Глава 11. Контейнерные классы

Контейнерные классы — это классы, предназначенные для хранения данных, организованных определенным образом. Один и тот же вид контейнера можно использовать для хранения данных различных типов. Эта возможность реализована с помощью шаблонов классов, поэтому часть библиотеки языка C++, в которую входят контейнерные классы, а также алгоритмы и итераторы (см. следующие главы), называют *стандартной библиотекой шаблонов* (Standard Template Library, STL).

Данные хранятся в *контейнерах*, а операции с ними определяются методами контейнеров и адаптируемыми *алгоритмами*. *Итераторы* «склеивают» эти два компонента. Благодаря им любой алгоритм может работать с любым контейнером.

Профессиональное программирование невозможно представить без применения библиотечных классов, и в частности контейнеров. Их использование позволяет повысить надежность, переносимость и универсальность программ, а также сократить сроки их разработки. Освоение библиотеки требует больших усилий, но они окупаются сторицей.

STL содержит контейнеры, реализующие основные структуры данных, используемые при написании программ: *векторы*, *очереди*, *списки*, *словари* и *множества*. Контейнеры можно разделить на два типа: последовательные и ассоциативные.

Последовательные контейнеры обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности. К ним относятся векторы (`vector`), двусторонние очереди (`deque`), списки (`list`) и односвязные списки (`forward_list`), а также так называемые *адаптеры*, то есть варианты контейнеров, включая стеки (`stack`), очереди (`queue`) и очереди с приоритетами (`priority_queue`). Еще одним видом контейнера с ограниченным набором операций является массив (`array`).

Каждый вид контейнера обеспечивает свой набор действий над данными. Выбор вида контейнера зависит от того, что требуется делать с данными в программе. Например, при необходимости часто вставлять и удалять элементы в середине последовательности следует использовать списки, а если включение элементов выполняется главным образом в конец или начало — двустороннюю очередь.

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров: словари (`map`), словари с дубликатами (`multimap`), множества (`set`), множества с дубликатами (`multiset`) и битовые множества (`bitset`).

Программист может создавать собственные контейнерные классы на основе имеющихся в стандартной библиотеке.

Центральным понятием STL является *шаблон*, поэтому перед тем, как приступить к изучению материала этой главы, читателю рекомендуется убедиться, что это понятие не является для него загадкой (см. разделы «Шаблоны функций» на с. 131 и «Шаблоны классов» на с. 226). Также необходимо знать, что такое пространства имен (см. с. 149), перегрузка функций (см. с. 129) и перегрузка операций (см. с. 191).

Все контейнерные классы предоставляют стандартизованный интерфейс. Смысл одноименных операций для различных контейнеров одинаков, основные операции применимы ко всем типам контейнеров. Стандарт определяет только интерфейс контейнеров, поэтому разные реализации могут сильно отличаться по эффективности. Все контейнеры сами управляют собственной памятью, поэтому программисту нет необходимости об этом думать.

Практически в любом контейнере определены поля перечисленных следующих типов:

- `value_type` — тип элемента контейнера;
- `size_type` — тип индексов, счетчиков элементов и т. д.;
- `iterator` — итератор;
- `const_iterator` — константный итератор;
- `reverse_iterator` — обратный итератор;
- `const_reverse_iterator` — константный обратный итератор;
- `reference` — ссылка на элемент;
- `const_reference` — константная ссылка на элемент;
- `key_type` — тип ключа (для ассоциативных контейнеров);
- `key_compare` — тип критерия сравнения (для ассоциативных контейнеров).

Итератор является аналогом указателя на элемент. Он используется для просмотра контейнера в прямом или обратном направлении. Все, что требуется от итератора, — уметь ссылаться на элемент контейнера и реализовывать операцию перехода к его следующему элементу. Константные итераторы используются, когда значения соответствующих элементов контейнера не изменяются. Итераторам посвящена глава 12.

При помощи итераторов можно просматривать контейнеры, не заботясь о фактических типах данных, используемых для доступа к элементам. Для этого в каждом контейнере определено несколько методов, перечисленных в табл. 11.1. В каждом контейнере эти типы и методы определяются способом, зависящим от их реализации.

Во всех контейнерах есть методы получения сведений о размере контейнера:

- `size()` — число элементов;
- `max_size()` — максимальный размер контейнера (порядка 1 миллиарда элементов);
- `empty()` — булевский метод, показывающий, пуст ли контейнер.

Другие поля и методы контейнеров будем рассматривать по мере необходимости.

Таблица 11.1. Общие методы контейнерных классов

Метод	Пояснение
iterator begin() const_iterator begin () const	Указывают на первый элемент
iterator end() const_iterator end () const	Указывают на элемент, следующий за последним
reverse_iterator rbegin() const_reverse_iterator rbegin () const	Указывают на первый элемент в обратной последовательности
reverse_iterator rend() const_reverse_iterator rend () const	Указывают на элемент, следующий за последним, в обратной последовательности

ПРИМЕЧАНИЕ

Элементы любого контейнера являются копиями вставляемых в него объектов. Поэтому для них должны быть определены конструктор копирования и операция присваивания.

Последовательные контейнеры

Для использования последовательных контейнеров необходимо подключить к программе один из заголовочных файлов, `<array>`, `<deque>`, `<dynarray>`, `<forward_list>`, `<list>` или `<vector>`.

Вектор (`vector`), массив (`array`), динамический массив (`dynarray`), двусторонняя очередь (`deque`), двусвязный список (`list`) и односвязный список (`forward_list`) поддерживают разные наборы операций, среди которых есть совпадающие. Они могут быть реализованы с разной эффективностью, примеры представлены в табл. 11.2.

Таблица 11.2. Операции с контейнерами

Операция	Метод	vector	deque	list
Вставка в начало	push_front	–	+	+
Удаление из начала	pop_front	–	+	+
Вставка в конец	push_back	+	+	+
Удаление из конца	pop_back	+	+	+
Вставка в произвольное место	insert	(+)	(+)	+
Удаление из произвольного места	erase	(+)	(+)	+
Произвольный доступ к элементу	[ ], at	+	+	–

Знак + означает, что соответствующая операция реализуется за постоянное время, не зависящее от количества n элементов в контейнере. Знак (+) означает, что соответствующая операция реализуется за время, пропорциональное n.

Для малых n время операций, обозначенных знаком +, может превышать время операций, обозначенных как (+), но для большого количества элементов последние могут оказаться очень дорогими. Как видно из таблицы, такими операциями являются вставка и удаление произвольных элементов очереди и вектора, поскольку при этом все последующие элементы требуется переписывать на новые места.

Итак, *вектор* — это структура, эффективно реализующая произвольный доступ к элементам, добавление в конец и удаление из конца.

Двусторонняя очередь эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов.

Списки эффективно реализуют вставку и удаление элементов в произвольном месте, но не имеют произвольного доступа к своим элементам.

Массивы эффективно реализуют произвольный доступ к своим элементам. Массив типа `array` предназначен для хранения данных в количестве, известном при компиляции программы, массив типа `dynarray` применяется, когда количество данных известно при создании объекта.

Любой последовательный контейнер можно создать с помощью *конструкторов*:

- без аргументов;
- с аргументами-итераторами, в частности указателями;
- с количеством элементов и значением по умолчанию;
- с другим контейнером в качестве аргумента.

Пример работы с контейнером приведен в листинге 11.1. В файле находится произвольное количество целых чисел. Программа считывает их в вектор и выводит на экран в том же порядке.

Листинг 11.1. Первое знакомство с контейнерными классами

```
#include <fstream>
#include <vector>
using namespace std;
int main(){
    ifstream input("inpnum.txt");
    if ( !input ) { cout << "Ошибка открытия входного файла"; exit(1); }
    vector<int> v;       // создание контейнера
    int x;
    while ( input >> x, !input.eof() ) v.push_back(x);
    for ( auto temp : v ) cout << temp << " ";
}
```

Поскольку файл содержит целые числа, используется соответствующая специализация шаблона `vector` — `vector<int>`. Для создания вектора `v` применяется конструктор по умолчанию. Организуется цикл до конца файла, в котором из него

считывается очередное целое число. С помощью метода push_back оно заносится в вектор, размер которого увеличивается автоматически[1].

Перебор элементов вектора выполняется с помощью введенного в C++11 *цикла по диапазону* (см. с. 69). Раньше для этого требовались итераторы, например:

```
for ( vector<int>::iterator i = v.begin(); i != v.end(); ++i )
     cout << *i << " ";
```

Здесь объявляется переменная i типа «итератор для конкретной специализации шаблона». С помощью этого итератора выполняется доступ ко всем по порядку элементам контейнера начиная с первого. Метод begin возвращает указатель на первый элемент, метод end — на элемент, следующий за последним. Реализация гарантирует, что этот указатель определен. Сравнивать текущее значение с граничным следует именно с помощью операции !=, так как операции < или <= могут быть не определены для некоторых контейнеров. Операция инкремента (++i) реализована так, чтобы после нее итератор указывал на следующий элемент контейнера в порядке обхода. Доступ к элементу вектора выполняется с помощью операции разадресации, как для обычных указателей.

В этом листинге вместо вектора можно использовать любой последовательный контейнер путем простой замены слова vector, например словом deque или list. При этом изменится внутреннее представление данных и набор доступных операций, а поведение программы останется прежним. Однако есть нюансы, например:

```
for ( int i = 0; i < v.size(); ++i ) cout << v[i] << " ";
```

Если записать цикл for с использованием операции доступа по индексу [], как в данном примере, программа не будет работать для контейнера типа list, поскольку в нем эта операция не определена.

Вектор

Вектор (vector) предназначен для хранения конечного количества объектов одного типа. Вектор эффективно реализует произвольный доступ к элементам, добавление элемента в конец и удаление последнего элемента. Менее эффективна вставка и удаление произвольного элемента, еще медленнее выполняются те же операции с первым элементом контейнера. Впрочем, скорость выполнения операций начинает сказываться только при большом количестве элементов, входящих в вектор.

Поскольку изменение размера вектора обходится дорого, весьма полезно при его создании задавать начальный размер. При этом для встроенных типов выполняется инициализация каждого элемента значением по умолчанию. Если оно не указано, элементы глобальных векторов инициализируются нулем.

---

[1] Размер вектора не изменяется каждый раз при добавлении элемента, это было бы нерационально. Он увеличивается по определенному алгоритму, которым можно управлять (см. с. 303).

Если тип элемента вектора определен пользователем, начальное значение формируется с помощью конструктора по умолчанию для данного типа.

Примеры создания векторов:

```
// Создается вектор из 10 равных единице элементов:
vector <int> v1 (10, 1);
// Создается вектор, равный вектору v1:
vector <int> v2 (v1);
// Создается вектор из двух элементов, равных первым двум элементам v1:
vector <int> v3 (v1.begin(), v1.begin() + 2);
// Создается вектор из 10 объектов класса monster (см. с. 184)
// (работает конструктор по умолчанию):
vector <monster> m1 (10);
// Создается вектор из 5 объектов класса monster с заданным именем
// (работает конструктор с параметром char*):
vector <monster> m2 ( 5, monster( "Вася" ) );
```

В шаблоне `vector` определены операция *присваивания* и метод *копирования* (`assign`). Векторы можно присваивать друг другу так же, как стандартные типы данных или строки. После присваивания размер вектора становится равным новому значению, все старые элементы удаляются.

Метод `assign` применяется к существующему объекту. Параметры метода аналогичны параметрам конструктора, например:

```
vector <int> v1, v2;
// Первым 10 элементам вектора v1 присваивается значение 1:
v1.assign( 10, 1 );
// Первым 3 элементам вектора v2 присваиваются значения v1[5], v1[6], v1[7]:
v2.assign( v1.begin() + 5, v1.begin() + 8 );
```

Итераторы класса `vector` перечислены в табл. 11.1.

Доступ к элементам вектора выполняется с помощью следующих операций и методов:

```
reference          operator[]( size_type n );
const_reference    operator[]( size_type n ) const;
const_reference    at( size_type n ) const;
reference          at( size_type n );
reference          front();
const_reference    front() const;
reference          back();
const_reference    back() const;
```

Операция `[]` выполняет доступ к элементу вектора по индексу без проверки его выхода за границу вектора. Метод `at` выполняет такую проверку и порождает исключение `out_of_range` в случае выхода за границу вектора. Естественно, что метод `at` работает медленнее, чем операция `[]`, поэтому в случаях, когда диапазон определен явно, предпочтительнее пользоваться ею. В противном случае применяется метод `at` с обработкой исключения:

```
try {
    // ...
    v.at(i) = v.at(j);
}
catch ( out_of_range) { ... }
```

Операции доступа возвращают значение ссылки на элемент (`reference`) или константной ссылки (`const_reference`) в зависимости от того, применяются они к константному объекту или нет.

Методы `front` и `back` возвращают ссылки на первый и последний элементы вектора соответственно (это не то же самое, что `begin` — указатель на первый элемент и `end` — указатель на элемент, следующий за последним). Пример:

```
vector <int> v(5, 10);
v.front() = 100;    v.back() = 100;
cout << v[0] << "  " << v[v.size() - 1];    //    Вывод: 100 100
```

Метод `capacity` определяет *объем памяти*, занимаемой вектором:

```
 size_type capacity() const;
```

Память под вектор выделяется динамически, но не под один элемент в каждый момент времени (это было бы расточительным расходованием ресурсов), а сразу под группу элементов, например 256 или 1024. Перераспределение памяти происходит только при превышении этого количества элементов, при этом объем выделенного пространства удваивается. После перераспределения любые итераторы, ссылающиеся на элементы вектора, становятся недействительными, поскольку вектор может быть перемещен в другой участок памяти и нельзя ожидать, что связанные с ним ссылки будут обновлены автоматически.

Существует также метод *выделения памяти* `reserve`, который позволяет задать, сколько памяти требуется для хранения вектора:

```
void reserve( size_type n );
```

Пример применения метода:

```
vector <int> v;
v.reserve(1000);    // Выделение памяти под 1000 элементов
```

После выполнения этого метода результат метода `capacity` будет равен по меньшей мере `n`. Метод `reserve` полезно применять, когда размер вектора известен заранее.

Для *изменения размеров* вектора служит метод `resize`:

```
void resize( size_type sz, T c = T() );
```

Этот метод увеличивает или уменьшает размер вектора в зависимости от того, больше задаваемое значение `sz`, чем значение `size()`, или меньше. Второй параметр представляет собой значение, которое присваивается всем новым элементам век-

тора. Они помещаются в конец вектора. Если новый размер меньше, чем значение `size()`, из конца вектора удаляется `size() - sz` элементов.

Определены следующие *методы изменения объектов* класса `vector`:

```
void push_back( const T& value );
void pop_back();
iterator insert( iterator position, const T& value );
void insert( iterator position, size_type n, const T& value );
template <class InputIter>
    void insert( iterator position, InputIter first, InputIter last );
iterator erase( iterator position );
iterator erase( iterator first, iterator last );
void swap();
void clear(); // Очистка вектора
```

Метод `push_back` добавляет элемент в конец вектора, метод `pop_back` — удаляет элемент из конца вектора.

Метод `insert` служит для вставки элемента в вектор. Первая форма метода вставляет элемент `value` в позицию, заданную первым параметром (итератором), и возвращает итератор, ссылающийся на вставленный элемент. Вторая форма метода вставляет в вектор `n` одинаковых элементов. Третья форма метода позволяет вставить несколько элементов, которые могут быть заданы любым диапазоном элементов подходящего типа, например:

```
vector <int> v(2), v1( 3, 9 );
int m[3] = { 3, 4, 5 };
v.insert( v.begin(), m, m + 3 );    // Содержимое v: 3 4 5 0 0
v1.insert( v1.begin() + 1, v.begin(), v.begin() + 2 );
// Содержимое v1: 9 3 4 9 9
```

Вставка в вектор занимает время, пропорциональное количеству сдвигаемых на новые позиции элементов. Если при этом новый размер вектора превышает объем занимаемой памяти, происходит перераспределение памяти. Это плата за легкость доступа по индексу. Если перераспределения не происходит, все итераторы сохраняют свои значения, в противном случае они становятся недействительными.

Метод `erase` служит для удаления одного элемента вектора (первая форма метода) или диапазона, заданного с помощью итераторов (вторая форма):

```
vector <int> v;
for ( int i = 1; i < 6; ++i ) v.push_back(i);
// Содержимое v: 1 2 3 4 5
v.erase( v.begin() );                    // Содержимое v: 2 3 4 5
v.erase( v.begin(), v.begin() + 2 );     // Содержимое v: 4 5
```

Обратите внимание, что третьим параметром задается не последний удаляемый элемент, а элемент, следующий за ним.

Каждый вызов метода `erase` так же, как и в случае вставки, занимает время, пропорциональное количеству сдвигаемых на новые позиции элементов. Все итераторы и ссылки «правее» места удаления становятся недействительными.

Функция `swap` служит для обмена элементов двух векторов одного типа, но не обязательно одного размера:

```
vector <int> v1, v2;
...
v1.swap(v2);    // Эквивалентно v2.swap(v1);
```

Для векторов определены *операции сравнения* `==`, `!=`, `<`, `>`, `<=` и `>=`. Два вектора считаются равными, если равны их размеры и все соответствующие пары элементов. Один вектор меньше другого, если, во-первых, в нем меньше элементов или, во-вторых, при равном количестве элементов первый из элементов одного вектора, не равный соответствующему элементу другого, меньше него (то есть сравнение лексикографическое). Пример приведен в листинге 11.2.

Листинг 11.2. Сравнение векторов

```
#include <vector>
using namespace std;
int main(){
    vector <int> v1, v2;
    for (int i = 0; i < 6; ++i ) v1.push_back(i);
    cout << "v1: ";
    for ( auto temp : v1 ) cout << temp << " ";
    cout << endl;
    for ( int i = 0; i < 3; ++i ) v2.push_back( i + 1 );
    cout << "v2: ";
    for ( auto temp : v2 ) cout << temp << " ";
    for ( int i = 0; i < 3; ++i ) cout << v2[i] << " ";
    cout << endl;
    if ( v1 < v2 )  cout << " v1 < v2" << endl;
    else  cout << " v1 >= v2" << endl;
}
```

Результат работы программы:

```
v1: 0 1 2 3 4 5
v2: 1 2 3
v1 < v2
```

Для эффективной работы с векторами в стандартной библиотеке определены шаблоны функций, называемые алгоритмами. Они включают в себя поиск значений, сортировку элементов, вставку, замену, удаление и другие операции. Алгоритмы описаны в главе 13.

Вектор логических значений

Специализация *вектора логических значений* (`vector <bool>`) определена для оптимизации размещения памяти, поскольку можно реализовать его так, чтобы его элемент занимал 1 бит. При этом адресация отдельных битов выполняется программно. Итератор такого вектора не может быть указателем. В остальном векторы

логических значений аналогичны обычным и реализуют тот же набор операций и методов. В дополнение к ним определены методы инвертирования бита и вектора в целом (`flip`).

Ссылка на элемент вектора логических значений реализована в виде класса `reference`, моделирующего обычную ссылку на элемент:

```
class reference {
        friend class vector;
        reference();
    public:
        ~reference();
        operator bool() const;
        reference& operator=( const bool x );
        reference& operator=( const reference& x );
        void flip();
};
```

В листинге 11.3 с клавиатуры в вектор вводится 10 значений 0 или 1, после чего они выводятся на экран.

Листинг 11.3. Вектор логических значений

```
#include <vector>
#include <iostream>
using namespace std;
int main(){
    vector <bool> v(10);
    int temp;
    for ( unsigned int i = 0; i < v.size(); ++i ) {
        cin >> temp;
        v[i] = temp;
    }
    for ( auto temp : v ) cout << temp;
}
```

Двусторонняя очередь

Двусторонняя очередь (`deque`), или *дек*, — это последовательный контейнер, который поддерживает произвольный доступ к элементам и обеспечивает вставку и удаление элементов на обоих концах очереди за постоянное время. Те же операции с элементами внутри очереди занимают время, пропорциональное количеству перемещаемых элементов. Распределение памяти выполняется автоматически.

Основное отличие дека от вектора — наличие методов добавления элемента в начало очереди и удаления элемента из ее конца:

```
void push_front( const T& );          // добавление элементов в начало дека
void pop_front();                     // удаление элементов из начала дека
```

При выборке элемент удаляется из очереди.

Рассмотрим схему организации очереди (рис. 11.1). Чтобы обеспечить произвольный доступ к элементам за постоянное время, очередь разбита на блоки, доступ к каждому из которых осуществляется через указатель. На рисунке закрашенные области соответствуют занятым элементам очереди. Если при добавлении в начало или в конец блок оказывается заполненным, выделяется память под очередной блок (например, после заполнения блока 4 будет выделена память под блок 5, а после заполнения блока 2 — под блок 1). При заполнении крайнего из блоков происходит перераспределение памяти под массив указателей так, чтобы использовались только средние элементы. Это не занимает много времени. Таким образом, доступ к элементам очереди осуществляется за постоянное время, хотя оно и несколько больше, чем для вектора.

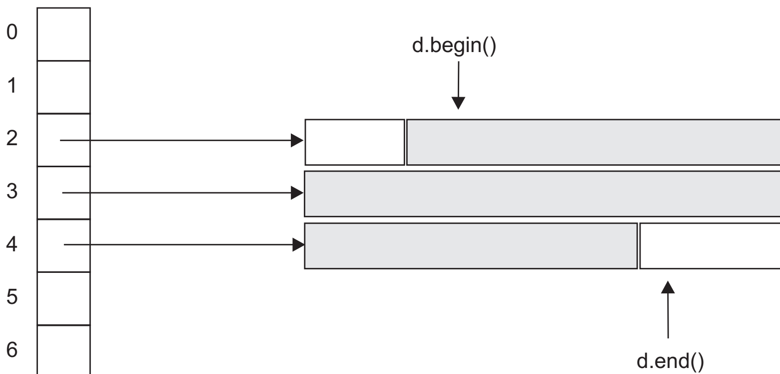

Рис. 11.1. Организация двусторонней очереди

Примеры создания очередей:

```
// Создается очередь из 10 равных единице элементов:
deque <int> d1 ( 10, 1 );
// Создается очередь, равная очереди d1:
deque <int> d2 ( v1 );
// Создается очередь из двух элементов, равных первым двум
// элементам вектора v1 из предыдущего раздела:
deque <int> d3 ( v1.begin(), v1.begin() + 2 );
// Создается очередь из 10 объектов класса monster (см. с. 184)
// (работает конструктор по умолчанию):
deque <monster> m1 (10);
// Создается очередь из 5 объектов класса monster с заданным именем
// (работает конструктор с параметром char*):
deque <monster> m2 ( 5, monster( "Вася в очереди" ) );
```

Для очереди не определены методы `capacity` и `reserve`, но есть методы `resize` и `size`. К очередям можно применять алгоритмы стандартной библиотеки, описанные в главе 13.