

Лабораторная работа №6. Стандартная библиотека шаблонов

Основные концепции стандартной библиотеки

Стандартная библиотека шаблонов (Standard Template Library, STL) состоит из двух основных частей: набора контейнерных классов и набора обобщенных алгоритмов.

Контейнеры — это объекты, содержащие другие однотипные объекты. Контейнерные классы являются шаблонными, поэтому хранимые в них объекты могут быть как встроенных, так и пользовательских типов. Эти объекты должны допускать копирование и присваивание. Встроенные типы этим требованиям удовлетворяют; то же самое относится к классам, если конструктор копирования или операция присваивания не объявлены в них закрытыми или защищенными. В контейнерных классах реализованы такие типовые структуры данных, как стек, список, очередь и т. д.

Обобщенные алгоритмы реализуют большое количество процедур, применимых к контейнерам — например, поиск, сортировку, слияние и т. п., но они не являются методами контейнерных классов. Наоборот, алгоритмы представлены в STL в форме глобальных шаблонных функций. Благодаря этому достигается их универсальность: эти функции можно применять не только к объектам контейнерных классов, но и к массивам. Независимость от типов контейнеров достигается за счет косвенной связи функции с контейнером: в функцию передается не сам контейнер, а пара адресов, *first*, *last*, задающая диапазон обрабатываемых элементов.

Реализация указанного механизма взаимодействия базируется на использовании так называемых *итераторов*. *Итераторы* — это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать для последовательного продвижения по контейнеру, как обычные указатели, а также разыменовывать для получения или изменения значения элемента.

Контейнеры

Контейнеры STL можно разделить на два типа: последовательные и ассоциативные.

Последовательные контейнеры обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым последовательным контейнерам относятся *векторы* (*vector*), *списки* (*list*) и *двусторонние очереди* (*deque*). Есть еще специализированные контейнеры (или *адаптеры* контейнеров), реализованные на основе базовых, — *стеки* (*stack*), *очереди* (*queue*) и *очереди с приоритетами* (*priority_queue*).

Кстати, обычный встроенный массив C++ также может рассматриваться как последовательный контейнер. Проблема с массивами заключается в том, что их размеры нужно указывать в исходном коде, а это часто бывает неизвестно заранее. Если же выделять память для массива динамически (операцией *new*), алгоритм усложняется из-за необходимости отслеживать время жизни массива и вовремя освобождать память. Использование контейнера *вектор* вместо динамического массива упрощает жизнь программиста, в чем вы могли убедиться на семинаре 11.

Для использования контейнера в программе необходимо включить в нее соответствующий заголовочный файл. Тип объектов, сохраняемых в контейнере, задается с помощью аргумента шаблона, например:

```
vector<int> aVect;           // создать вектор aVect целых чисел (типа int)
list<Man> department;       // создать список department из элементов типа Man
```

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

Итераторы

Чтобы понять, зачем нужны итераторы, давайте посмотрим, как можно реализовать шаблонную функцию для поиска значения *value* в обычном массиве, хранящем объекты типа *T*. Например, возможно следующее решение:

```
template <class T> T* Find( T* ar, int n, const T& value ) {
    for ( int i = 0; i < n; ++i ) if ( ar[i] == value ) return &ar[i];
    return 0;
}
```

Функция возвращает адрес найденного элемента или 0, если элемент с заданным значением не найден. Цикл *for* может быть записан и в несколько иной форме:

```
for ( int i = 0; i < n; ++i ) if ( *( ar + i ) == value ) return ar + i;
```

Работа функции при этом останется прежней. При продвижении по массиву адрес следующего элемента вычисляется и в первом, и во втором случаях с использованием арифметики указателей, то есть адреса соседних элементов различаются на количество байтов, требуемое для хранения одного элемента.

Попытаемся теперь расширить сферу применения нашей функции — хорошо бы, чтобы она решала задачу поиска заданного значения среди объектов, хранящихся в виде линейного списка! Однако, к сожалению, ничего не выйдет: адрес следующего элемента в списке нельзя вычислить, пользуясь арифметикой указателей.

Элементы списка могут размещаться в памяти самым причудливым образом, а информация об адресе следующего объекта хранится в одном из полей текущего объекта.

Авторы STL решили эту проблему, введя понятие *итератора* как более абстрактной сущности, чем указатель, но обладающей похожим поведением. Для всех контейнерных классов STL определен тип `iterator`, однако реализация его в разных классах разная. Например, в классе `vector`, где объекты размещаются один за другим, как в массиве, тип итератора определяется посредством `typedef T* iterator`. А вот в классе `list` тип итератора реализован как встроенный класс `iterator`, поддерживающий основные операции с итераторами.

К основным операциям, выполняемым с любыми итераторами, относятся:

- Разыменование итератора: если `p` — итератор, то `*p` — значение объекта, на который он ссылается.
- Присваивание одного итератора другому.
- Сравнение итераторов на равенство и неравенство (`==` и `!=`).
- Перемещение его по всем элементам контейнера с помощью префиксного (`++p`) или постфиксного инкремента (`p++`).

Так как реализация итератора специфична для каждого класса, при объявлении объектов типа «итератор» всегда указывается область видимости в форме `имя_шаблона::`, например:

```
vector<int>::iterator iter1;  
list<Man>::iterator iter2;
```

Организация циклов просмотра элементов контейнеров тоже имеет некоторую специфику. Так, если `i` — некоторый итератор, то вместо привычной формы

```
for ( i = 0; i < n; ++i )
```

используется следующая:

```
for ( i = first; i != last; ++i )
```

где `first` — значение итератора, указывающее на первый элемент в контейнере, а `last` — значение итератора, указывающее на воображаемый элемент, который следует за последним элементом контейнера. Операция `<` заменена на операцию `!=`, поскольку операции `<` и `>` для итераторов в общем случае не поддерживаются.

Для всех контейнерных классов определены унифицированные методы `begin` и `end`, возвращающие адреса `first` и `last` соответственно.

Вообще, все типы итераторов в STL принадлежат одной из пяти категорий: входные, выходные, прямые, двунаправленные итераторы и итераторы произвольного доступа.

Входные итераторы (*InputIterator*) используются алгоритмами STL для чтения значений из контейнера, аналогично тому, как вводятся данные из потока `cin`. *Выходные итераторы* (*OutputIterator*) используются алгоритмами для записи значений в контейнер, аналогично тому, как выводятся данные в поток `cout`.

Прямые итераторы (*ForwardIterator*) используются алгоритмами для навигации по контейнеру только в прямом направлении, причем они позволяют и читать, и изменять данные в контейнере.

Двунаправленные итераторы (*BidirectionalIterator*) имеют все свойства прямых итераторов, но позволяют осуществлять навигацию по контейнеру и в прямом, и в обратном направлениях (для них дополнительно реализованы операции префиксного и постфиксного декремента).

Итераторы произвольного доступа (*RandomAccessIterator*) имеют все свойства двунаправленных итераторов плюс операции (наподобие сложения указателей) для доступа к произвольному элементу контейнера.

Значения прямых, двунаправленных и итераторов произвольного доступа могут быть сохранены, а вот значения входных и выходных итераторов сохраняться не могут (аналогично тому, как не может

быть гарантирован ввод тех же самых значений при вторичном обращении к потоку `cin`). Как следствие, любые алгоритмы, основанные на входных или выходных итераторах, должны быть однопроходными.

Вернемся к функции `Find`, которую мы безуспешно пытались обобщить для любых типов контейнеров. В STL аналогичный алгоритм имеет следующий прототип:

```
template <class InputIterator, class T>
InputIterator find( InputIterator first, InputIterator last, const T& value );
```

Для двунаправленных итераторов и итераторов произвольного доступа определены разновидности, называемые *адаптерами итераторов*. Адаптер, просматривающий последовательность в обратном направлении, называется `reverse_iterator`. Другие специализированные итераторы-адаптеры мы рассмотрим ниже.

Общие свойства контейнеров

В табл. 6.1 приведены имена типов, определенные с помощью `typedef` в большинстве контейнерных классов, а в табл. 6.2 — некоторые общие для всех контейнеров операции.

Таблица 6.1. Унифицированные типы, определенные в STL

Поле	Пояснение
<code>value_type</code>	Тип элемента контейнера
<code>size_type</code> ¹	Тип индексов, счетчиков элементов и т. д.
<code>iterator</code>	Итератор
<code>const_iterator</code>	Константный итератор (значения элементов изменять запрещено)
<code>reference</code>	Ссылка на элемент
<code>const_reference</code>	Константная ссылка на элемент (значение элемента изменять запрещено)
<code>key_type</code>	Тип ключа (для ассоциативных контейнеров)
<code>key_compare</code>	Тип критерия сравнения (для ассоциативных контейнеров)

Таблица 6.2. Операции и методы, общие для всех контейнеров

Операция или метод	Пояснение
Операции равенства (<code>==</code>) и неравенства (<code>!=</code>)	Возвращают значение <code>true</code> или <code>false</code>
Операция присваивания (<code>=</code>)	Копирует один контейнер в другой
<code>clear</code>	Удаляет все элементы
<code>insert</code>	Добавляет один элемент или диапазон элементов
<code>erase</code>	Удаляет один элемент или диапазон элементов
<code>size_type size()</code> <code>const</code>	Возвращает число элементов
<code>size_type max_size()</code> <code>const</code>	Возвращает максимально допустимый размер контейнера
<code>bool empty()</code> <code>const</code>	Возвращает <code>true</code> , если контейнер пуст
<code>iterator begin()</code>	Возвращают итератор на начало контейнера (итерации будут производиться в прямом направлении)
<code>iterator end()</code>	Возвращают итератор на конец контейнера (итерации в прямом направлении будут закончены)
<code>reverse_iterator begin()</code>	Возвращают реверсивный итератор на конец контейнера (итерации будут производиться в обратном направлении)

¹ Эквивалентен `unsigned int`.

<code>reverse_iterator</code> <code>end()</code>	Возвращают реверсивный итератор на начало контейнера (итерации в обратном направлении будут закончены)
---	--

Алгоритмы

Алгоритм — это функция, которая производит некоторые действия над элементами контейнера (контейнеров). Чтобы использовать обобщенные алгоритмы, нужно подключить к программе заголовочный файл `<algorithm>`. В табл. 6.3 приведены наиболее популярные алгоритмы STL.

Таблица 6.3. Некоторые типичные алгоритмы STL

Алгоритм	Назначение
<code>accumulate</code>	Вычисление суммы элементов в заданном диапазоне
<code>copy</code>	Копирование последовательности, начиная с первого элемента
Алгоритм	Назначение
<code>count</code>	Подсчет количества вхождений значения в последовательность
<code>count_if</code>	Подсчет количества выполнений условия в последовательности
<code>equal</code>	Попарное равенство элементов двух последовательностей
<code>fill</code>	Замена всех элементов заданным значением
<code>find</code>	Нахождение первого вхождения значения в последовательность
<code>find_first_of</code>	Нахождение первого значения из одной последовательности в другой
<code>find_if</code>	Нахождение первого соответствия условию в последовательности
<code>for_each</code>	Вызов функции для каждого элемента последовательности
<code>merge</code>	Слияние отсортированных последовательностей
<code>remove</code>	Перемещение элементов с заданным значением
<code>replace</code>	Замена элементов с заданным значением
<code>search</code>	Нахождение первого вхождения в первую последовательность второй последовательности
<code>sort</code>	Сортировка
<code>swap</code>	Обмен местами двух элементов
<code>transform</code>	Выполнение заданной операции над каждым элементом последовательности

В списках параметров всех алгоритмов первые два параметра задают диапазон обрабатываемых элементов в виде полуинтервала $[first, last)^2$, где `first` — итератор, указывающий на начало диапазона, а `last` — итератор, указывающий на выход за границы диапазона. Пример сортировки массива с помощью алгоритма `sort` (имя массива и `arr` имеет тип указателя `int*` и используется как итератор):

```
int arr[7] = { 15, 2, 19, -3, 28, 6, 8 };
sort( arr, arr + 7 );
```

Примеры использования некоторых алгоритмов будут даны ниже.

² Полуинтервал $[a, b)$ — это промежуток, включающий a , но не включающий b , — последний элемент полуинтервала предшествует элементу b .

Использование последовательных контейнеров

К основным последовательным контейнерам относятся *вектор* (vector), *список* (list) и *двусторонняя очередь* (deque). Чтобы использовать последовательный контейнер, нужно включить в программу соответствующий заголовочный файл: <vector>, <list> или <deque>.

Контейнер *вектор* является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации [] или метода at³. Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

Контейнер *список* организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список не поддерживает произвольного доступа к своим элементам: например, для выборки n-го элемента нужно последовательно выбрать предыдущие n-1 элементов. Контейнер *двусторонняя очередь* (дек) во многом аналогичен вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора дек эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

Существует пять способов определить объект для последовательного контейнера.

1. Создать пустой контейнер:
vector<int> vec1;
list<string> list1;
2. Создать контейнер заданного размера и инициализировать его элементы значениями по умолчанию:
vector<string> vec1(100);
list<double> list1(20);
3. Создать контейнер заданного размера и инициализировать его элементы указанным значением:
vector<string> vec1(100, "Hello!");
deque<int> dec1(300, -1);
4. Создать контейнер и инициализировать его элементы значениями диапазона [first, last) элементов другого контейнера:
int arr[7] = { 15, 2, 19, -3, 28, 6, 8 };
vector<int> v1(arr, arr + 7);
list<int> lst(v1.begin() + 2, v1.end());
5. Создать контейнер и инициализировать его элементы значениями элементов другого *однотипного* контейнера:

```
vector<int> v1;  
// ... здесь добавление элементов в v1  
vector<int> v2( v1 );
```

Для вставки и удаления последнего элемента контейнера любого из трех рассматриваемых классов предназначены методы `push_back` и `pop_back`. Кроме того, список и очередь (но не вектор) поддерживают операции вставки и удаления первого элемента контейнера `push_front` и `pop_front`. Учтите, что методы `pop_back` и `pop_front` не возвращают удаленное значение. Чтобы считать первый элемент, используется метод `front`, а для считывания последнего элемента — метод `back`. Кроме этого, все типы контейнеров имеют более общие операции вставки и удаления, перечисленные в табл. 6.4.

Таблица 6.4. Методы вставки в контейнер и удаления из контейнера

Метод	Пояснение
<code>insert(iterator position, const T& value)</code>	Вставка элемента со значением value в позицию, заданную итератором position

³ Метод at аналогичен операции индексации, но в отличие от нее проверяет выход за границу вектора и в случае ошибки генерирует исключение `out_of_range`.

<code>insert(iterator position, size_type n, const T& value)</code>	Вставка <code>n</code> элементов со значением <code>value</code> , начиная с позиции <code>position</code>
<code>template <class InputIter> void insert(iterator position, InputIter first, InputIter last)</code>	Вставка диапазона элементов, заданного итераторами <code>first</code> и <code>last</code> , начиная с позиции <code>position</code>
<code>erase(iterator position)</code>	Удаление элемента, на который указывает итератор <code>position</code>
<code>erase(iterator first, iterator last)</code>	Удаление диапазона элементов, заданного позициями <code>first</code> и <code>last</code>

Задача 6.1. Сортировка вектора

В файле находится произвольное количество целых чисел. Вывести их на экран в порядке возрастания.

Решение этой задачи приводится в листинге 6.1. Вместо вектора можно использовать любой последовательный контейнер, заменив слово `vector` на `deque` или `list`. При этом изменится внутреннее представление данных, а результат работы программы останется таким же.

Листинг 6.1. Сортировка вектора

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    ifstream in ( "inpnum.txt" );
    if ( !in ) { cerr << "File not found\n"; exit( 1 ); }
    vector<int> v;
    int x;
    while ( in >> x ) v.push_back( x );
    sort( v.begin(), v.end() );
    vector<int>::const_iterator i;
    for ( i = v.begin(); i != v.end(); ++i ) cout << *i << " ";
}
```

Приведем еще один пример работы с векторами, демонстрирующий использование методов `swap`, `empty`, `back`, `pop_back`:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    double arr[] = { 1.1, 2.2, 3.3, 4.4 };
    int n = sizeof( arr )/sizeof( double );
    vector<double> v1( arr, arr + n ); // Инициализация вектора массивом
    vector<double> v2;                // пустой вектор
    v1.swap( v2 );                    // обменять содержимое v1 и v2
    while ( !v2.empty() ) {
        cout << v2.back() << ' ';    // вывести последний элемент
        v2.pop_back();               // и удалить его
    }
    // Результат выполнения программы: 4.4 3.3 2.2 1.1
}
```

Шаблонная функция print для вывода содержимого контейнера

В процессе работы над программами, использующими контейнеры, часто приходится выводить на экран их текущее содержимое. Приведем шаблон функции, решающей эту задачу для любого типа контейнера (листинг 6.2).

Листинг 6.2. Шаблонная функция print для вывода содержимого контейнера

```
template <class T> void print( T& cont ) {
    typename T::const_iterator p = cont.begin();
    if ( cont.empty() ) cout << "Container is empty.";
    for ( p; p != cont.end(); ++p ) cout << *p << ' ';
    cout << endl;
}
```


Теперь можно пользоваться функцией `print`, включая ее определение в исходный файл с программой, как, например, в следующем эксперименте с очередью:

```
#include <iostream>
#include <deque>
using namespace std;
/* ... определение функции print ... */
int main() {
    deque<int> dec;      print( dec ); // Пустой контейнер
    dec.push_back( 4 );  print( dec ); // 4
    dec.push_front( 3 ); print( dec ); // 3 4
    dec.push_back( 5 );  print( dec ); // 3 4 5
    dec.push_front( 2 ); print( dec ); // 2 3 4 5
    dec.push_back( 6 );  print( dec ); // 2 3 4 5 6
    dec.push_front( 1 ); print( dec ); // 1 2 3 4 5 6
}
```

Адаптеры контейнеров

Специализированные последовательные контейнеры — *стек, очередь и очередь с приоритетами* — не являются самостоятельными контейнерными классами, а реализованы на основе рассмотренных выше классов, поэтому они называются *адаптерами контейнеров*.

Стек

Шаблонный класс `stack` (заголовочный файл `<stack>`) определен как

```
template <class T, class Container = deque<T> >
class stack { /* ... */ };
```

где параметр `Container` задает класс-прототип. По умолчанию для стека прототипом является класс `deque`. Смысл такой реализации заключается в том, что специализированный класс переопределяет интерфейс класса-прототипа, ограничивая его только методами, необходимыми новому классу. В табл. 6.5 показано, как сформирован интерфейс класса `stack` из методов класса-прототипа.

Таблица 6.5. Интерфейс класса `stack`

Методы класса <code>stack</code>	Методы класса-прототипа
<code>push</code>	<code>push_back</code>
<code>pop</code>	<code>pop_back</code>
<code>top</code>	<code>back</code>
<code>empty</code>	<code>empty</code>
<code>size</code>	<code>size</code>

В соответствии со своим назначением стек не только не позволяет выполнить произвольный доступ к своим элементам, но даже не дает возможности пошагового перемещения, в связи с чем концепция итераторов в стеке не поддерживается. Напоминаем, что метод `pop` *не возвращает* удаленное значение. Чтобы прочитать значение на вершине стека, используется метод `top`. Ниже приведен пример работы со стеком. Программа вводит из файла числа и выводит их в обратном порядке⁴:

```
int main() {
    ifstream in ( "inpnun.txt" );
    stack<int> s;
    int x;
    while ( in >> x ) s.push( x );
    while ( !s.empty() ) {
        cout << s.top() << ' ';
        s.pop();
    }
}
```

Объявление `stack<int> s` создает стек на базе двусторонней очереди (по умолчанию). Если это нас по каким-то причинам не устраивает и мы хотим создать стек на базе списка, то объявление будет выглядеть следующим образом: `stack<int, list<int> > s; //` Не забывайте о пробеле между угловыми скобками `> >`

⁴ В дальнейших примерах мы будем опускать директивы `#include` и объявление `using namespace std` для экономии места, то есть стоимости книги.

Очередь

Шаблонный класс `queue` (заголовочный файл `<queue>`) является адаптером, который может быть реализован на основе двусторонней очереди (по умолчанию) или списка. Класс `vector` в качестве прототипа не подходит, так как в нем нет выборки из начала контейнера. Очередь использует для проталкивания данных один конец, а для выталкивания — другой. В соответствии с этим ее интерфейс образуют методы из табл. 6.6.

Таблица 6.6. Интерфейс класса `queue`

Методы класса <code>queue</code>	Методы класса-прототипа
<code>push</code>	<code>push_back</code>
<code>pop</code>	<code>pop_front</code>
<code>front</code>	<code>front</code>
<code>back</code>	<code>back</code>
<code>empty</code>	<code>empty</code>
<code>size</code>	<code>size</code>

Очередь с приоритетами

Шаблонный класс `priority_queue` (заголовочный файл `<queue>`) поддерживает такие же операции, что и класс `queue`, но реализация класса возможна либо на основе вектора (реализация по умолчанию), либо на основе списка. *Очередь с приоритетами* отличается от обычной тем, что для извлечения выбирается максимальный элемент из хранимых в контейнере. Поэтому после каждого изменения состояния очереди максимальный элемент из оставшихся сдвигается в начало контейнера. Если очередь с приоритетами организуется для объектов класса, определенного программистом, в этом классе следует определить операцию `<`. Пример работы с очередью с приоритетами приведен в листинге 6.3.

Листинг 6.3. Пример работы с очередью с приоритетами

```
#include <queue>
#include <iostream>
using namespace std;
int main() {
    priority_queue<int> P;
    P.push( 17 ); P.push( 5 ); P.push( 400 ); P.push( 2500 );
    P.push( 1 );
    while ( !P.empty() ) {
        cout << P.top() << ' ';
        P.pop();
    }
    // Результат выполнения программы: 2500 400 17 5 1
}
```

Использование алгоритмов

Вернемся к изучению алгоритмов. Не забывайте включать заголовочный файл `<algorithm>` и добавлять определение функции `print`, если она используется.

Алгоритмы `count` и `find`

Алгоритм `count` подсчитывает количество вхождений в контейнер (или его часть) значения, заданного его третьим аргументом. Алгоритм `find` выполняет поиск заданного значения и возвращает итератор на самое первое вхождение этого значения. Если значение не найдено, возвращается итератор, соответствующий возврату метода `end`. В листинге 6.4 показано использование этих алгоритмов.

Листинг 6.4. Пример использования алгоритмов `count` и `find` int

```
#include <string>
#include <iostream>
#include <list>
#include <bits/stdc++.h>
using namespace std;
template <class T> void print( T& cont ) {
    typename T::const_iterator p = cont.begin();
    if ( cont.empty() ) cout << "Container is empty.";
    for ( p; p != cont.end(); ++p ) cout << *p << ' ';
    cout << endl;
}
```



```

int main() {
    int arr[] = { 1, 2, 3, 4, 5, 2, 6, 2, 7 };
    int n = sizeof( arr ) / sizeof( int );
    vector<int> v1( arr, arr + n );
    int value = 2; // искомая величина
    int how_much = count( v1.begin(), v1.end(), value ); // 1
    cout << how_much << endl; // вывод: 3
    list<int> loc_list; // список позиций искомой величины
    vector<int>::iterator location = v1.begin();
    while ( 1 ) {
        location = find( location, v1.end(), value ); // 2
        if ( location == v1.end() ) break;
        loc_list.push_back( location - v1.begin() );
        location++;
    }
    print( loc_list ); // функция из листинга 15.2 // вывод: 1 5 7
}

```

Вектор `v1` создается, инициализируясь значениями массива `arr`. Затем с помощью алгоритма `count` подсчитывается количество вхождений в вектор значения `value`, равного двум. В цикле `while` выясняется, в каких позициях вектора размещена эта величина. Первый аргумент алгоритма `find` (переменная `location`) первоначально, перед входом в цикл, принимает значение итератора, указывающего на нулевой элемент контейнера (элементы вектора нумеруются с нуля). Затем `location` получает значение итератора, указывающего на найденный элемент.

Если поиск завершился успешно, то, во-первых, вычисляется позиция найденного элемента как разность значений `location` и адреса нулевого элемента и полученное значение заносится в список `loc_list`. Во-вторых, итератор `location` сдвигается операцией инкремента на следующую позицию в контейнере, чтобы обеспечить на следующей итерации цикла продолжение поиска в оставшейся части контейнера. Если поиск завершился неудачей, то `break` приведет к выходу из цикла.

Алгоритмы `count_if` и `find_if`

Алгоритмы `count_if` и `find_if` отличаются от алгоритмов `count` и `find` тем, что их третьим аргументом является некоторый *предикат*. *Предикат* — это функция или функциональный объект, возвращающие значение типа `bool`. Например, если в листинг 6.4 добавить определение глобальной функции `bool isMyValue(int x) { return ((x > 2) && (x < 5)); }` и заменить инструкцию с вызовом `count` (оператор, помеченный цифрой 1) на:

```
int how_much = count_if( v1.begin(), v1.end(), isMyValue );
```

то программа определит, что контейнер содержит два числа, значение которых больше двух, но меньше пяти.

Аналогично, замена инструкции с вызовом `find` (оператор 2) на:

```
location = find_if( location, v1.end(), isMyValue );
```

приведет к наполнению списка `loc_list` значениями 2 и 3 (номера позиций вектора `v1`, в которых находятся числа, удовлетворяющие предикату `isMyValue`).

Алгоритм `for_each`

Этот алгоритм позволяет выполнить некоторое действие над каждым элементом диапазона `[first, last)`. Чтобы определить, какое именно действие должно быть выполнено, нужно написать соответствующую функцию с одним аргументом типа `T` (`T` — тип данных, содержащихся в контейнере). Функция не имеет права модифицировать данные в контейнере, но может их использовать в своей работе. Имя этой функции передается алгоритму третьим аргументом. Например, в следующей программе `for_each` используется для перевода всех значений массива из дюймов в сантиметры и вывода их на экран:

```

void InchToCm( double inch ) { cout << ( inch * 2.54 ) << ' '; }
int main() {
    double inches[] = {0.5, 1.0, 1.5, 2.0, 2.5};
    for_each( inches, inches + 5, InchToCm );
}

```

Алгоритм `search`

Некоторые алгоритмы оперируют одновременно двумя контейнерами. Таков и алгоритм `search`, находящий первое вхождение в первую последовательность `[first1, last1)` второй последовательности `[first2, last2)`. Пример приведен в листинге 6.5.

Листинг 6.5. Пример использования алгоритма search

```
#include <string>
#include <iostream>
#include <list>
#include <bits/stdc++.h>
using namespace std;
int main() {
    int arr[] = { 11, 77, 33, 11, 22, 33, 11, 22, 55 };
    int pattern[] = { 11, 22, 33 };
    int* ptr = search( arr, arr + 9, pattern, pattern + 3 );
    if ( ptr == arr + 9 ) cout << "Pattern not found" << endl;
    else cout << "Found at position " << ( ptr - arr ) << endl;
    list<int> lst( arr, arr + 9 );
    list<int>::iterator ifound;
    ifound = search( lst.begin(), lst.end(), pattern, pattern + 3 );
    if ( ifound == lst.end() ) cout << "Pattern not found" << endl;
    else cout << "Found." << endl;
}
```

Результат выполнения программы:

Found at position 3

Found.

Отметим, что список не поддерживает произвольного доступа к своим элементам, а значит, не допускает операций "+" и "-" с итераторами. Поэтому мы можем только зафиксировать факт вхождения последовательности *pattern* в контейнер *lst*.

Алгоритм sort

Алгоритм сортировки можно применять только для тех контейнеров, которые обеспечивают произвольный доступ к элементам, — этому требованию удовлетворяют *массив*, *вектор* и *двусторонняя очередь*, но не удовлетворяет *список*. В связи с этим класс *list* содержит метод *sort*, решающий задачу сортировки.

Алгоритм *sort* имеет две сигнатуры:

```
template<class RandomAccessIt>
void sort( RandomAccessIt first, RandomAccessIt last );
template<class RandomAccessIt>
void sort( RandomAccessIt first, RandomAccessIt last, Compare comp );
```

Первая форма алгоритма обеспечивает сортировку элементов из диапазона *[first, last)*, причем для упорядочения по умолчанию используется операция *<*, которая должна быть определена для типа *T* (тип данных, содержащихся в контейнере). То есть сортировка по умолчанию выполняется по возрастанию значений, например:

```
int main() {
    double arr[6] = { 2.2, 0.0, 4.4, 1.1, 3.3, 1.1 };
    vector<double> v1( arr, arr + 6 );
    sort( v1.begin(), v1.end() );
    print( v1 ); // Функция из листинга 15.2
} // Результат выполнения программы: 0 1.1 1.1 2.2 3.3 4.4
```

Вторая форма алгоритма позволяет задать произвольный критерий упорядочения. Для этого нужно передать через третий аргумент соответствующий предикат, то есть функцию или функциональный объект, возвращающие значение типа *bool*. Использование функции в качестве предиката было показано при описании алгоритмов *count_if* и *find_if*. Использованию функциональных объектов посвящен следующий раздел.

Функциональные объекты

На семинаре 12 было показано, как можно использовать функциональные объекты для настройки шаблонных классов, поэтому рекомендуем вам еще раз просмотреть этот материал. *Функциональным объектом* называется объект некоторого класса, для которого определена единственная операция вызова функции *operator()*. В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения, встроенных в язык C++. Они возвращают значение типа *bool*, то есть являются предикатами (табл. 6.7).

Таблица 6.7. Предикаты стандартной библиотеки

Операция	Эквивалентный предикат (функциональный объект)	Операция	Эквивалентный предикат (функциональный)
----------	--	----------	---


```
}
```

Обратные итераторы

Эта разновидность итераторов (`reverse_iterator`) удобна для прохода по контейнеру от конца к началу. Например, для вывода содержимого контейнера `vector<double> v1` в обратном порядке можно написать

```
vector<double>::reverse_iterator ri;  
ri = v1.rbegin();  
while ( ri != v1.rend() ) cout << *ri++ << ' ' ;
```

Операция инкремента для такого итератора перемещает указатель на предыдущий элемент контейнера.

Итераторы вставки и алгоритм `copy`

Мы можем использовать алгоритм `copy` для копирования элементов одного контейнера в другой, причем источником может быть, например, вектор, а приемником — список, как показывает следующий фрагмент:

```
int main() {  
    int a[4] = { 10, 20, 30, 40 };  
    vector<int> v( a, a + 4 );  
    list<int> L( 4 ); // список из 4 элементов  
    copy( v.begin(), v.end(), L.begin() );  
    print( L ); // функция из листинга 15.2  
}
```

Алгоритм `copy` при таком использовании, как в этом примере, работает в *режиме замещения*. Это означает, что *i*-й элемент контейнера-источника замещает *i*-й элемент контейнера-приемника (это напоминает режим замены при вводе текста в текстовом редакторе). Этот же алгоритм может работать и в *режиме вставки*, если третьим аргументом использовать так называемый *итератор вставки*.

Итераторы вставки `front_inserter`, `back_inserter`, `inserter` предназначены для добавления новых элементов в начало, конец или произвольное место контейнера. Покажем использование этих итераторов на следующем примере.

```
int main() {  
    int a[4] = { 40, 30, 20, 10 }; vector<int> va( a, a + 4 );  
    int b[3] = { 80, 90, 100 }; vector<int> vb( b, b + 3 );  
    int c[3] = { 50, 60, 70 }; vector<int> vc( c, c + 3 );  
    list<int> L; // пустой список  
    copy( va.begin(), va.end(), front_inserter(L) ); // 1  
    print( L ); // функция из листинга 15.2  
    vb.end(), back_inserter(L) ); // 2  
    print( L );  
    list<int>::iterator from = L.begin();  
    advance( from, 4 );  
    copy( vc.begin(), vc.end(), inserter( L, from) ); // 3  
    print( L );  
}
```

Результат выполнения программы:

```
10 20 30 40  
10 20 30 40 80 90 100  
10 20 30 40 50 60 70 80 90 100
```

В *операторе 1* выполняется копирование (вставка) вектора `va` в список `L`. Итератор вставки `front_inserter` обеспечивает вставку очередного элемента вектора `va` в начало списка, поэтому порядок элементов в списке изменяется на обратный.

В *операторе 2* элементы вектора `vb` пересылаются в конец списка `L` благодаря итератору вставки `back_inserter` (порядок копируемых элементов не меняется). В *операторе 3* вектор `vc` копируется в заданное итератором `from` место списка `L`, а именно, после четвертого элемента списка. Чтобы определить нужное значение итератора `from`, мы предварительно устанавливаем его в начало списка, а затем обеспечиваем приращение на 4 вызовом функции `advance`.

Алгоритм `merge`

Алгоритм `merge` выполняет слияние отсортированных последовательностей для любого типа последовательного контейнера, более того — все три участника алгоритма могут представлять различные контейнерные типы. Например, вектор `a` и массив `b` могут быть слиты в список `c`:

```
int main() {
```

```

int arr[5] = { 2, 3, 8, 20, 25 };
vector<int> a( arr, arr + 5 );
int b[6] = { 7, 9, 23, 28, 30, 33 };
list<int> c; // Список с сначала пуст
merge( a.begin(), a.end(), b, b + 6, back_inserter(c) );
print( c ); // функция из листинга 6.2
} // Результат выполнения программы: 2 3 7 8 9 20 23 25 28 30 33

```

Использование ассоциативных контейнеров

В ассоциативных контейнерах элементы не выстроены в линейную последовательность. Они организованы в более сложные структуры, что дает большой выигрыш в скорости поиска. Поиск выполняется с помощью *ключа*, обычно представляющего собой одно число или строку. Рассмотрим две основные категории ассоциативных контейнеров в STL: множества и словари (отображения).

В *множестве* (set) хранятся объекты, упорядоченные по некоторому ключу, являющемуся атрибутом самого объекта. Например, множество может хранить объекты класса Map, упорядоченные в алфавитном порядке по значению ключевого поля name. Если в множестве хранятся значения одного из встроженных типов, например int, то ключом является сам элемент.

Словарь (map) можно представить себе как своего рода таблицу из двух столбцов, в первом из которых хранятся объекты, содержащие ключи, а во втором — объекты, содержащие значения. Похожая организация данных рассматривалась нами в задаче 3.1 (шаблонный класс для разреженных массивов).

И в множествах, и в словарях все ключи являются уникальными (только одно значение соответствует ключу). *Мультимножества* (multiset) и *мультисловари* (multimap) аналогичны своим родственным контейнерам, но в них одному ключу может соответствовать несколько значений.

Ассоциативные контейнеры имеют много общих методов с последовательными контейнерами, но некоторые методы и алгоритмы характерны только для них.

Множества

Шаблон множества имеет два параметра: тип ключа и тип функционального объекта, определяющего отношение «меньше»:

```

template <class Key, class Compare = less<Key> >
class set{ /* ... */ };

```

Таким образом, если объявить некоторое множество set<int> s1 с опущенным вторым параметром шаблона, то по умолчанию для упорядочения членов множества будет использован предикат less<int>. Точно так же можно опустить второй параметр при объявлении множества set<MyClass> s2, если в классе MyClass определена операция operator<()>.

Для использования контейнеров типа set необходимо подключить заголовочный файл <set>. Имеется три способа определить объект типа set:

```

set<int> set1; // создается пустое множество
int a[5] = { 1, 2, 3, 4, 5 };
set<int> set2( a, a + 5 ); // инициализация копированием массива
// инициализация другим множеством

```

Для вставки элементов в множество можно использовать метод insert, для удаления — метод erase. Также к множествам применимы общие для всех контейнеров методы, указанные в табл. 6.2.

Во всех ассоциативных контейнерах есть метод count, возвращающий количество объектов с заданным ключом. Так как и в множествах, и в словарях все ключи уникальны, то метод count возвращает либо 0, если элемент не обнаружен, либо 1. Для множеств библиотека содержит некоторые специальные алгоритмы, в частности, реализующие традиционные теоретико-множественные операции. Эти алгоритмы перечислены ниже.

Алгоритм includes выполняет проверку включения одной последовательности в другую. Результат равен true в случае, когда каждый элемент последовательности [first2, last2) содержится в последовательности [first1, last1).

Алгоритм set_intersection создает отсортированное *пересечение множеств*, то есть множество, содержащее только те элементы, которые одновременно входят и в первое, и во второе множество.

Алгоритм set_union создает отсортированное *объединение множеств*, то есть множество, содержащее элементы первого и второго множеств без повторяющихся элементов. Использование этих алгоритмов показано в листинге 6.7.

Листинг 6.7. Пример работы с множествами

```
#include <string>
#include <iostream>
#include <list>
#include <bits/stdc++.h>
using namespace std;
template <class T> void print( T& cont ) {
    typename T::const_iterator p = cont.begin();
    if ( cont.empty() ) cout << "Container is empty.";
    for ( p; p != cont.end(); ++p ) cout << *p << ' ';
    cout << endl;
}
int main() {
    const int N = 5;
    string s1[N] = { "Bill", "Jessica", "Ben", "Mary", "Monica"};
    string s2[N] = { "Sju", "Monica", "John", "Bill", "Sju" };
    typedef set<string> SetS;
    SetS A( s1, s1 + N );
    SetS B( s2, s2 + N );
    print( A ); print( B );
    SetS prod, sum;
    set_intersection( A.begin(), A.end(), B.begin(), B.end(),
                     inserter( prod, prod.begin() ) );
    print( prod ); // функция из листинга 6.2
    set_union( A.begin(), A.end(), B.begin(), B.end(),
              inserter( sum, sum.begin() ) );
    print( sum );
    if ( includes( A.begin(), A.end(), prod.begin(), prod.end() ) )
        cout << "Yes" << endl;
    else cout << "No" << endl;
}
```

Результат выполнения программы:

```
Ben Bill Jessica Mary Monica
Bill John Monica Sju
Bill Monica
Ben Bill Jessica John Mary Monica Sju
Yes
```

Словари

В определении класса `map` используется тип `pair`, который описан в заголовочном файле `<utility>` следующим образом:

```
template <class T1, class T2> struct pair{
    T1 first;
    T2 second;
    pair( const T1& x, const T2& y );
    . . .
};
```

Шаблон `pair` имеет два параметра, представляющих собой типы элементов пары. Первый элемент пары имеет имя `first`, второй — `second`. В этом же файле определены шаблонные операции `==`, `!=`, `<`, `>`, `<=`, `>=` для двух объектов типа `pair`.

Шаблон словаря имеет три параметра: тип ключа, тип элемента и тип функционального объекта, определяющего отношение «меньше»:

```
template <class Key, class T, class Compare = less<Key> >
class map {
public:
    typedef pair <const Key, T> value_type;
    explicit map ( const Compare& comp = Compare() );
    map( const value_type* first, const value_type* last,
         const Compare& comp = Compare() );
    map( const map <Key, T, Compare>& x );
    . . .
};
```

Тип элементов словаря `value_type` определяется как пара элементов типа `Key` и `T`. Первый конструктор класса `map` создает пустой словарь. Второй — создает словарь и записывает в него

элементы, определяемые диапазоном `[first, last)`. Третий конструктор является конструктором копирования.

Для доступа к элементам по ключу определена операция `[]`:

```
T& operator[]( const Key & x );
```

С помощью нее можно не только получать значения элементов, но и добавлять в словарь новые. Для использования контейнеров типа `map` необходимо подключить заголовочный файл `<map>`.

Задача 6.2. Формирование частотного словаря

Написать программу формирования частотного словаря появления отдельных слов в некотором тексте. Исходный текст читается из файла `prose.txt`, результат (частотный словарь) записывается в файл `freq_map.txt`.

Текст программы приведен в листинге 6.8.

Листинг 6.8. Формирование частотного словаря

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <map>
#include <set>
#include <string>
using namespace std;
int main() {
    char punct[6] = { '.', ',', '?', '!', ':', ';' };
    set<char> punctuation( punct, punct + 6 );
    ifstream in( "prose.txt" );
    if ( !in ) { cerr << "File not found\n"; exit( 1 ); }
    map<string, int> wordCount;
    string s;
    while ( in >> s ) {
        int n = s.size();
        if ( punctuation.count( s[n - 1] ) ) s.erase( n - 1, n );
        ++wordCount[s];
    }
    ofstream out( "freq_map.txt" );
    map<string, int>::const_iterator it = wordCount.begin();
    for ( it; it != wordCount.end(); ++it )
        out << setw( 20 ) << left << it->first
            << setw( 4 ) << right << it->second << endl;
}
```

Определяя объект `wordCount` как словарь типа `map<string, int>`, мы показываем намерение связать каждое прочитанное слово с целочисленным счетчиком. В цикле `while` разворачиваются следующие события.

- В строку `s` пословно считываются данные из входного файла.
- Определяется длина `n` строки `s`.
- С помощью метода `count` проверяется, принадлежит ли последний символ строки `s` множеству `punctuation`, содержащему знаки препинания, которыми может завершаться слово. Если да, то последний символ удаляется из строки (метод `erase`).
- Заслуживает особого внимания лаконичная инструкция `++wordCount[s]`. Здесь мы как бы «заглядываем» в объект `wordCount`, используя только что считанное слово в качестве ключа. Результат выражения `wordCount[s]` представляет собой некоторое целочисленное значение, обозначающее, сколько раз слово `s` уже встречалось ранее. Затем операция инкремента увеличивает это целое значение на единицу. А что будет, если мы встречаем некоторое слово в первый раз? Если в словаре нет элемента с таким ключом, он будет создан с инициализацией поля типа `int` значением по умолчанию, то есть нулем. Следовательно, после операции инкремента это значение будет равно единице.

Завершив считывание входных данных и формирование словаря `wordCount`, мы должны вывести в выходной файл `freq_map.txt` значения обнаруженных слов и соответствующих им счетчиков. Вывод результатов реализуется здесь практически так же, как и для последовательных контейнеров — с помощью соответствующего итератора. Однако есть тонкость, связанная с тем, что при разыменовании итератора `map`-объекта мы получаем значение, которое имеет тип `pair`, соответствующий данному `map`-

объекту. Так как `pair` — это структура, доступ к ее полям через «указатель» `it` выполняется посредством выражений `it->first`, `it->second`.

Рассмотрим теперь более сложную задачу, чтобы продемонстрировать применение принципов ООП на практике и удобства, которые дает использование STL.

Задача 6.3. Морской бой

Написать программу, реализующую упрощенную версию игры «Морской бой» между двумя игроками: пользователь и компьютер. Упрощения: все корабли размещаются только вертикально; размещение кораблей — случайное у обоих игроков.

Тем, у кого было тяжелое детство, израненное компьютером, напомним правила:

- Имеется два игровых поля: «свое» и «противника», каждое 10×10 клеток. У каждого игрока по 10 кораблей: один четырехпалубный (состоящий из четырех клеток), два трехпалубных (из трех клеток), три двухпалубных (из двух), четыре однопалубных (из одной клетки). При расстановке кораблей они не должны касаться друг друга (находиться в соседних клетках).
- Каждый игрок видит размещение кораблей на своем игровом поле, но не имеет информации о размещении кораблей на поле противника.
- После расстановки кораблей игроки начинают «стрелять» друг в друга. Для этого стреляющий выбирает клетку на поле противника и объявляет ему ее координаты (A1, E5 и т. д.). Противник смотрит на своем поле, находится ли по указанным координатам его корабль, и сообщает результат выстрела:
 - *промах* — на данной клетке нет корабля противника;
 - *ранен (поврежден)* — на данной клетке есть корабль противника с хотя бы еще одной непораженной клеткой (палубой);
 - *убит* — на данной клетке есть корабль противника, и все его клетки (палубы) уже поражены.
- В случае попадания в корабль противника игроку дается право на внеочередной выстрел, в противном случае ход переходит к противнику.
- Стрельба ведется до тех пор, пока у одного из игроков не окажутся «убитыми» все корабли (он признается проигравшим, а его противник — победителем). Так как мы пишем консольное приложение, доступные нам графические средства сильно ограничены — это текстовые символы и символы псевдографики. Примем решение, что после некоторого хода играющих картинка в консольном окне будет иметь примерно такой вид, какой показан на рис.6.1.

Изображенные на рисунке игровые поля «Мой флот» и «Флот неприятеля» отображают текущее состояние игры со стороны пользователя. Изначальное размещение кораблей на поле пользователя — в клетках, помеченных символом «заштрихованный прямоугольник» (символ 176 в кодовой таблице cp866/MS DOS). Символом «.» (точка) обозначены свободные клетки, по которым еще ни разу не стреляли, символом «O» — промахи стреляющих, символом «X» — пораженные клетки (палубы) кораблей. Пробелами обозначены клетки, в которых по правилам уже не могут находиться корабли противника. Эти «мертвые зоны» выявляются после гибели очередного корабля.



Рис. 6.1. Возможный вид консольного окна после i -го хода ($i = 31$)

После сделанных разъяснений мы можем приступить к решению задачи, и начнем, как всегда, с выявления понятий (классов) и их взаимосвязей.

Итак, имеется два игрока: пользователь (`User`) и компьютер, выступающий в роли робота (`Robot`). Каждый игрок «управляет» своим собственным флотом, поэтому логично создать два класса: `UserNavy` (флот пользователя) и `RobotNavy` (флот робота). Очевидно, что они обладают различным

поведением — например, метод `FireOff` (выстрел по неприятелю) в первом классе должен пригласить пользователя ввести координаты выстрела, а во втором классе — автоматически сформировать координаты выстрела, сообразуясь с искусственным интеллектом робота. В то же время в этих классах есть и общие атрибуты, например, игровые поля (свое и неприятеля), корабли своего флота и т. д. Поэтому выделим все общие поля и методы в базовый класс `Navy`, который будут наследовать классы `UserNavy` и `RobotNavy`.

Каждый флот состоит из кораблей, отсюда вытекает потребность в классе `Ship`, объекты которого инкапсулируют такую информацию, как координаты размещения корабля, имя корабля, общее количество палуб, количество неповрежденных палуб.

Для описания размещения кораблей воспользуемся классом `Rect`, который позволяет задать любой прямоугольник в двумерном дискретном пространстве. Конечно, наши прямоугольники в формализме двумерного дискретного пространства вырождаются в линию, но такое описание удобно для единообразного представления как вертикально, так и горизонтально размещенных кораблей⁵.

Игровое поле (двумерное дискретное пространство) состоит из клеток (точек двумерного пространства), для представления которых используем класс `Cell`. Наконец, игроки должны обмениваться информацией (координаты и результаты очередного выстрела). Для моделирования процесса обмена информацией создадим класс `Space`, поля которого будут использоваться как глобальные переменные, поэтому они должны быть статическими, а сам класс — базовым для класса `Navy`.

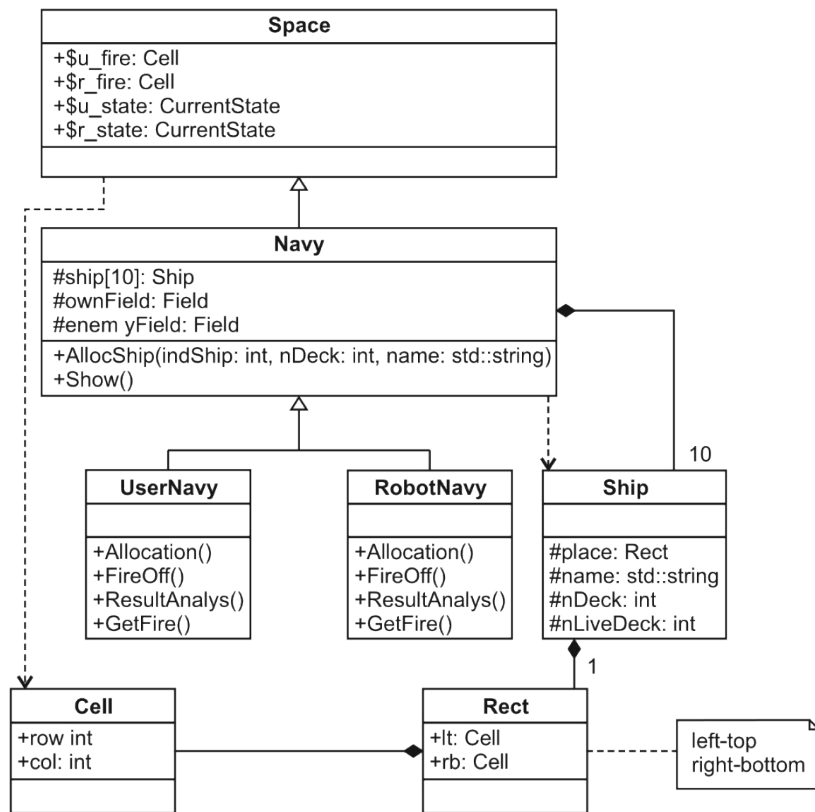


Рис. 6.2. Диаграмма классов для задачи 6.3

На рис. 6.2 показана диаграмма классов, обобщающая наши рассуждения по составу и взаимоотношениям классов для решения рассматриваемой задачи. Текст программы приводится в листинге 6.9.

Листинг 6.9. «Морской бой»

```

//////////////////////////////////// Ship.h
#ifndef SHIP_H
#define SHIP_H
#include <set>
#include <map>
#include <string>

```

⁵ Хотя условие имеет упрощающее ограничение «все корабли размещаются вертикально», при проектировании структур данных рекомендуется учитывать возможность последующей модификации программы, снимающей это ограничение.

```

#define N 10 // размер поля для размещения флота (N * N клеток)
struct Cell;
typedef std::set<Cell> CellSet; // множество клеток
struct Cell { // ----- Клетка (ячейка) на игровом поле
    Cell( int _r = 0, int _c = 0 ) : row( _r ), col( _c ) {}
    bool InSet( const CellSet& ) const; // определяет
        // принадлежность клетки множеству типа CellSet
    bool operator<( const Cell& ) const;
    int row; // ряд
    int col; // колонка
};
struct Rect { // --- Прямоугольная область (размещение кораблей и их "оболочек")
    Rect() {}
    Rect( Cell _lt, Cell _rb ) : lt( _lt ), rb( _rb ) { FillCset(); }
    void FillCset(); // заполнить cset клетками данной области
    bool Intersect( const CellSet& cs ) const; // определить наличие
        // непустого пересечения прямоугольника с множеством cs
    Cell lt; // left-top клетка
    Cell rb; // right-bottom клетка
    CellSet cset; // множество клеток, принадлежащих прямоугольнику
};
class Ship { // ----- Класс Ship (для представления корабля)
    friend class UserNavy;
    friend class RobotNavy;
public:
    Ship() : nDeck( 0 ), nLiveDeck( 0 ) {}
    Ship( int, std::string, Rect );
protected:
    Rect place; // координаты размещения
    std::string name; // имя корабля
    int nDeck; // количество палуб
    int nLiveDeck; // количество неповрежденных палуб
};
#endif /* SHIP_H */
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// Ship.cpp
#include <string>
#include <algorithm>
#include "Ship.h"
using namespace std;
bool Cell::InSet( const CellSet& cs ) const { //----- Класс Cell
    return ( cs.count( Cell( row, col ) ) > 0 );
}
bool Cell::operator<( const Cell& c ) const {
    return ( ( row < c.row ) || ( ( row == c.row ) && ( col < c.col ) ) );
}
void Rect::FillCset() { // ----- Класс Rect
    for ( int i = lt.row; i <= rb.row; i++ )
        for ( int j = lt.col; j <= rb.col; j++ )
            cset.insert( Cell( i, j ) );
}
bool Rect::Intersect( const CellSet& cs ) const {
    CellSet common_cell;
    set_intersection( cset.begin(), cset.end(), cs.begin(), cs.end(),
        inserter( common_cell, common_cell.begin() ) );
    return ( common_cell.size() > 0 );
}
Ship::Ship( int _nDeck, string _name, Rect _place ) : // Класс Ship
    place( _place ), name( _name ), nDeck( _nDeck ), nLiveDeck( _nDeck ) {}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// Navy.h
#include "Ship.h"
#define DECK 176 // исправная клетка-палуба
#define DAMAGE 'X' // разрушенная клетка-палуба
#define MISS 'o' // пустая клетка, в к-рую упал снаряд
typedef unsigned char Field[N][N]; // игровое поле
typedef std::map<Cell, int> ShipMap; // словарь
// ассоциаций "клетка - индекс корабля"
enum CurrentState { Miss, Damage, Kill }; // результат попадания в цель

```

```

struct Space { // -- Класс Space - информационное пространство для обмена
public:
    static Cell u_fire;           // информация между игроками
    static Cell r_fire;           // огонь от пользователя
    static CurrentState u_state;  // огонь от робота (компьютера)
    static CurrentState r_state;  // состояние пользователя
    static int step;              // состояние робота
};
class Navy : public Space { //----- Базовый класс Navy
public:
    Navy();
    void AllocShip( int, int, std::string ); // разместить корабль
    void Show() const;                       // показать поля ownField и enemyField
    int GetInt();                             // ввод целого числа
    bool IsLive() { return ( nLiveShip > 0 ); } // мы еще живы?
    Rect Shell( Rect ) const; /* вернуть "оболочку" для заданного
                               прямоугольника
    (сам прямоугольник плюс пограничные клетки) */
    void AddToVetoSet( Rect ); // добавить клетки прямоугольника
                               // в множество vetoSet.
protected:
    Ship ship[10]; // корабли флота
    Field ownField; // мое игровое поле
    Field enemyField; // игровое поле неприятеля
    ShipMap shipMap; // словарь ассоциаций "клетка - индекс корабля"
    CellSet vetoSet; // множество "запрещенных" клеток
    CellSet crushSet; // множество "уничтоженных" клеток
    int nLiveShip; // количество боеспособных кораблей
};
class UserNavy : public Navy { //----- Класс UserNavy
public:
    UserNavy() { Allocation(); }
    void Allocation();
    void FireOff(); // выстрел по неприятелю
    void ResultAnalys(); // анализ результатов выстрела
    void GetFire(); // "прием" огня противника
    void FillDeadZone( Rect r, Field& ); // заполнить
    // пробелами пограничные клетки для r
};
class RobotNavy : public Navy { //----- Класс RobotNavy
public:
    RobotNavy();
    void Allocation();
    void FireOff(); // выстрел по неприятелю
    void ResultAnalys(); // анализ результатов выстрела
    void GetFire(); // "прием" огня противника
private:
    bool isCrushContinue; // предыдущий выстрел был успешным
    bool upEmpty; // у поврежденного корабля противника
    // нет "живых" клеток в верхнем направлении
};
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// Navy.cpp
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <algorithm>
#include "Navy.h"
using namespace std;
Cell Space::u_fire;
Cell Space::r_fire;
CurrentState Space::u_state = Miss;
CurrentState Space::r_state = Miss;
int Space::step = 1;
// Функция gap(n) возвращает строку из n пробелов
string gap( int n ) { return string( n, ' ' ); }
Navy::Navy() : nLiveShip( 10 ) { // ----- Класс Navy
    for ( int i = 0; i < N; i++ ) // Заполняем игровые поля символом "точка"

```

```

        for ( int j = 0; j < N; j++ ) {
            ownField[i][j] = '.';
            enemyField[i][j] = '.'; }
    }
    Rect Navy::Shell( Rect r ) const {
        Rect sh( r );
        sh.lt.row = ( --sh.lt.row < 0 ) ? 0 : sh.lt.row;
        sh.lt.col = ( --sh.lt.col < 0 ) ? 0 : sh.lt.col;
        sh.rb.row = ( ++sh.rb.row > ( N - 1 ) ) ? ( N - 1 ) : sh.rb.row;
        sh.rb.col = ( ++sh.rb.col > ( N - 1 ) ) ? ( N - 1 ) : sh.rb.col;
        return sh;
    }
    void Navy::AddToVetoSet( Rect r ) {
        for ( int i = r.lt.row; i <= r.rb.row; i++ )
            for ( int j = r.lt.col; j <= r.rb.col; j++ )
                vetoSet.insert( Cell( i, j ) );
    }
    void Navy::AllocShip( int indShip, int nDeck, string name ) {
        int i, j;
        Cell lt, rb;
        // Генерация случайно размещенной начальной клетки корабля учетом
        // недопустимости "пересечения" нового корабля с множеством клеток vetoSet
        while( 1 ) {
            lt.row = rand( ) % ( N + 1 - nDeck );
            lt.col = rb.col = rand( ) % N;
            rb.row = lt.row + nDeck - 1;
            if ( !Rect( lt, rb ).Intersect( vetoSet ) ) break;
        }
        // Сохраняем данные о новом корабле
        ship[indShip] = Ship( nDeck, name, Rect( lt, rb ) );
        // Заносим новый корабль на игровое поле (символ DECK).
        // Добавляем соответствующие элементы в словарь ассоциаций
        for ( i = lt.row; i <= rb.row; i++ )
            for ( j = lt.col; j <= rb.col; j++ ) {
                ownField[i][j] = DECK;
                shipMap[Cell( i, j )] = indShip;
            }
        // Добавляем в множество vetoSet клетки нового корабля
        // вместе с пограничными клетками
        AddToVetoSet( Shell( Rect( lt, rb ) ) );
    }
    void Navy::Show() const {
        char rowName[10] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'};
        string colName( "1 2 3 4 5 6 7 8 9 10" );
        int i, j;
        cout << "-----\n";
        cout << gap( 3 ) << "Мой флот" << gap( 18 ) << "Флот неприятеля" << endl;
        cout << gap( 3 ) << colName << gap( 6 ) << colName << endl;
        for ( i = 0; i < N; i++ ) {
            string line = gap( 1 ) + rowName[i]; // Own
            for ( j = 0; j < N; j++ )
                line += gap( 1 ) + ( char )ownField[i][j];
            line += gap( 5 ) + rowName[i]; // Enemy
            for ( j = 0; j < N; j++ )
                line += gap( 1 ) + ( char )enemyField[i][j];
            cout << line << endl;
        }
        cout << endl << "=====\n";
        cout << step << ". " << "Мой выстрел: ";
        step++;
    }
    int Navy::GetInt() {
        int value;
        while ( true ) {
            cin >> value;
            if ( '\n' == cin.peek() ) { cin.get(); break; }
            else {

```



```

        cout << "Повторите ввод колонки (ожидается целое число):" << endl;
        cin.clear();
        while ( cin.get() != '\n' ) {};
    }
}
return value;
}
void UserNavy::Allocation() { // -----Класс UserNavy
    srand( ( unsigned )time( NULL ) );
    AllocShip(0, 4, "Авианосец 'Варяг'"); AllocShip(1, 3, "Линкор 'Муромец'");
    AllocShip(2, 3, "Линкор 'Никитич'"); AllocShip(3, 2, "Крейсер 'Чудный'");
    AllocShip(4, 2, "Крейсер 'Добрый'"); AllocShip(5, 2, "Крейсер 'Справедливый'");
    AllocShip(6, 1, "Миноносец 'Храбрый'"); AllocShip(7, 1, "Миноносец 'Ушлый'");
    AllocShip(8, 1, "Миноносец 'Проворный'"); AllocShip(9,1, "Миноносец 'Смелый'");
    vetoSet.clear();
}
void UserNavy::FillDeadZone( Rect r, Field& field ) {
    int i, j;
    Rect sh = Shell( r );
    for ( i = sh.lt.row, j = sh.lt.col; j <= sh.rb.col; j++ )
        if ( sh.lt.row < r.lt.row ) field[i][j] = ' ';
    for ( i = sh.rb.row, j = sh.lt.col; j <= sh.rb.col; j++ )
        if ( sh.rb.row > r.rb.row ) field[i][j] = ' ';
    for ( j = sh.lt.col, i = sh.lt.row; i <= sh.rb.row; i++ )
        if ( sh.lt.col < r.lt.col ) field[i][j] = ' ';
    for ( j = sh.rb.col, i = sh.lt.row; i <= sh.rb.row; i++ )
        if ( sh.rb.col > r.rb.col ) field[i][j] = ' ';
}
void UserNavy::FireOff() {
    string capital_letter = "ABCDEFGH IJ";
    string small_letter = "abcdefghij";
    unsigned char rowName; // обозначение ряда (A, B, ... , J)
    int colName;           // обозначение колонки (1, 2, ..., 10)
    int row;               // индекс ряда (0, 1, ... , 9)
    int col;               // индекс колонки (0, 1, ... , 9)
    bool success = false;
    while ( !success ) {
        cin >> rowName;
        row = capital_letter.find( rowName );
        if ( -1 == row ) row = small_letter.find( rowName );
        if ( -1 == row ) { cout << "Ошибка. Повторите ввод.\n"; continue; }
        colName = GetInt();
        col = colName - 1;
        if ( ( col < 0 ) || ( col > 9 ) ) {
            cout << "Ошибка. Повторите ввод.\n"; continue;
        }
        success = true;
    }
    u_fire = Cell( row, col );
}
void UserNavy::ResultAnalys() {
    // r_state - сообщение робота о результате выстрела
    // пользователя по клетке u_fire
    switch( r_state ) {
        case Miss:
            enemyField[u_fire.row][u_fire.col] = MISS; break;
        case Damage:
            enemyField[u_fire.row][u_fire.col] = DAMAGE;
            crushSet.insert( u_fire ); break;
        case Kill:
            enemyField[u_fire.row][u_fire.col] = DAMAGE;
            crushSet.insert( u_fire );
            Rect kill;
            kill.lt = *crushSet.begin();
            kill.rb = *( --crushSet.end() );
            FillDeadZone( kill, enemyField ); // Заполняем "обрамление" пробелами
            crushSet.clear();
    }
}

```

```

    }
}
void UserNavy::GetFire() {
    // выстрел робота - по клетке r_fire
    string capital_letter = "ABCDEFGHJIJ";
    char rowName = capital_letter[r_fire.row];
    int colName = r_fire.col + 1;
    cout << "\nВыстрел неприятеля: " << rowName << colName << endl;
    if ( DECK == ownField[r_fire.row][r_fire.col] ) {
        cout << "**** Есть попадание! ****";
        ownField[r_fire.row][r_fire.col] = DAMAGE;
        u_state = Damage;
        int ind = shipMap[r_fire]; // индекс корабля, занимающего клетку r_fire
        ship[ind].nLiveDeck--;
        if ( !ship[ind].nLiveDeck ) {
            u_state = Kill;
            cout << gap( 6 ) << "О ужас! Погиб " << ship[ind].name << " !!!";
            nLiveShip--;
            Rect kill = ship[ind].place;
            FillDeadZone( kill, ownField );
        }
    }
    else {
        u_state = Miss;
        cout << "**** МИМО! ****";
        ownField[r_fire.row][r_fire.col] = MISS;
    }
    cout << endl;
}
RobotNavy::RobotNavy() { // ----- Класс RobotNavy
    Allocation();
    isCrushContinue = false;
    upEmpty = false;
}
void RobotNavy::Allocation() {
    AllocShip(0, 4, "Авианосец 'Алькаида'"); AllocShip( 1, 3, "Линкор 'БенЛаден'");
    AllocShip(2, 3, "Линкор 'Хусейн'");      AllocShip(3, 2, "Крейсер 'Подлый'");
    AllocShip(4, 2, "Крейсер 'Коварный'");   AllocShip(5, 2, "Крейсер 'Злой'");
    AllocShip(6, 1, "Миноносец 'Гадкий'");   AllocShip(7, 1, "Миноносец 'Мерзкий'");
    AllocShip(8, 1, "Миноносец 'Пакостный'"); AllocShip(9, 1, "Миноносец 'Душный'");
    vetoSet.clear();
}
void RobotNavy::FireOff() {
    Cell c, cUp;
    if ( !isCrushContinue ) {
        while( 1 ) { // случайный выбор координат выстрела
            c.row = rand() % N;
            c.col = rand() % N;
            if ( !c.InSet( vetoSet ) ) break;
        }
    }
    else {
        c = cUp; // "пляшем" от предыдущего попадания
        cUp.row--;
        if ( ( !upEmpty ) && c.row && ( !cUp.InSet( vetoSet ) ) ) c.row--;
        else { c = *( --crushSet.end() ); c.row++; }
    }
    r_fire = c;
    vetoSet.insert( r_fire );
}
void RobotNavy::ResultAnalys() {
    // u_state - сообщение пользователя о результате
    // выстрела робота по клетке r_fire
    switch( u_state ) {
        case Miss:
            if ( isCrushContinue ) upEmpty = true; break;
        case Damage:
            isCrushContinue = true;
            crushSet.insert( r_fire );
            break;
    }
}

```

```

        case Kill:
            isCrushContinue = false;
            upEmpty = false;
            crushSet.insert( r_fire );
            Rect kill;
            kill.lt = *crushSet.begin();
            kill.rb = *( --crushSet.end() );
            AddToVetoSet( Shell( kill ) );
            crushSet.clear();
        }
    }
}

void RobotNavy::GetFire() {
    // выстрел пользователя - по клетке u_fire
    if ( DECK == ownField[u_fire.row][u_fire.col] ) {
        cout << "**** Есть попадание! ****";
        r_state = Damage;
        int ind = shipMap[u_fire];    // индекс корабля, занимающего клетку u_fire
        ship[ind].nLiveDeck--;
        if ( !ship[ind].nLiveDeck ) {
            r_state = Kill;
            cout << gap( 6 ) << "Уничтожен " << ship[ind].name << " !!!";
            nLiveShip--; }
        }
    else { r_state = Miss; cout << "**** Мимо! ****"; }
    cout << endl;
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////// Main.cpp
#include <iostream>
#include "Navy.cpp"
#include "Ship.cpp"
using namespace std;
int main() {
    UserNavy userNavy;                // Начальная позиция
    RobotNavy robotNavy;
    userNavy.Show();
    while ( userNavy.IsLive() && robotNavy.IsLive() ) {
        if ( Space::u_state != Miss ) {    // -- Выстрел пользователя
            cout << "пропускается...: <Enter>" << endl;
            cin.get(); }
        else {
            userNavy.FireOff();
            robotNavy.GetFire();
            userNavy.ResultAnalys();
            if ( !robotNavy.IsLive() ) { userNavy.Show(); break; }
        }
        if ( Space::r_state != Miss )      // --- Выстрел робота
            cout << "\nВыстрел неприятеля: пропускается..." << endl;
        else {
            robotNavy.FireOff();
            userNavy.GetFire();
            robotNavy.ResultAnalys();
        }
        userNavy.Show();
    }
    if ( userNavy.IsLive() ) cout << "\n:-))) Ура! Победа!!! :-)))" << endl;
    else { cout << "\n:-((( Увы. Непрятель оказался сильнее." << endl;
        cout << ":-((( Но ничего, в следующий раз мы ему покажем!!!" << endl;
    }
    cin.get();
}

```

Обратите внимание на использование объектов контейнерных классов: cset типа set<Cell> в классе Rect; vetoSet и crushSet типа set<Cell> в классе Navy; shipMap типа map<Cell, int> в классе Navy.

В качестве упражнения рекомендуем вам доработать эту программу, сняв ограничение «только вертикальное расположение кораблей» (решение об ориентации размещаемого корабля принимать случайным образом).

Итоги

1. Стандартная библиотека шаблонов содержит общецелевые классы и функции, которые реализуют широко используемые алгоритмы и структуры данных.
2. STL построена на основе шаблонных классов, поэтому входящие в нее алгоритмы и структуры могут настраиваться на различные типы данных.
3. Использование STL позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки.
4. Везде, где это возможно, используйте классы и алгоритмы STL!

Задания

Вариант 1

Написать программу для моделирования T-образного сортировочного узла на железной дороге с использованием контейнерного класса `stack` из STL.

Программа должна разделять на два направления состав, состоящий из вагонов двух типов (на каждое направление формируется состав из вагонов одного типа). Предусмотреть возможность ввода исходных данных с клавиатуры и из файла.

Вариант 2

Написать программу, отыскивающую проход по лабиринту, с использованием контейнерного класса `stack` из STL.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Программа находит проход через лабиринт, двигаясь от заданного входа. После отыскания прохода программа выводит найденный путь в виде координат квадратов.

Вариант 3

Написать программу, моделирующую управление каталогом в файловой системе. Для каждого файла в каталоге содержатся следующие сведения: имя файла, дата создания, количество обращений к файлу. Программа должна обеспечивать:

- начальное формирование каталога файлов;
- вывод каталога файлов;
- удаление файлов, дата создания которых раньше заданной;
- выборку файла с наибольшим количеством обращений.

Выбор моделируемой функции должен осуществляться с помощью меню. Для представления каталога использовать контейнерный класс `list` из STL.

Вариант 4

Написать программу моделирования работы автобусного парка. Сведения о каждом автобусе содержат: номер автобуса, фамилию и инициалы водителя, номер маршрута. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- начальное формирование данных о всех автобусах в парке в виде списка (ввод с клавиатуры или из файла);
- имитация выезда автобуса из парка: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;
- имитация въезда автобуса в парк: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;
- вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте.
- Для представления необходимых списков использовать контейнерный класс `list`.

Вариант 5

Написать программу учета заявок на авиабилеты. Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- добавление заявок в список;
- удаление заявок;
- вывод заявок по заданному номеру рейса и дате вылета;
- вывод всех заявок.

Для хранения данных использовать контейнерный класс `list`.

Вариант 6

Написать программу учета книг в библиотеке. Сведения о книгах содержат: фамилию и инициалы автора, название, год издания, количество экземпляров данной книги в библиотеке. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- добавление данных о книгах, вновь поступающих в библиотеку;
- удаление данных о списываемых книгах;
- выдача сведений о всех книгах, упорядоченных по авторам;
- выдача сведений о всех книгах, упорядоченных по годам издания.

Хранение данных организовать с применением контейнерного класса `multimap`, в качестве ключа использовать «фамилию и инициалы автора».

Вариант 7

Написать программу «Моя записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Хранение данных организовать с помощью класса `map` или `multimap`.

Вариант 8

Написать программу учета заявок на обмен квартир и поиска вариантов обмена. Каждая заявка содержит сведения о двух квартирах: требуемой (искомой) и имеющейся. Сведения о каждой квартире содержат: количество комнат, площадь, этаж, район. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ввод заявки на обмен;
- поиск в картотеке подходящего варианта: при совпадении требований и предложений по количеству комнат и этажности и различии по показателю «площадь» в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в картотеку;
- вывод всей картотеки.

Для хранения данных картотеки использовать контейнерный класс `list`.

Вариант 9

Написать программу «Автоматизированная информационная система на железнодорожном вокзале». Информационная система содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается: номер поезда, станция назначения, время отправления. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- первоначальный ввод данных в систему (с клавиатуры или из файла);
- вывод сведений по всем поездам;
- вывод сведений по поезду с запрошенным номером;
- вывод сведений по поездам, следующим до запрошенной станции назначения.

Хранение данных организовать с применением контейнерного класса `vector`.

Вариант 10

Написать программу «Англо-русский и русско-английский словарь». «База данных» словаря должна содержать синонимичные варианты перевода слов. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- загрузка «базы данных» словаря (из файла);
- выбор режима работы: англо-русский или русско-английский;
- вывод вариантов перевода заданного английского слова;
- вывод вариантов перевода заданного русского слова.

Базу данных словаря реализовать в виде двух контейнеров типа `map`.

Вариант 11

Написать программу, реализующую игру «Крестики-нолики» между двумя игроками: пользователь и компьютер (робот). В программе использовать контейнеры `STL`.

Вариант 12

Написать программу, решающую игру-головоломку «Игра в 15». Начальное размещение номеров — случайное. Предусмотреть два режима демонстрации решения: непрерывный (с некоторой задержкой визуализации) и пошаговый (по нажатию любой клавиши). В программе использовать контейнерные классы STL.

Вариант 13

Составить программу формирования списка кандидатов, участвующих в выборах губернатора. Каждая заявка от кандидата содержит: фамилию и инициалы, дату рождения, место рождения, индекс популярности. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- добавление заявки в список кандидатов. Для ввода индекса популярности (значение указано в скобках) предусмотреть выбор с помощью подменю одного из следующих вариантов: поддержан президентом (70);
- поддержан оппозиционной партией (15);
- оппозиционный кандидат, который снимет свою кандидатуру в пользу кандидата № 1 (10);
- прочие (5).
- удаление заявки по заявлению кандидата;
- формирование и вывод списка для голосования.

Хранение данных организовать с применением контейнерного класса `priority_queue` из STL. Для надлежащего функционирования очереди с приоритетами побеспокоиться о надлежащем определении операции `<` (меньше) в классе, описывающем заявку кандидата. Формирование и вывод списка для голосования реализовать посредством выборки заявок из очереди.

Вариант 14

Составить программу моделирования работы автобусного парка. Сведения о каждом автобусе содержат: номер автобуса, фамилию и инициалы водителя, номер маршрута. Следует обеспечить выбор с помощью меню и выполнение функций:

- начальное формирование данных о всех автобусах в парке в виде списка (ввод с клавиатуры или из файла);
- имитация выезда автобуса из парка: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;
- имитация въезда автобуса в парк: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;
- вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте, упорядоченных по номерам автобусов;
- вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте, упорядоченных по номерам маршрутов.

Хранение всех необходимых списков организовать с применением контейнерного класса `map`, в качестве ключа использовать «номер автобуса».

Вариант 15

Составить программу учета заявок на авиабилеты. Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- добавление заявок в список;
- удаление заявок;
- вывод заявок по заданному номеру рейса и дате вылета;
- вывод всех заявок, упорядоченных по пунктам назначения;
- вывод всех заявок, упорядоченных по датам вылета.

Хранение данных организовать с применением контейнерного класса `multimap`, в качестве ключа использовать «пункт назначения».

Вариант 16

Написать программу учета книг в библиотеке. Сведения о книгах содержат: фамилию и инициалы автора, название, год издания, количество экземпляров данной книги в библиотеке. Следует обеспечить выбор с помощью меню и выполнение функций:

- добавление данных о книгах, вновь поступающих в библиотеку;
- удаление данных о списываемых книгах;
- выдача сведений о всех книгах, упорядоченных по авторам;
- выдача сведений о всех книгах, упорядоченных по годам издания.

Хранение данных организовать с применением контейнерного класса `vector`.

Вариант 17

Написать программу «Моя записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Хранение данных организовать с применением контейнерного класса `list`.

Вариант 18

Написать программу учета заявок на обмен квартир и поиска вариантов обмена. Каждая заявка содержит фамилию и инициалы заявителя, а также сведения о двух квартирах: требуемой (искомой) и имеющейся. Сведения о каждой квартире содержат: количество комнат, площадь, этаж, район. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ввод заявки на обмен;
- поиск в картотеке подходящего варианта: при совпадении требований и предложений по количеству комнат и этажности и при различии по показателю «площадь» в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в картотеку;
- вывод всей картотеки.

Хранение данных организовать с применением контейнерного класса `set`.

Вариант 19

Написать программу «Автоматизированная информационная система на железнодорожном вокзале». Информационная система содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается: номер поезда, станция назначения, время отправления. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- первоначальный ввод данных в информационную систему (с клавиатуры или из файла);
- вывод сведений по всем поездам;
- вывод сведений по поезду с запрошенным номером;
- вывод сведений по поездам, следующим до запрошенной станции назначения. Хранение данных организовать с применением контейнерного класса `set`.

Вариант 20

Написать программу «Англо-русский и русско-английский словарь». «База данных» словаря должна содержать синонимичные варианты перевода слов. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- загрузка «базы данных» словаря (из файла);
- выбор режима работы: англо-русский или русско-английский;
- вывод вариантов перевода заданного английского слова;
- вывод вариантов перевода заданного русского слова.

Базу данных словаря реализовать в виде двух контейнеров типа `set`.