Контейнер map (ассоциативный массив, словарь)

Обычные массивы представляют собой набор пронумерованных элементов, то есть для обращения к какому-либо элементу списка необходимо указать его номер. Номер элемента в списке однозначно идентифицирует сам элемент. Но идентифицировать данные по числовым номерам не всегда оказывается удобно. Например, маршруты поездов в России идентифицируются численно-буквенным кодом (число и одна цифра), также численно-буквенным кодом идентифицируются авиарейсы, то есть для хранения информации о рейсах поездов или самолетов в качестве идентификатора удобно было бы использовать не число, а текстовую строку.

Структура данных, позволяющая идентифицировать ее элементы не по числовому индексу, а по произвольному, называется *словарем* или *ассоциативным массивом*. Соответствующая структура данных в библиотеке STL называется map (отображение), поскольку словарь является сопоставлением элементам одного множества (ключам, то есть индексам) значения из другого множества. Для использования этой структуры необходимо подключить заголовочный файл map.

Рассмотрим простой пример использования словаря. Заведем словарь Capitals, где индексом является название страны, а значением — название столицы этой страны. Это позволит легко определять по строке с названием страны ее столицу.

```
// Создадим словарь
map <string, string> Capitals;

// Заполним его несколькими значениями
Capitals["Russia"] = "Moscow";
Capitals["Ukraine"] = "Kiev";
Capitals["USA"] = "Washington";

cout << "В какой стране вы живете? ";
cin >> country;

// Проверим, есть ли такая страна в словаре Capitals
if (Capitals.count(country))
{
    // Если есть - выведем ее столицу
    cout << "Столица вашей страны " << Capitals[country]
<< endl;
}
else
{
```

```
    // Запросим название столицы и добавив его в словарь
    cout << "Как называется столица вашей страны? ";
    cin >> city;
    Capitals[country] = city;
}
```

Итак, каждый элемент словаря состоит из двух объектов: *ключа* и *значения*. В нашем примере ключом является название страны, значением является название столицы. Ключ идентифицирует элемент словаря, значение является данными, которые соответствуют данному ключу. Значения ключей — уникальны, двух одинаковых ключей в словаре быть не может.

При объявлении структуры данных из шаблона map необходимо указать типы данных ключа и значения. В нашем примере оба этих типа string.

В жизни широко распространены словари, например, привычные бумажные словари (толковые, орфографические, лингвистические). В них ключом является слово-заголовок статьи, а значением — сама статья. Для того, чтобы получить доступ к статье, необходимо указать слово-ключ.

Другой пример словаря, как структуры данных — телефонный справочник. В нем ключом является имя, а значением — номер телефона. И словарь, и телефонный справочник хранятся так, что легко найти элемент словаря по известному ключу (например, если записи хранятся в алфавитном порядке ключей, то легко можно найти известный ключ, например, бинарным поиском), но если ключ неизвестен, а известно лишь значение, то поиск элемента с данным значением может потребовать последовательного просмотра всех элементов словаря.

Особенностью ассоциативного массива является его динамичность: в него можно добавлять новые элементы с произвольными ключами и удалять уже существующие элементы. При этом размер используемой памяти пропорционален размеру ассоциативного массива. Доступ к элементам ассоциативного массива выполняется хоть и медленнее, чем к обычным массивам, но в целом довольно быстро.

Когда нужно использовать словари

Словари нужно использовать в следующих случаях:

- Подсчет числа каких-то объектов. В этом случае нужно завести словарь, в котором ключами являются объекты, а значениями — их количество.
- Хранение каких-либо данных, связанных с объектом. Ключи — объекты, значения — связанные с ними данные. Например, если нужно

по названию месяца определить его порядковый номер, то это можно сделать при помощи словаря Num["January"] = 1; Num["February"] = 2; ....
- Установка соответствия между объектами (например, “родитель—потомок”). Ключ — объект, значение — соответствующий ему объект.
- Если нужен обычный массив, но при этом максимальное значение индекса элемента очень велико, но при этом будут использоваться не все возможные индексы (так называемый “разреженный массив”), то можно использовать ассоциативный массив для экономии памяти.

Работа с элементами словаря

Основная операция со словарем: получение значения элемента по ключу, записывается так же, как и для массивов: A[key]. Если элемента с заданным ключом не существует в словаре, то возвращается 0, если значения словаря числовые, пустая строка для строковых значений и значение, возвращаемое конструктором по умолчанию для более сложных объектов.

Также для доступа к элементам словаря можно использовать метод at(key). Как и в случае с вектором, этот метод производит проверку наличия в словаре элемента с ключом key, а при его отсутствии генерирует исключение (ошибка исполнения).

Проверить принадлежность ключа key словарю можно методами count(key) (возвращает количество вхождений ключа в словать, то есть 0 или 1) или find(key) (возвращает итератор на найденный элемент или значение end(), если элемент отсутствует в словаре).

Для добавления нового элемента в словарь нужно просто присвоить ему какое-то значение: A[key] = value.

Операции доступа к элементу по его ключу и добавления элемента в словарь производятся за $O(\log(n))$

Для удаления элемента из словаря используется метод `erase()`. В качестве параметра ему нужно передать либо значение ключа удаляемого элемента (тогда удаление производится за $O(\log(n))$) , либо итератор на удаляемый элемент (тогда удаление будет проводиться за $O(\log(1))$.

Перебор элементов словаря

Как и для множеств, для словарей определены итераторы. Методы find, upper_bound, lower_bound, begin, end, rbegin, rend возвращают итератор на элемент словаря. Назначение этих элементов такое же, как для контейнера set. Методы поиска в качестве параметра получают ключ элемента.

Разыменование итератора возвращает объект типа pair, у которого поле first — это ключ элемента, а поле second — его значение.

Используя итераторы можно организовать перебор всех элементов словаря:

```
    for (auto it = Capitals.begin(); it != Capitals.end(); ++it)
{
    cout << "Страна: " << (*it).first << endl;
    cout << "Столица: " << (*it).second << endl;
}
```

Вместо громоздкой записи (*it).first и (*it).second можно использовать более компактный оператор доступа к полю через указатель “->”: it->first, it->second.

Также элементы словаря можно перебирать и при помощи range-based циклов. В этом случае значение элемента словаря - это пара из двух полей: ключ и значение. Аналогичный пример вывода элементов словаря, разыменовывать auto-переменную не нужно:

```
    for (auto elem: Capitals) {
    cout << "Страна: " << elem.first << endl;
    cout << "Столица: " << elem.second << endl;
}
```

Контейнер set (множество)

Множество — это структура данных, эквивалентная множествам в математике. Множество состоит из различных элементов заданного типа и поддерживает операции добавления элемента в множество, удаления элемента из множества, проверка принадлежности элемента множеству. Одно и то же значение хранится в множестве только один раз.

Для представления множеств в библиотеке STL имеется контейнер `set`, который реализован при помощи сбалансированного двоичного дерева поиска (красно-черного дерева), поэтому множества в STL хранятся в виде упорядоченной структуры, что позволяет перебирать элементы множества в порядке возрастания их значений. Для использования контейнера `set` нужно подключить заголовочный файл <set>.

Подробней о возможностях контейнера `set` можно прочитать, например, на сайте cppreference.com.

В простейшем случае множество, например, данных типа `int` объявляется так:

```
set <int> S;
```

Для добавления элемента в множество используется метод `insert`:

```
S.insert(x);
```

Для проверки принадлежности элемента множеству используется метод `count`. Этот метод возвращает количество вхождения передаваемого параметра в данный контейнер, но поскольку в множестве все элементы уникальные, то `count` для типа `set` всегда возвращает 0 или 1. То есть для проверки принадлежности значения `x` множеству `S` можно использовать следующий код:

```
if (S.count(x)) { ...
```

Для удаления элемента используется метод `erase`. Ему можно передать значение элемента, итератор, указывающий на элемент или два итератора (в этом случае удаляется целый интервал элементов, содержащийся между заданными итераторами). Вот два способа удалить элемент `x`:

```
S.erase(x);
```

```
S.erase(S.find(x));
```

Метод size() возвращает количество элементов в множестве, метод empty(), возвращает логическое значение, равное `true`, если в множестве нет элементов, метод `clear()` удаляет все элементы из множества.

Итераторы

С итераторами контейнера `set` можно выполнять операции инкремента (что означает переход к следующему элементу) и декремента (переход к предыдущему элементу). Итераторы можно сравнивать на равенство и неравенство. Операции сравнения итераторов при помощи "<", "<=", ">", ">=" невозможны, также невозможно использовать операции прибавления к итератору числа.

Разыменование итератора (применение унарного оператора `*`) возвращает значение элемента множества, на который указывает итератор.

У множества есть метод `begin()`, который возвращает итератор на первый элемент множества, и метод `end()`, который возвращает фиктивный итератор на элемет, следующий за последним элементом в множестве. Таким образом, вывести все элементы множества можно так:

```
set <int> S;
```

```
set <int>::iterator it;
```

```
for (it = S.begin(); it != S.end(); ++it)
```

```
    cout << *it << " "
```

Благодаря тому, что множества хранятся в упорядоченном виде, все элементы будут выведены в порядке возрастания значений.

В стандарте C++11 разрешается перебор всех элементом множества при помощи range-based цикла:

```
for (auto elem: S)

    cout << elem << " ";
```

Элементы также будут выведены в порядке возрастания.

Для вывода элементов в порядке убывания можно использовать reverse_iterator аналогично векторам:

```
for (auto it = S.rbegin(); it != S.rend(); ++it)

    cout << *it << " ";
```

Функции удаления элементов могут принимать итератор в качестве параметра. В этом случае удаляется элемент, на который указывает итератор. Например, чтобы удалить наименьший элемент:

```
S.erase(S.begin());
```

Но для удаления последнего (наибольшего) элемента в set нельзя использовать reverse_iterator, нужно взять обычный итератор, указывающий на end(), уменьшить и удалить:

```
auto it = S.begin();

--it;

S.erase(it);
```

Поиск элемента в set

Для поиска конкретного элемента в `set` используется метод `find`. Этот метод возвращает *итератор* на элемент, а если элемент не найден, то он возвращает итератор `end()` (т.е. на фиктивный элемент, следующий за последним элементом множества. Используя этот метод проверить принадлежность элемента множеству можно так:

```
if (S.find(x) != S.end()) { ...
```

Также есть методы `lower_bound` и `upper_bound`, которые находят первых элемент, больше или равный x и первый элемент, строго больший x (аналогично двоичному поиску элемента в массиве).

Эти методы также возвращают итераторы, а если таких элементов (больше или равных или строго больших) нет в множестве, они возвращают `end()`.

Например, удалить из `set` минимальный элемент, строго больший x можно так:

```
auto it = S.upper_bound(x);

if (it != S.end())

    S.erase(it);
```

Контейнер multiset

Контейнер `multiset` (он также определен в заголовочном файле `set`) реализует упорядоченное множество, которое может содержать несколько равных друг другу элементов.

Во многом этот контейнер похож на `set`, есть некоторые отличия.

1. Метод `erase` если ему передать значение удаляемого элемента, удаляет все элементы, равные данному. Для удаления ровно одного элемента нужно методу `erase` передать итератор на удаляемый элемент. Например, чтобы удалить ровно один элемент, равный x, нужно сделать так:

```
s.erase(s.find(x));
```

2. Метод `count(x)` возвращает количество элементов, равных x. Сложность этого алгоритма равна $O(\log n + k)$, где n -- количество элементов в множестве, k -- количество элементов, равных x. То есть единственный способ понять, сколько в множестве содержится элементов, равных x -- это найти первый элемент, а затем двигаться дальше по дереву просматривая все элементы, до появления первого элемента, не равного x.

Контейнер multimap

Контейнер `multimap` хранит пары элементов (ключ, значение), при этом могут существовать элементы с повторяющимися значениями ключей. Этот контейнер объявлен в заголовочном файле map и используется крайне редко.

У этого элемента нет возможности обращаться к элементам по индексам при помощи операции `[]` или метода `at`, поскольку непонятно, какое значение они должны возвращать при совпадении ключей.

Для добавления элементов в контейнер `multimap` используется метод `insert`, аргументом которого является пара (ключ, значение). Тем самым можно добавлять значения, имеющие одинаковые ключи, путем многократного `insert`.

Для поиска элементов в контейнере можно использовать методы `find`, `lower_bound`, `upper_bound`, для перебора элементов можно использовать итераторы или `range-based` циклы.