора `operator<` для множеств и мультимножеств, поскольку общее определение этого оператора для всех контейнеров использует обобщенный алгоритм `lexicographical_compare`, который, в свою очередь, использует оператор `<`, определенный для типа элементов. Если `key_compare` не сравнивает ключи так же, как и оператор `<` для этого типа, сравнение множеств и мультимножеств может дать не тот результат, который вы ожидаете.

7.1.7. Присваивание

Как уже упоминалось в разделе 6.1.7, `operator=` определен для всех контейнеров STL, так что `x = y` делает истинным `x == y`. Это операция с линейным временем работы. У отсортированных ассоциативных контейнеров функции-члена `assign` не имеется, но есть функция-член `swap`, обменивающая при помощи вызова `x.swap(y)` значения `x` и `y` за константное время. Существует соответствующая специализация обобщенного алгоритма `swap`, вызывающая использование соответствующей функции-члена `swap`.

7.2. Отображения и мультиотображения

Контейнеры отображений могут рассматриваться как массивы, индексированные ключами некоторого произвольного типа `Key`, а не целыми числами $0,1,2,\ldots$. Они представляют собой отсортированные ассоциативные контейнеры, обеспечивающие быструю выборку информации некоторого типа `T` на основе ключей другого типа `Key`, причем все сохраненные ключи единственны. Мультиотображения обладают той же функциональностью, но допускают дублирующиеся ключи. Отношения мультиотображений и отображений те же, что и отношения мультимножеств и множеств.

Отображения и мультиотображения можно рассматривать как множества и мультимножества, у которых с каждым ключом хранится дополнительная информация типа `T`. Эти дополнительные данные не влияют на способ поиска информации в контейнере или обхода его элементов и играют роль только в некоторых дополнительных операциях, предназначенных специально для их хранения и выборки. Наше рассмотрение отображений и мультиотображений основывается на этом соображении, а потому будет достаточно коротким.

Конечно, можно было бы начать рассмотрение отсортированных ассоциативных контейнеров с отображений и мультиотображений, а затем заявить, что множества и мультимножества представляют собой отображения и мультиотображения, в которых не используются дополнительные данные. Возможно, это лучший способ представления, поскольку отображения и мультиотображения имеют более широкое применение, чем множества и мультимножества, но мы решили начать именно с последних, так как они немного проще и их легче проиллюстрировать маленькими программами-примерами. В этой главе мы представим только несколько примеров отображений и мультиотображений, но вы еще встретитесь с ними в главах 14, “Улучшенная программа поиска групп анаграмм”, 15, “Ускорение программы поиска анаграмм: использование мультиотображений”, 18, “Программа вывода дерева ученых в области теории вычислительных машин и систем”, и 19, “Класс для хронометража обобщенных алгоритмов”.

Возвращаясь к аналогии, с которой мы начали, массивы (а также векторы и деки) предоставляют возможности отображений, ключи которых являются целыми числами из некоторого диапазона $0,1,2,\ldots,N-1$ для некоторого неотрицательного целого N. При высокой эффективности они достаточно ограничены по сравнению с контейнерами отображений, во-первых тем, что позволяют применять в качестве ключей только целые числа, а во-вторых, тем, что требуют большого количества памяти, пропорционального N, даже если связанные с ними данные имеет лишь небольшой процент ключей. Отображения и мультиотображения преодолевают оба ограничения, допуская ключи почти любого типа и используя количество памяти, пропорциональное количеству фактически сохраненных ключей.

Даже с целочисленными ключами “разреженное представление”, обеспечиваемое отображениями и мультиотображениями, может существенно сэкономить память и время вычислений, как будет показано в разделе 7.2.3 в примере вычисления скалярного произведения двух разреженных векторов (где под вектором мы подразумеваем кортеж из N чисел; он будет представлен как `map`, а не `vector`). К примерам отображений и мультиотображений с неце-

лыми ключами мы перейдем в главах 14, “Улучшенная программа поиска групп анаграмм”, и 15, “Ускорение программы поиска анаграмм: использование мультиотображений”.

7.2.1. Типы

Шаблонные параметры `map` и `multimap` задаются следующим образом:

```
template <typename Key, typename T,
          typename Compare = less<Key>,
          class Allocator = allocator<pair<const Key, T> > >
```

Первый параметр представляет собой тип хранимых ключей, второй — тип объектов, ассоциированных с ключами, третий — функцию сравнения, используемую для определения порядка элементов, а четвертый — используемый аллокатор. Это те же параметры, что и у множеств и мультимножеств, за исключением дополнительного параметра `T`.

Оба класса предоставляют определения тех же типов, что и последовательные контейнеры: `value_type` (который определен как тип `pair<const Key, T>`), `pointer`, `reference`, `iterator`, `reverse_iterator`, `difference_type`, `size_type`, `const_iterator`, `const_reverse_iterator`, `const_pointer` и `const_reference`, а также следующие:

- `key_type` — определен как тип `Key`;
- `key_compare` — определен как тип `Compare`;
- `value_compare` — функциональный тип, определенный для сравнения двух объектов `value_type` на основе их ключей:

```
class value_compare
  : public binary_function<value_type, value_type, bool>
{
protected:
    Compare comp;
    value_compare(Compare c) : comp(c) { }
public:
    bool operator()(const value_type& x,
                    const value_type& y) const {
            return comp(x.first, y.first);
    }
};
```

Как и в случае множеств и мультимножеств, типы итераторов отображений и мультиотображений двунаправленные и произвольный доступ не поддерживают.

7.2.2. Конструкторы

Конструкторы отображений и мультиотображений имеют тот же вид, что и конструкторы множеств и мультимножеств, и то же отличие друг от друга: при создании отображения из диапазона дублирующиеся ключи удаляются, а при создании мультиотображения — остаются в нем. Временны*е* границы те же, что и для конструкторов множеств и мультимножеств.

7.2.3. Вставка

Вставки в отображения и мультиотображения могут выполняться при помощи функций-членов `insert`, которые имеют те же интерфейсы, что и соответствующие функции множеств и мультимножеств. Заметим, однако, что здесь `value_type` представляет собой не просто `Key`, а `pair<const Key, T>`; т.е., один аргумент типа `value_type`, переданный функции-члену `insert`, содержит как значение типа `Key`, так и типа `T`. Уже вставленные пары можно изменять только присваиванием новых значений их членам `T`; спецификатор

`const` перед `Key` предотвращает изменение ключа. Это ограничение критично для целостности внутреннего представления отображения.

Еще один способ вставки в отображение использует `operator[]`:

```
map1[k] = t;
```

Если в `map1` нет пары с ключом `k`, то в него будет вставлена пара `(k,t)`. Если же в нем уже имеется пара `(k,t0)` с некоторым значением `t0`, то `t0` заменяется на `t`. Еще один способ выполнить такую замену (отклонимся на секунду от темы вставки) — воспользоваться выражением

```
i->second = t;
```

где `i` представляет собой `map<Key, T>::iterator` такой, что `i->first == k`. (Но мы не можем записать `i->first = k1`, поскольку `i->first` является константным членом `value type`.)

Возвращаясь к `operator[]`, следует сказать, что он *не* определен для мультиотображений. Это может показаться сюрпризом, но если бы он был определен, то во многих случаях сюрпризы были бы бо́льшими и существенно неприятнее. Связанное значение, например, должно было бы обновляться для всех пар (ключ, значение) с одинаковыми ключами, чтобы обеспечить выполнение требования, согласно которому присваивание `multimap1[k] = v` гарантирует сразу же после его выполнения равенство `multimap1[k] == v`. Такой эффект вряд ли желателен в большинстве случаев.

Но даже в случае отображений может возникнуть неожиданная ситуация: `operator[]` может привести к вставке в отображение просто потому, что он имеется в выражении, причем необязательно в левой части присваивания. То есть, наличие

```
map1[k]
```

в любом выражении возвращает значение `T`, связанное с ключом `k`, если таковой имеется; но если такого ключа в отображении не имеется, то в него будет вставлена пара `(k,T())` и указанное выражение вернет `T()`. См. также вторую сноску в главе 6, "Последовательные контейнеры", на стр. 140 о смысле `T()` для встроенных типов наподобие `int`.

Рассмотрим некоторые примеры вставки с использованием `operator[]`. В процессе рассмотрения мы будем отмечать сходства и отличия между отображением и массивом, а также векторами и деками. Рассмотрим проблему вычисления скалярного произведения двух векторов действительных чисел. Здесь слово "вектор" мы используем в традиционном математическом смысле, как означающее кортеж действительных чисел, а не как вектор STL. Далее во избежание путаницы мы будем использовать термин кортеж. Если $x = (x_0, x_1, \ldots, x_{n-1})$ и $y = (y_0, y_1, \ldots, y_{n-1})$, то скалярное произведение x и y определяется как

$$x \circ y = \sum_{i=0}^{n-1} x_i \cdot y_i$$

Если x и y хранятся в массивах, векторах или деках, то скалярное произведение легко вычислить следующим образом:

```
double sum = 0;
for (int i = 0; i < n; ++i)
    sum += x[i] * y[i];
```

Имеется также обобщенный алгоритм STL `inner_product`, который можно использовать для этой цели (см. раздел 5.5.4). Но предположим, что мы работаем с разреженными кортежами, т.е. такими, у которых большинство элементов равны 0. Предположим, например, что N равно миллиону, но в x и y имеется только несколько тысяч ненулевых элементов. Ни показанный выше цикл `for`, ни обобщенный алгоритм STL `inner_product` не используют преимущества разреженности; оба варианта будут "тупо" вычислять и суммировать миллион произведений.

Хранение *x* и *y* в отображениях, а не в векторах, причем хранение только ненулевых элементов, позволяет существенно уменьшить затраты памяти. Далее, точно так же можно сократить и время вычисления скалярного произведения — путем обхода ненулевых элементов одного из кортежей.

Перед тем как рассмотреть версию с отображениями, обратимся к векторам.

Пример 7.5. Вычисление скалярного произведения кортежей, представленных векторами

```
"ex07-05.cpp" 176 ≡
  #include <vector>
  #include <iostream>
  using namespace std;

  int main()
  {
      cout << "Вычисление скалярного произведения\n"
           << "кортежей, представленных векторами."
           << endl;

      const long N = 600000; // Длина кортежей x и y
      const long S = 10;     // Коэффициент разреженности

      cout << "\nИнициализация..." << flush;
      vector<double> x(N), y(N);
      long k;
      for (k = 0; 3 * k * S < N; ++k)
          x[3 * k * S] = 1.0;
      for (k = 0; 5 * k * S < N; ++k)
          y[5 * k * S] = 1.0;

      cout << "\n\nВычисление скалярного произведения "
           << "методом грубой силы: " << flush;
      double sum = 0.0;
      for (k = 0; k < N; ++k)
          sum += x[k] * y[k];
      cout << sum << endl;
      return 0;
  }
```

Вывод примера 7.5

```
Вычисление скалярного произведения
кортежей, представленных векторами.

Инициализация...

Вычисление скалярного произведения методом грубой силы: 4000
```

Для инициализации *x* и *y* в этой программе использован конструктор вектора, принимающий два аргумента, так что векторы после инициализации имеют по `N` элементов, равных 0.0. Это важно, поскольку `operator[]` у вектора не приводит к увеличению последнего при запросе индекса, превышающего текущий размер. Отображения не требуют такой инициализации, поскольку никаких ограничений, кроме ограничений, накладываемых самим их типом, на размер индексов (ключей) не накладывается. Можно начать работать с пустыми отображениями:

```
map<long, double> x, y;
```

Вычисление скалярного произведения теперь можно записать следующим образом:

```
map<long, double>::iterator ix, iy;
for (sum = 0.0, ix = x.begin(); ix != x.end(); ++ix) {
    long k = ix->first;
    if (y.find(k) != y.end())
        sum += x[k] * y[k];
}
```

где используется функция-член `find` отображения, которая имеет тот же смысл, что и для множеств и мультимножеств: поиск заданного ключа и возврат итератора, указывающего на запись с ключом, если таковой найден, или на позицию в контейнере, следующую за последней, если не найден.

Вычисление `y[k]` включает, по сути, тот же поиск, но его повторения можно избежать, сохраняя и разыменовывая итератор, возвращаемый вызовом `y.find(k)`, для получения значения `y[k]`. Аналогично, можно избежать неявного поиска и при получении `x[k]` путем разыменования `ix`.

Пример 7.6. Вычисление скалярного произведения кортежей, представленных отображениями

```
"ex07-06.cpp" 177 ≡
  #include <map>
  #include <iostream>
  using namespace std;
  int main()
  {
  cout << "Вычисление скалярного произведения кортежей,\n"
       << "представленных отображениями." << endl;
      const long N = 600000; // Длины кортежей x и y
      const long S = 10;     // Коэффициент разреженности

      cout << "\nИнициализация..." << flush;
      map<long, double> x, y;
      long k;
      for (k = 0; 3 * k * S < N; ++k)
          x[3 * k * S] = 1.0;
      for (k = 0; 5 * k * S < N; ++k)
          y[5 * k * S] = 1.0;

      cout << "\n\nВычисление скалярного произведения\n"
           << "с учетом разреженности: " << flush;
      double sum;
      map<long, double>::iterator ix, iy;
      for (sum = 0.0, ix = x.begin(); ix != x.end(); ++ix) {
          long i = ix->first;
          iy = y.find(i);
          if (iy != y.end())
              sum += ix->second * iy->second;
      }

      cout << sum << endl;
      return 0;
  }
```

Вывод примера 7.6

```
Вычисление скалярного произведения кортежей,
представленных отображениями.

Инициализация…

Вычисление скалярного произведения
с учетом разреженности: 4000
```

Эта программа выполняет только 4000 сложений и 4000 умножений вместо 600000 в исходной программе. Приведенная ниже таблица показывает также количество индексирований и операций с итераторами, а также шагов, выполняемых операциями `find`. Вычисления основаны на том факте, что в `x` хранятся $600\,000/30 = 20\,000$ записей, а в `y` — $600\,000/50 = 12\,000$ и в предположении, что в среднем каждая из 20000 операций `find` над `y` требует $\log_2 12\,000 \approx 13.5$ шагов поиска.

Операции	*Использование векторов*	*Использование отображений*
Сложения	600000	4000
Умножения	600000	4000
Инкремент `++k`	600000	
Итерирование `++ix`		20000
Разыменование `x[k]`	600000	
Разыменование `*ix`		20000
Разыменование `y[k]`	600000	
Шаги поиска в `y.find(i)`		270000
Разыменование `*iy`		12000

7.2.4. Удаление

Как и в случае множеств и мультимножеств, элементы из отображений и мультиотображений могут быть удалены по ключу или по позиции. Данные типа `T` не играют никакой роли в операциях удаления, так что эти операции ведут себя в точности так же, как и в случае множеств и мультимножеств и имеют то же самое время работы $O(\log N + E)$, где N — размер отображения или мультиотображения, а E — количество удаляемых элементов.

7.2.5. Аксессоры

И вновь все очень похоже на множества и мультимножества. Отображения и мультиотображения имеют те же аксессоры, что и множества и мультимножества для обращения к информации об элементах как о линейной, отсортированной последовательности: `begin`, `end`, `rbegin`, `rend`, `empty`, `size` и `max_size`. Они имеют также одинаковые со множествами и мультимножествами функции-члены для получения информации по ключу: `find`, `lower_bound`, `upper_bound` и `equal_range`, а также `count`. Отображения имеют новый аксессор `operator[]`. Этот аксессор неприменим к мультиотображениям, поскольку в них с одним ключом могут быть связаны несколько элементов. Временные границы всех операций поиска, включая `operator[]`, одинаковы — $O(\log N)$, где N — размер отображения

или мультиотображения. Единственным исключением является `count` со временем работы $O(\log N + E)$, где E — количество элементов с данным ключом.

7.2.6. Отношения равенства и “меньше, чем”

В основном отношения равенства и “меньше, чем” у отображений и мультиотображений обладают теми же свойствами, что и у множеств и мультимножеств. Единственное отличие заключается в наличии дополнительной информации типа `T`, означающем менее частую согласованность эквивалентности ключей с оператором `value_type::operator==`, поскольку последний, в отличие от `key_compare`, (обычно) учитывает член типа `T`. То же самое замечание относится и к различиям между `key_compare` и `value_type::operator<`.

7.2.7. Присваивания

Определениям оператора `=` и функции-члена `swap`, рассматривавшимся в разделе 6.1.7, соответствуют аналогичные определения с тем же временем работы для отображений и мультиотображений.