МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Курганский государственный университет»

А. В. Маер

ВВЕДЕНИЕ В СТАНДАРТНУЮ БИБЛИОТЕКУ ШАБЛОНОВ (STL)

Учебное пособие

Курган 2020

УДК 519.685.7
ББК 32.973
М 13

Рецензенты:

кандидат технических наук, доцент кафедры информатики и вычислительной техники ФГБОУ ВО «Омский государственный технический университет» А. С. Грицай;

кандидат физико-математических наук, доцент кафедры оптико-электронных систем и дистанционного зондирования ФГАОУ ВО «Национальный исследовательский Томский государственный университет» В. В. Брюханова.

Печатается по решению методического совета Курганского государственного университета.

Маер, А. В.

Введение в стандартную библиотеку шаблонов (STL) : учебное пособие / А. В. Маер. – Курган : Изд-во Курганского гос. ун-та, 2020. – 87 с.

Учебное пособие состоит из четырех глав. Каждая глава содержит набор заданий для выполнения учебных проектов. Первая глава посвящена последовательным и отсортированным ассоциативным контейнерам. Вторая и третья главы рассказывают об итераторах и обобщенных алгоритмах. Четвертая глава расширяет функциональные возможности STL за счет введения функциональных объектов и адаптеров.

Учебное пособие предназначено для студентов направлений: 09.03.03, 09.03.04.

ISBN 978-5-4217-0540-6

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Стандартная библиотека шаблонов (Standard Template Library, STL) представляет набор классов-контейнеров C++ и шаблонные алгоритмы, которые разработаны для совместного использования и обеспечивают полезную функциональность в широком диапазоне. Хотя в библиотеке представлено лишь небольшое количество классов-контейнеров, она включает все наиболее часто применяемые и полезные контейнеры, такие как векторы, списки, множества и ассоциативные множества. Множество алгоритмов включает широкий диапазон фундаментальных алгоритмов для наиболее распространенных манипуляций с данными, таких как поиск, сортировка и слияние.

STL разработана с учетом четырех фундаментальных идей:

- обобщенное программирование;
- абстрактность без потери эффективности;
- вычислительная модель фон Неймана;
- семантика значений.

Обобщенное программирование. Некоторые из вас могли слышать, что STL представляет собой пример технологии программирования, именуемой «обобщенным программированием». Это так. Некоторые из вас могли слышать, что обобщенное программирование – это стиль программирования с применением шаблонов C++. Это не так. Обобщенное программирование не имеет никакого отношения к C++ и шаблонам. Обобщенное программирование – это предмет, изучающий систематическую организацию полезных программных компонентов. Его целью является разработка систематики алгоритмов, структур данных, механизмов распределения памяти и прочих программных артефактов таким образом, чтобы обеспечить максимальный уровень повторного использования, модульности и удобства.

Абстрактность без потери эффективности. Математикам, бывает, приходится работать с объектами, которые не могут быть сконструированы вообще или могут быть сконструированы только за произвольно большое время. В области же вычислительной техники эффективность играет важную роль. Недостаточно знать, что

некоторая операция может быть выполнена – важно знать, что она может быть выполнена за разумное время.

Вычислительная модель фон Неймана. Хотя абстрактная математика использует простые числовые факты в качестве основы для своих абстракций (не забывайте, что математика – наука экспериментальная), то что в качестве основы для абстракций должны использовать разработчики библиотеки? Есть мнение, что единственной основой является архитектура реальных компьютеров. Важно помнить, что архитектуры современных компьютеров являются результатом многолетней эволюции, руководимой необходимостью решать все более и более разнообразные задачи. Если вас интересует проектирование обобщенных схем для числовых типов, важно не только знать математическую теорию целых и действительных чисел, но и понимать, как работают встроенные числовые типы.

Семантика значений. STL рассматривает контейнеры как обобщение структур. Контейнер, как и структура, владеет своими компонентами. При копировании структур копируются все их компоненты. При уничтожении структуры уничтожаются все ее компоненты. То же самое происходит и с контейнерами. Это важнейшие свойства, которые позволяют структурам и контейнерам моделировать ключевой атрибут объектов реального мира – взаимоотношения между целым и частью. Конечно, это не единственное взаимоотношение в реальном мире, и остальные отношения должны моделироваться итераторами.

1 Контейнеры STL

STL содержит компоненты шести основных видов: контейнеры, обобщенные алгоритмы, итераторы, функциональные объекты, адаптеры и аллокаторы. Контейнеры в STL представляют собой объекты, которые хранят коллекции других объектов. Имеется две категории контейнеров STL: контейнеры последовательностей и отсортированные ассоциативные контейнеры.

1.1 Стандартная библиотека (STL)

Контейнеры последовательностей. Организуют набор объектов одного и того же типа T в строго линейную последовательность. В STL имеются следующие контейнеры последовательностей:

- vector <T>. Вектор обеспечивает произвольный доступ к последовательности переменной длины (произвольный доступ означает, что время, требуемое для достижения i-го элемента, представляет собой константу, т. е. оно не зависит от конкретного значения i) с амортизированным константным временем вставки и удаления в конце последовательности;
- deque <T>. Дек также обеспечивает произвольный доступ к последовательности переменной длины с амортизированным константным временем вставки и удаления с обоих концов последовательности;
- list <T>. Список обеспечивает линейное время доступа к последовательности переменной длины (O(N), где N – текущая длина), но с константным временем вставки и удаления в любом месте последовательности.

Перед рассмотрением этих типов контейнеров последовательностей заметим, что обычный массив C++ T a[N] может использоваться в качестве контейнера последовательности, поскольку все обобщенные алгоритмы STL спроектированы для работы с массивами точно так же, как и с другими типами последовательностей. Имеется еще один важный тип string (из заголовочного файла <string.h>), представляющий последовательности символов способом, совместимым с алгоритмами и соглашениями STL. Например, STL предоставляет обобщенный алгоритм reverse, который может обратить последовательности различного вида, включая объекты string и массивы.

Как вы увидите в дальнейшем, векторы, списки и деки не являются полностью взаимозаменяемыми, но в данном случае все они работают одинаково. Это связано с тем, что каждое определение функций-членов begin и end имеет один и тот же абстрактный смысл, хотя их реализация существенно различна: векторы представлены с применением массивов, списки – с использованием дважды связанных

узлов, а деки обычно реализуются при помощи двухуровневого массива.

Отсортированные ассоциативные контейнеры. Обеспечивают возможность быстрой выборки объектов из коллекции на основе значения ключа. Размер коллекции может изменяться во время работы программы. В STL имеется четыре типа отсортированных ассоциативных контейнеров:

- set <Key>. Множество поддерживает уникальные ключи (содержит не более одного значения каждого ключа) и обеспечивает быструю выборку искомого ключа;
- multiset <Key>. Мультимножество поддерживает дублированные ключи (возможно наличие нескольких копий одного и того же значения ключа) и обеспечивает быструю выборку искомого ключа;
- map <Key, T>. Отображение (или словарь) поддерживает уникальные ключи (типа Key) и обеспечивает быструю выборку другого типа T на основе ключа;
- multimap <Key, T>. Мультиотображение (или мультисловарь) поддерживает дублированные ключи (типа Key) и обеспечивает быструю выборку другого типа T на основе ключа.

Подход STL к контейнерам отличается от подходов других библиотек классов контейнеров C++: контейнеры STL не предоставляют множества операций над содержащимися в них объектами данных. Вместо этого STL предоставляет обобщенные алгоритмы.

Обобщенные алгоритмы. Фактически алгоритм find может использоваться для поиска во всех контейнерах STL. Главное, что следует сказать о find и всех других обобщенных алгоритмах STL – это то, что поскольку они могут использоваться со многими или даже со всеми контейнерами, отпадает необходимость в определении соответствующих функций-членов у отдельных контейнеров, что снижает размер кода и упрощает интерфейсы контейнеров.

Аллокаторы. Каждый класс контейнера STL использует класс аллокатора для инкапсуляции информации о модели распределения памяти, используемой программой. Различные модели распределения памяти используют разные способы получения памяти от

операционной системы. Класс аллокатора инкапсулирует информацию об указателях, константных указателях, ссылках, константных ссылках, размерах объектов, типах разности между указателями, функции выделения и освобождения памяти и некоторые другие. Все операции аллокаторов имеют константное амортизированное время работы.

Поскольку модель распределения памяти может быть инкапсулирована в аллокаторе, контейнеры STL могут работать с разными моделями распределения памяти, просто используя разные аллокаторы.

1.1.1 Задания для выполнения лабораторной работы № 1 «Стандартная библиотека (STL)»

Вариант 1

Реализовать использование алгоритма STL reverse со строкой и массивом.

```
#include <iostream>
#include <string>
#include <cassert>
#include <algorithm> // Для алгоритма reverse
using namespace std;

int main()
{cout << "Using reverse algorithm with a string" << endl;
 string string1 = "mark twain";
 reverse(string1.begin(), string1.end());
 assert (string1 == "niawt kram");
 cout << " --- Ok." << endl;

 cout << "Using reverse algorithm with an array" << endl;
 char array1[] = "mark twain";
 int N1 = strlen(array1);
 reverse(&array1[0], &array1[N1]);
 assert (string(array1) == "niawt kram");
 cout << " --- Ok." << endl;
 return 0;}
```

Вариант 2

Реализовать использование алгоритма STL reverse к вектору.

```
#include <iostream>
#include <vector>
#include <cassert>
#include <algorithm> // Для алгоритма reverse
using namespace std;
template <typename Container>
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}
int main()
{
  cout << "Using reverse algorithm with a vector" << endl;
  vector<char> vector1 = make< vector<char> >("mark twain");
  reverse(vector1.begin(), vector1.end());
  assert (vector1 == make< vector<char> >("niawt kram"));
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 3

Реализовать использование алгоритма STL reverse к списку.

```
#include <iostream>
#include <cassert>
#include <list>
#include <algorithm>  // Для алгоритма reverse
using namespace std;
template <typename Container>
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}
```

```
int main()
{
  cout << "Demonstrating generic reverse algorithm on a list"
     << endl;
  list<char> list1 = make< list<char> >("mark twain");
  reverse(list1.begin(), list1.end());
  assert (list1 == make< list<char> >("niawt kram"));
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 4

Реализовать использование контейнера STL map для хранения имен и телефонных номеров.

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

int main()
{
  map<string, long> directory;
  directory["Bogart"] = 1234567;
  directory["Bacall"] = 9876543;
  directory["Cagney"] = 3459876;
  // и т.д.

  // Прочитать имена и просмотреть для них телефонные номера:
  string name;
  while (cin >> name)
    if (directory.find(name) != directory.end())
      cout << "The phone number for " << name
           << " is " << directory[name] << "\n";
    else
      cout << "Sorry, no listing for " << name << "\n";
  return 0;
}
```

Вариант 5

Реализовать использование обобщенного алгоритма STL find с массивом.

```
#include <iostream>
#include <cassert>
#include <algorithm>  // Для алгоритма find
using namespace std;

int main()
{
  cout << "Demonstrating generic find algorithm with "
       << "an array." << endl;
  char s[] = "C++ is a better C";
  int len = strlen(s);

  // Найти первое вхождение символа «e»:
  const char* where = find(&s[0], &s[len], 'e');

  assert (*where == 'e' && *(where+1) == 't');
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 6

Реализовать использование обобщенного алгоритма STL find с вектором.

```
#include <iostream>
#include <cassert>
#include <vector>
#include <algorithm>  // Для алгоритма find
using namespace std;
template <typename Container>
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}
```

```
int main()
{
  cout << "Demonstrating generic find algorithm with "
       << "a vector." << endl;

  vector<char> vector1 =
    make< vector<char> >("C++ is a better C");

  // Найти первое вхождение символа «e»:
  vector<char>::iterator
    where = find(vector1.begin(), vector1.end(), 'e');

  assert (*where == 'e' && *(where + 1) == 't');
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 7

Реализовать использование обобщенного алгоритма STL find со списком.

```
#include <iostream>
#include <cassert>
#include <list>
#include <algorithm> // Для алгоритма find
using namespace std;
template <typename Container>
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}

int main()
{
  cout << "Demonstrating generic find algorithm with "
       << "a list." << endl;
```

```
  list<char> list1 = make< list<char> >("C++ is a better C");

  // Найти первое вхождение символа «e»:
  list<char>::iterator
    where = find(list1.begin(), list1.end(), 'e');

  list<char>::iterator next = where;
  ++next;
  assert (*where == 'e' && *next == 't');
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 8

Реализовать использование обобщенного алгоритма STL find с деком.

```
#include <iostream>
#include <cassert>
#include <deque>
#include <algorithm>  // Для алгоритма find
using namespace std;
template <typename Container>
Container make(const char s[])
{return Container(&s[0], &s[strlen(s)]);}

int main()
{cout << "Demonstrating generic find algorithm with "
      << "a deque." << endl;
  deque<char> deque1 =
    make< deque<char> >("C++ is a better C");

  // Найти первое вхождение символа «e»:
  deque<char>::iterator
    where = find(deque1.begin(), deque1.end(), 'e');
  assert (*where == 'e' && *(where + 1) == 't');
  cout << " --- Ok." << endl;
  return 0;}
```

Вариант 9

Реализовать использование обобщенного алгоритма STL merge с массивом, списком и деком.

```
#include <iostream>
#include <cassert>
#include <list>
#include <deque>
#include <algorithm>  // Для алгоритма merge
using namespace std;
template <typename Container>
Container make(const char s[])
{
 return Container(&s[0], &s[strlen(s)]);
}

int main()
{
 cout << "Demonstrating generic merge algorithm with "
      << "an array, a list, and a deque." << endl;
 char s[] = "aeiou";
 int len = strlen(s);
 list<char> list1 =
   make< list<char> >("bcdfghjklmnpqrstvwxyz");

 // Инициализация deque1 26 копиями символа «x»:
 deque<char> deque1(26, 'x');

 // Слияние массива s и списка list1, результат записывается в deque1:
 merge(&s[0], &s[len], list1.begin(), list1.end(), deque1.begin());
 assert (deque1 == make< deque<char> >("abcdefghijklmnopqrstuvwxyz"));
 cout << " --- Ok." << endl;
 return 0;
}
```

Вариант 10

Реализовать использование обобщенного алгоритма STL merge путем объединения частей массива и дека с помещением результата в список.

```
#include <iostream>
#include <string>
#include <cassert>
#include <list>
#include <deque>
#include <algorithm>  //Для алгоритма merge
using namespace std;
template <typename Container>
Container make(const char s[])
{
 return Container(&s[0], &s[strlen(s)]);
}
int main()
{
 cout << "Demonstrating generic merge algorithm,\n"
      << "merging parts of an array and a deque, putting\n"
      << "the result into a list." << endl;
 char s[] = "acegikm";

 deque<char> deque1 =
  make< deque<char> >("bdfhjlnopqrstuvwxyz");

 // Инициализация списка list1 26 копиями символа «x»:
 list<char> list1(26, 'x');

 // Слияние первых 5 символов из массива s с первыми 10 символами
 // дека deque1, результат записывается в list1:
 merge(&s[0], &s[5], deque1.begin(), deque1.begin() + 10,
      list1.begin());
```

```
  assert (list1 ==
          make< list<char> >("abcdefghijlnopqxxxxxxxxxxx"));
  cout << " --- Ok." << endl;
  return 0;
}
```

1.2 Контейнеры последовательностей

Векторы представляют собой последовательные контейнеры с быстрым произвольным доступом к последовательностям переменной длины, а также с быстрой вставкой и удалением в конце. Этот контейнер следует выбирать, когда требуется максимально быстрый произвольный доступ и выполняется очень малое количество вставок и удалений не в конце последовательности. Если же вставка и удаления необходимы и в начале последовательности, то при использовании вектора они будут занимать линейное время. Если таких операций должно быть много, лучшим выбором является дек, вставки и удаления в обоих концах которого выполняются за константное время при сохранении произвольного доступа к элементам. (Компромисс в данном случае заключается в большем, чем для вектора, времени доступа к элементам). Наконец, если вставки и удаления должны выполняться во внутренних позициях, то лучше использовать список, а не вектор или дек. (Кроме того, списки не поддерживают произвольный доступ, но многие вычисления могут быть выполнены путем последовательного обхода контейнеров, поддерживаемого списками).

Деки. В плане функциональности деки мало отличаются от векторов. Основное отличие заключается в производительности: вставки и удаления в начале дека выполняются гораздо быстрее, чем в векторе, требуя не линейного, а константного времени; в то же время прочие операции имеют то же время работы (или медленнее на константный множитель), что и соответствующие операции векторов. Подобно векторам, деки предоставляют итераторы с произвольным доступом; таким образом, все обобщенные алгоритмы STL могут быть применены к декам. Поэтому деки следует выбирать тогда, когда вставки и удаления требуются с обоих концов последовательности, и их достаточно много для того, чтобы общая производительность приложения с деками оказалась выше, чем с векторами. Хотя деки

поддерживают вставку и удаление в средине последовательности, эти операции выполняются за линейное время. Если в приложении должно выполняться много таких операций, лучшим выбором могут оказаться списки.

Списки. Абстракция последовательности списков существенно отличается от векторов и деков в смысле предоставляемых функций-членов. Причина этого в том, что списки не поддерживают итераторов с произвольным доступом – это цена за константное время вставки и удалений; при отсутствии произвольного доступа некоторые обобщенные алгоритмы, такие как алгоритмы сортировки, не могут быть применены, а потому соответствующие операции предоставляются в качестве функций-членов класса списка.

Кроме того, в виде функций-членов предоставляются и некоторые другие операции, такие как, например, обращение последовательности, которое хотя и может быть выполнено при помощи обобщенного алгоритма reverse и двунаправленных итераторов, которые предоставляет класс list, но при применении специальных алгоритмов, использующих связные структуры списков, может быть выполнено существенно эффективнее.

Функции-члены для вставок и удалений предоставляют по сути тот же интерфейс, что векторы и деки, но с существенной разницей в производительности. Вставки и удаления в произвольной позиции списка (а не только в концах) выполняются за константное время. Связное представление списков обеспечивает, кроме того, выполнение некоторых дополнительных операций, именуемых склейкой, для передачи элементов из одной последовательности в другую за константное время – эти операции также реализованы как функции-члены.

Другим существенным отличием от векторов и деков является то, что вставка в список никогда не делает недействительными никакие итераторы, а удаления – только итераторы, указывающие на удаленный элемент.

К спискам следует прибегать, когда по ходу работы программы требуется большое количество вставок и/или удалений во внутренних позициях и практически не требуется переходить из одной позиции в другую, весьма удаленную.

1.2.1 Задания для выполнения лабораторной работы № 2 «Контейнеры последовательностей»

Вариант 1

Продемонстрируйте работу конструкторов контейнера vector.

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

int main()
{
  cout << "Demonstrating simplest vector constructors"
       << endl;
  vector<char> vector1, vector2(3, 'x');
  assert (vector1.size() == 0);
  assert (vector2.size() == 3);
  assert (vector2[0] == 'x' && vector2[1] == 'x' &&
          vector2[2] == 'x');
  assert (vector2 == vector<char>(3, 'x') &&
          vector2 != vector<char>(4, 'x'));
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 2

Продемонстрируйте работу конструкторов контейнера vector с пользовательским типом.

```
#include <iostream>
#include <cassert>
#include <vector>
using namespace std;
class U {
public:
  unsigned long id;
  U() : id(0) { }
  U(unsigned long x) : id(x) { }
};
```

```
bool operator==(const U& x, const U& y)
{
  return x.id == y.id;
}

bool operator!=(const U& x, const U& y)
{
  return x.id != y.id;
}

int main()
{
  cout << "Demonstrating STL vector constructors with "
       << "a user-defined type." << endl;
  vector<U> vector1, vector2(3);
  assert (vector1.size() == 0);
  assert (vector2.size() == 3);
  assert (vector2[0] == U() && vector2[1] == U() &&
          vector2[2] == U());
  assert (vector2 == vector<U>(3, U()));
  return 0;
}
```

Вариант 3

Продемонстрируйте работу конструкторов контейнера vector с использованием пользовательского типа и явным копированием.

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

class U {
public:
  unsigned long id;
  unsigned long generation;
  static unsigned long total_copies;
  U() : id(0), generation(0) { }
```

```
  U(unsigned long n) : id(n), generation(0) { }
  U(const U& z) : id(z.id), generation(z.generation + 1) {
    ++total_copies;
  }
};

bool operator==(const U& x, const U& y)
{
  return x.id == y.id;
}

bool operator!=(const U& x, const U& y)
{
  return x.id != y.id;
}

unsigned long U::total_copies = 0;

int main()
{cout << "Demonstrating STL vector constructors with "
       << "a user-defined type and showing copying "
       << "explicitly" << endl;
  vector<U> vector1, vector2(3);

  assert (vector1.size() == 0);
  assert (vector2.size() == 3);

  assert (vector2[0] == U() && vector2[1] == U() &&
          vector2[2] == U());

  for (int i = 0; i != 3; ++i)
    cout << "vector2[" << i << "].generation: "
         << vector2[i].generation << endl;

  cout << "Total copies: " << U::total_copies << endl;
  return 0;}
```

Вариант 4

Продемонстрируйте работу копирующих конструкторов контейнера vector.

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

int main()
{
  cout << "Demonstrating STL vector copying constructors"
       << endl;
  char name[] = "George Foreman";
  vector<char> George(name, name + 6);

  vector<char> anotherGeorge(George.begin(), George.end());
  assert (anotherGeorge == George);

  vector<char> son1(George); // Uses copy constructor
  assert (son1 == anotherGeorge);

  vector<char> son2 = George; // Also uses copy constructor
  assert (son2 == anotherGeorge);
  return 0;
}
```

Вариант 5

Продемонстрируйте работу функций push_back и insert контейнера vector.

```
#include <iostream>
#include <cassert>
#include <vector>
#include <string>
#include <algorithm>  // для алгоритма reverse
using namespace std;
template <typename Container>
```

```
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}

int main()
{
  vector<char> vector1 =
                  make< vector<char> >("Bjarne Stroustrup"),
               vector2;
  vector<char>::iterator i;

  cout << "Demonstrating vector push_back function" << endl;
  for (i = vector1.begin(); i != vector1.end(); ++i)
    vector2.push_back(*i);
  assert (vector1 == vector2);

  vector1 = make< vector<char> >("Bjarne Stroustrup");
  vector2 = make< vector<char> >("");

  cout << "Demonstrating vector insertion at beginning"
       << endl;
  for (i = vector1.begin(); i != vector1.end(); ++i)
    vector2.insert(vector2.begin(), *i);
  assert (vector2 ==
          make< vector<char> >("purtsuortS enrajB"));

  // Show that vector2 is the reverse of vector1, by using
  // the generic reverse function to reverse vector1:
  reverse(vector1.begin(), vector1.end());
  assert (vector2 == vector1);
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 6

Продемонстрируйте работу функций capacity и reserve контейнера vector.

```
#include <iostream>
#include <cassert>
#include <vector>
using namespace std;
class U {
public:
  unsigned long id;
  U() : id(0) { }
  U(unsigned long x) : id(x) { }
};

int main()
{
  cout << "Demonstrating STL vector capacity and reserve "
          "functions." << endl;

  int N = 10000; // размер векторов

  vector<U> vector1, vector2;

  cout << "Doing " << N << " insertions in vector1,\n"
       << "with no advance reservation.\n";
  int k;
  for (k = 0; k != N; ++k) {
    vector<U>::size_type cap = vector1.capacity();
    vector1.push_back(U(k));
    if (vector1.capacity() != cap)
      cout << "k: " << k << ", new capacity: "
           << vector1.capacity() << endl;
  }

  vector2.reserve(N);
  cout << "\nNow doing the same thing with vector2,\n"
```

```
        << "after starting with reserve(" << N << ").\n";
    for (k = 0; k != N; ++k) {
      vector<U>::size_type cap = vector2.capacity();
      vector2.push_back(U(k));
      if (vector2.capacity() != cap)
        cout << "k: " << k << ", new capacity: "
            << vector2.capacity() << "\n";
    }
    return 0;
}
```

Вариант 7

Продемонстрируйте работу функций back и pop_back контейнера vector.

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
template <typename Container>
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}

int main()
{cout << "Demonstrating STL vector back "
      << "and pop_back operations." << endl;
  vector<char> vector1 = make< vector<char> >("abcdefghij");

  cout << "Popping characters off the back produces: ";

  while (vector1.size() > 0) {
    cout << vector1.back();
    vector1.pop_back();
  }
  cout << endl;
  return 0;}
```

Вариант 8

Продемонстрируйте работу функции erase контейнера vector.

```
#include <iostream>
#include <cassert>
#include <vector>
#include <string>
#include <algorithm>  // для алгоритма find
using namespace std;
template <typename Container>
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}

int main()
{
  cout << "Demonstrating STL vector erase function." << endl;

  vector<char> vector1 = make< vector<char> >("remembering");
  vector<char>::iterator j;

  j = find(vector1.begin(), vector1.end(), 'm');

  // итератор j сейчас указывает на первый символ 'm':
  assert (*j == 'm' && *(j+1) == 'e');

  vector1.erase(j--);
  assert (vector1 == make< vector<char> >("reembering"));

  // итератор j сейчас указывает на первый символ 'e':
  assert (*j == 'e' && *(j+1) == 'e');

  vector1.erase(j--);
  assert (vector1 == make< vector<char> >("rembering"));
  assert (*j == 'r');
```

```
  // удаляем первые три символа:
  vector1.erase(j, j + 3);
  assert (vector1 == make< vector<char> >("bering"));

  vector1.erase(vector1.begin() + 1);
  assert (vector1 == make< vector<char> >("bring"));
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 9

Продемонстрируйте работу функции push_back и push_front контейнера deque.

```
#include <iostream>
#include <cassert>
#include <string>
#include <deque>
#include <algorithm> // для алгоритма reverse
using namespace std;
template <typename Container>
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}

int main()
{
  deque<char> deque1 =
              make< deque<char> >("Bjarne Stroustrup"),
            deque2;
  deque<char>::iterator i;

  cout << "Demonstrating deque push_back function" << endl;
  for (i = deque1.begin(); i != deque1.end(); ++i)
    deque2.push_back(*i);
  assert (deque1 == deque2);
```

```
  deque1 = make< deque<char> >("Bjarne Stroustrup");
  deque2 = make< deque<char> >("");

  cout << "Demonstrating deque push_front function" << endl;
  for (i = deque1.begin(); i != deque1.end(); ++i)
    deque2.push_front(*i);
  assert (deque2 == make< deque<char> >("purtsuortS enrajB"));

 //Покажем, что deque2 является обращением deque1.
 //Для этого воспользуемся обобщенным алгоритмом STL //reverse
 и выполним с помощью него преобразование deque1:
  reverse(deque1.begin(), deque1.end());
  assert (deque2 == deque1);
  cout << " --- Ok." << endl;
  return 0;
 }
```

Вариант 10

Продемонстрируйте работу функции push_back и push_front контейнера list.

```
 #include <iostream>
 #include <cassert>
 #include <list>
 #include <string>
 #include <algorithm>  // для алгоритма everse
 using namespace std;
 template <typename Container>
 Container make(const char s[])
 {
  return Container(&s[0], &s[strlen(s)]);
 }

 int main()
 {
  list<char> list1 = make< list<char> >("Bjarne Stroustrup"),
            list2;
  list<char>::iterator i;
```

```
  cout << "Demonstrating list push_back function" << endl;
  for (i = list1.begin(); i != list1.end(); ++i)
    list2.push_back(*i);
  assert (list1 == list2);

  list1 = make< list<char> >("Bjarne Stroustrup");
  list2 = make< list<char> >("");

  cout << "Demonstrating list push_front function" << endl;
  for (i = list1.begin(); i != list1.end(); ++i)
    list2.push_front(*i);
  assert (list2 == make< list<char> >("purtsuortS enrajB"));

// Покажем, что list2 является обращением list1.
// Для этого воспользуемся обобщенным алгоритмом STL
// reverse и выполним с помощью него преобразование list1:
  reverse(list1.begin(), list1.end());
  assert (list2 == list1);
  cout << " --- Ok." << endl;
  return 0;
}
```

1.3 Отсортированные ассоциативные контейнеры

В то время как последовательные контейнеры хранят свои данные линейно, с сохранением относительных позиций, в которые были вставлены элементы, отсортированные ассоциативные контейнеры обходятся без этого порядка, а вместо этого сосредотачиваются на максимальной скорости выборки на основе ключей, которые хранятся в элементе (или, в некоторых случаях, представляют собой сам элемент).

Один общий подход к ассоциативной выборке состоит в хранении ключей в отсортированном состоянии в соответствии с некоторым глобальным упорядочением, таким как числовой порядок или лексикографический, если ключи являются строками, и использовании бинарного алгоритма поиска. Еще одним подходом является хеширование: разделение пространства ключей на несколько

подмножеств и вставка каждого ключа в предназначенное ему подмножество; после этого поиск выполняется только в одном подмножестве. Каждый ключ связан со своим подмножеством при помощи так называемой хеш-функции. Первый подход – отсортированные ассоциативные контейнеры – можно реализовать в том числе с использованием сбалансированных бинарных деревьев поиска, а последний – хешированные ассоциативные контейнеры – при помощи любого из множества представлений хеш-таблиц.

В идеале в стандартной библиотеке C++ должны были быть и отсортированные, и хешированные ассоциативные контейнеры, но в нее оказались включены только отсортированные ассоциативные контейнеры.

Отсортированными ассоциативными контейнерами STL являются классы set (множество), multiset (мультимножество), map (отображения или словари) и multimap (мультиотображения или мультисловари). В случае множеств и мультимножеств элементами данных являются сами ключи, причем мультимножество допускает наличие одинаковых ключей, а множество – нет. В отображениях и мультиотображениях элементы данных представляют собой пары, состоящие из ключей и собственно данных некоторого другого типа, причем мультиотображения допускают наличие одинаковых ключей, а отображения – нет.

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

1.3.1 Задания для выполнения лабораторной работы № 3 «Отсортированные ассоциативные контейнеры»

Вариант 1

Продемонстрируйте создание контейнера set и вставки в него элементов.

```
#include <iostream>
#include <cassert>
#include <list>
#include <string>
#include <set>
using namespace std;
template <typename Container>
Container make(const char s[])
{return Container(&s[0], &s[strlen(s)]);}

int main()
{cout << "Demonstrating set construction and insertion."
     << endl;
 list<char> list1 =
    make< list<char> >("There is no distinctly native "
                       "American criminal class");

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

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

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

Вариант 2

Продемонстрируйте создание контейнера multiset и вставки в него элементов.

```
#include <iostream>
#include <cassert>
#include <list>
#include <string>
#include <set>
using namespace std;
template <typename Container>
Container make(const char s[])
{
 return Container(&s[0], &s[strlen(s)]);
}

int main()
{
 cout << "Demonstrating multiset construction "
      << "and insertion." << endl;
 list<char> list1 =
    make< list<char> >("There is no distinctly native "
                       "American criminal class");

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

 // Помесить символы из multiset1 в list2:
 list<char> list2;
 multiset<char>::iterator k;
 for (k = multiset1.begin(); k != multiset1.end(); ++k)
  list2.push_back(*k);
```

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

Вариант 3

Продемонстрируйте работу функции erase контейнера multiset.

```
#include <iostream>
#include <cassert>
#include <list>
#include <string>
#include <set>
using namespace std;
template <typename Container>
Container make(const char s[])
{
 return Container(&s[0], &s[strlen(s)]);
}
#include <functional>
template <typename Container>
string make_string(const Container& c)
{
 string s;
 copy(c.begin(), c.end(), inserter(s, s.end()));
 return s;
}

int main()
{
 cout << "Demonstrating multiset erase functions" << endl;
 list<char> list1 =
    make< list<char> >("There is no distinctly native "
                       "American criminal class");
```

```
// Поместить символы из list1 в multiset1:
  multiset<char> multiset1;
  copy(list1.begin(), list1.end(),
      inserter(multiset1, multiset1.end()));
  assert (make_string(multiset1) ==
     "       ATaaaaccccdeeeehiiiiiiiilllmmnnnnnorrrssssttttvy");
  multiset1.erase('a');
  assert (make_string(multiset1) ==
         "       ATccccdeeeehiiiiiiiilllmmnnnnnorrrssssttttvy");

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

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

Вариант 4

Продемонстрируйте работу функций-членов контейнера multiset для поиска.

```
#include <iostream>
#include <cassert>
#include <list>
#include <string>
#include <set>
using namespace std;
template <typename Container>
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}
#include <functional>
template <typename Container>
string make_string(const Container& c)
{
```

```
  string s;
  copy(c.begin(), c.end(), inserter(s, s.end()));
  return s;
}

int main()
{
  cout << "Demonstrating multiset search member functions."
       << endl;
  list<char> list1 =
     make< list<char> >("There is no distinctly native "
                        "American criminal class"),
          list2 =
     make< list<char> >("except Congress. - Mark Twain");

  // Поместить символы из list1 в multiset1:
  multiset<char> multiset1;
  copy(list1.begin(), list1.end(),
       inserter(multiset1, multiset1.end()));

  assert (make_string(multiset1) ==
     "       ATaaaaccccdeeeehiiiiiiilllmmnnnnnorrrssssttttvy");

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

  multiset1.erase(i, j);

  assert (make_string(multiset1) == "       ATaaaasssstttvy");

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

```
    not_found.push_back(*k);

  assert (found == make< list<char> >("t ss  a Ta"));
  assert (not_found ==
        make< list<char> >("excepCongre.-Mrkwin"));
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 5

Вычислите скалярное произведение кортежей, представленных векторами.

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

int main()
{cout << "Computing an inner product of tuples "
     << "represented as vectors." << endl;
  const long N = 600000; // Длина кортежей x and y
  const long S = 10; // Показатель разреженности
  cout << "\nInitializing..." << flush;
  vector<double> x(N), y(N);
  long k;
  for (k = 0; 3 * k * S < N; ++k)
    x[3 * k * S] = 1.0;
  for (k = 0; 5 * k * S < N; ++k)
    y[5 * k * S] = 1.0;

  cout << "\n\nComputing inner product by brute force: " << flush;
  double sum = 0.0;
  for (k = 0; k < N; ++k)
    sum += x[k] * y[k];

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

Вариант 6

Вычислите скалярное произведение кортежей, представленных отображениями.

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

int main()
{
 cout << "Computing an inner product of tuples "
     << "represented as maps." << endl;

 const long N = 600000; // Длина кортежей x and y
 const long S = 10; // Показатель разреженности

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

 cout << "\n\nComputing inner product taking advantage "
     << "of sparseness: " << flush;

 double sum;
 map<long, double>::iterator ix, iy;
 for (sum = 0.0, ix = x.begin(); ix != x.end(); ++ix) {
  long i = ix->first;
  iy = y.find(i);
  if (iy != y.end())
    sum += ix->second * iy->second;
 }
 cout << sum << endl;
 return 0;}
```

Вариант 7

Продемонстрируйте применение адаптера для указателей на функции для контейнера set.

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

bool less1(const string& x, const string& y)
{
 return x < y;
}

bool greater1(const string& x, const string& y)
{
 return x > y;
}

int main()
{
 cout << "Illustrating the use of an adaptor"
      << " for pointers to functions." << endl;

 typedef
  set<string,
      pointer_to_binary_function<const string&,
                                 const string&, bool> >
  set_type1;

 set_type1 set1(ptr_fun(less1));

 set1.insert("the");
 set1.insert("quick");
 set1.insert("brown");
 set1.insert("fox");
```

```
  set_type1::iterator i;
  for (i = set1.begin(); i != set1.end(); ++i)
    cout << *i << " ";
  cout << endl;

  set_type1 set2(ptr_fun(greater1));

  set2.insert("the");
  set2.insert("quick");
  set2.insert("brown");
  set2.insert("fox");

  for (i = set2.begin(); i != set2.end(); ++i)
    cout << *i << " ";
  cout << endl;
  return 0;
}
```

Вариант 8

Реализовать использование контейнера STL map для хранения имен и телефонных номеров.

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

int main()
{
  map<string, long> directory;
  directory["Bogart"] = 1234567;
  directory["Bacall"] = 9876543;
  directory["Cagney"] = 3459876;
  // и т.д.

  // Прочитать введенное имя и найти для него номер:
  string name;
  while (cin >> name)
```

```
  if (directory.find(name) != directory.end())
    cout << "The phone number for " << name
         << " is " << directory[name] << "\n";
  else
    cout << "Sorry, no listing for " << name << "\n";
 return 0;
}
```

Вариант 9

Выведите все группы анаграмм в порядке уменьшения размера с использованием контейнера map.

```
#include <algorithm>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <list>
#include <map>
#include <iterator>
using namespace std;
#include "ps.h" // Содержит объявления для PS, firstLess, firstEqual

typedef vector<PS>::const_iterator PSi;
typedef pair<PSi, PSi> PPS;

int main() {
 cout << "Anagram group finding program:\n"
      << "finds all anagram groups in a dictionary.\n\n"
      << flush;

 cout << "First, enter the name of the file containing\n"
      << "the dictionary: " << flush;
 string dictionary_name;
 cin >> dictionary_name;
 ifstream ifs(dictionary_name.c_str());
 if (!ifs.is_open()) {
```

```
  cout << "Eh? Could not open file named "
       << dictionary_name << endl;
  exit(1);
}
cout << "\nReading the dictionary ..." << flush;
typedef istream_iterator<string> string_input;

vector<PS> word_pairs;
copy(string_input(ifs), string_input(),
     back_inserter(word_pairs));
cout << "\nSearching " << word_pairs.size()
  << " words for anagram groups..." << flush;
sort(word_pairs.begin(), word_pairs.end(), firstLess);

typedef map<int, list<PPS>, greater<int> > map_1;
map_1 groups;
cout << "\n\nThe anagram groups are" << endl;
PSi j = word_pairs.begin(), finis = word_pairs.end(), k;
while (true) {
  j = adjacent_find(j, finis, firstEqual);
  if (j == finis) break;
  k = find_if(j + 1, finis, not1(bind1st(firstEqual, *j)));
  if (k-j > 1)
    // Save the positions j and k delimiting the anagram
    // group in the list of groups of size k-j:
    groups[k-j].push_back(PPS(j,k));
  j = k;
}
map_1::const_iterator m;
for (m = groups.begin(); m != groups.end(); ++m) {
  cout << "\nAnagram groups of size " << m->first << ":\n";
  list<PPS>::const_iterator l;
  for (l = m->second.begin(); l != m->second.end(); ++l) {
    cout << "   ";
    j = l->first;  // beginning of the anagram group
    k = l->second; // end of the anagram group
```

```
      copy(j, k, ostream_iterator<string>(cout, " "));
      cout << endl;
    }
  }
  return 0;
}
```

Ниже приведено описание подключаемого файла «ps.h»:

```
//Определяется структура PS и некоторые связанные с ней
//функции для поддержки строковых пар
#include <functional>
using namespace std;
struct PS : pair<string, string> {
  PS() : pair<string, string>(string(), string()) { }
  PS(const string& s) : pair<string, string>(s, s) {
    sort(first.begin(), first.end());
  }
  operator string() const { return second; }
};
struct FirstLess : binary_function<PS, PS, bool> {
  bool operator()(const PS& p, const PS& q) const
  {
    return p.first < q.first;
  }
} firstLess;
struct FirstEqual : binary_function<PS, PS, bool> {
  bool operator()(const PS& p, const PS& q) const
  {
    return p.first == q.first;
  }
} firstEqual;
```

Вариант 10

Поиск всех групп анаграмм в заданном словаре с использованием контейнера multimap.

2 Итераторы

Итераторы представляют собой указателеподобные объекты, которые алгоритмы STL используют для обхода последовательности объектов, хранящихся в контейнере. Итераторы занимают центральное место в дизайне STL благодаря их роли посредников между контейнерами и обобщенными алгоритмами. Они позволяют создавать обобщенные алгоритмы без учета того, как именно хранятся последовательности данных, а контейнеры – без необходимости написания большого количества исходного текста работающих с ними алгоритмов. Однако по причинам эффективности нельзя обеспечить возможность работы каждого обобщенного алгоритма с каждым контейнером.

Одна из ключевых технических идей, лежащих в основе STL – разделение итераторов на пять категорий: входные, выходные, однонаправленные, двунаправленные и произвольного доступа.

Все алгоритмы STL получают доступ к последовательностям через итераторы, обычно посредством пары итераторов first и last, которые указывают начало и конец последовательности. Для того чтобы разобраться, как такая пара итераторов определяет последовательность, используется концепция диапазона итераторов. Диапазон от first до last состоит из итераторов, которые получаются, начиная с итератора first, путем применения оператора operator++ до тех пор, пока не будет достигнут итератор last, но не включая его. Этот диапазон записывается в виде [first; last) и называется корректным «тогда и только тогда», когда итератор last достижим из итератора first. Все алгоритмы STL полагают, что все диапазоны, с которыми они работают, корректны; результат применения алгоритма к некорректному диапазону не определен.

Важным частным случаем является пустой диапазон, когда first==last. Пустой диапазон корректен, но не содержит итераторов, указывающих на корректные данные.

Входные итераторы. Для входных итераторов должны быть определены следующие выражения:

- выражение first!=last должно возвращать true, если first не равно last, в противном случае оно должно возвращать false;

- ++first должно выполнять инкремент first и возвращать новое значение итератора;

- *first должно возвращать значение, на которое указывает first.

Дополнительные требования, налагаемые на входные итераторы: чтобы был определен оператор ==, выполняющий проверку на равенство, и чтобы был также определен постфиксный оператор ++ с тем же действием, что и префиксный оператор ++, но возвращающий значение итератора до увеличения (как и в случае встроенных типов указателей). От входных итераторов не требуется поддержка записи в указанную позицию при помощи выражения
*first=....

Выходные итераторы обладают противоположной функциональностью: они позволяют записывать значения в последовательность, но не гарантируют возможность их чтения. То есть, если first – выходной итератор, то мы можем написать *first=..., но не гарантируется возможность применения *first в выражении для получения значения, на которое указывает итератор. Еще одно отличие от требований для входных итераторов состоит в отсутствии обязательной поддержки операторов == и != для выходных итераторов. Что касается префиксного и постфиксного операторов ++, то здесь выходные итераторы идентичны входным.

Однонаправленные итераторы объединяют в себе и входной, и выходной итераторы, тем самым обеспечивая возможность чтения и записи и обхода последовательности в одном направлении. Однонаправленные итераторы обладают также свойством, которое не требуется ни для входных, ни для выходных итераторов: возможностью сохранить однонаправленный итератор и использовать сохраненное значение для повторного прохода из того же положения. Это свойство позволяет однонаправленным итераторам использоваться в многопроходных алгоритмах, в отличие от однопроходных алгоритмов, таких как find и merge.

Двунаправленные итераторы аналогичны однонаправленному итератору, за исключением того, что они допускают обход в обоих направлениях. То есть, двунаправленные итераторы должны поддерживать все операции однонаправленных итераторов, а кроме

них – операцию --, делая возможным обход последовательности в обратном направлении.

Требуется наличие и префиксной, и постфиксной версий operator--; префиксная версия выполняет декремент итератора и возвращает новое его значение, а постфиксная – выполняет декремент итератора и возвращает его старое значение. И префиксный, и постфиксный operator-- должны выполняться за константное время.

Возможность обхода структуры данных в обратном порядке важна, потому что в противном случае некоторые алгоритмы не в состоянии эффективно работать. Например, алгоритм STL reverse может использоваться для обращения порядка элементов в последовательности при наличии двунаправленных итераторов.

Итераторы с произвольным доступом. Имеются некоторые алгоритмы, которые предъявляют к итераторам еще более высокие требования. Для эффективной работы эти алгоритмы требуют, чтобы любой элемент последовательности был достижим из любого другого за константное время. Итераторы с произвольным доступом должны поддерживать все операции двунаправленных итераторов, а также следующие (здесь r и s – итераторы с произвольным доступом, а n – целочисленное значение):

- прибавление и вычитание целых чисел, выражаемые r+s, n+r и r-n;
- доступ к n-му элементу при помощи выражения r[n], которое означает *(r+n);
- двунаправленные «большие обходы», выражаемые как r+=n и r-=n;
- вычитание итераторов, выражаемое как r-s и дающее целочисленное значение;
- сравнения, выражаемые как r<s, r>s, r<=s и r>=s и дающие значения типа bool.

2.1 Задания для выполнения лабораторной работы № 4 «Итераторы»

Вариант 1

Реализовать использование обобщенного алгоритма STL find с входными итераторами массивов, списков и входных потоков.

```
#include <iostream>
#include <cassert>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

int main()
{// Инициализация массива 10 целочисленными значениями:
 int a[10] = {12, 3, 25, 7, 11, 213, 7, 123, 29, -31};
 // Поиск первого элемента в массиве равного 7:
 int* ptr = find(&a[0], &a[10], 7);
 assert (*ptr == 7 && *(ptr+1) == 11);
 // Инициализация list1 значениями из массива a:
 list<int> list1(&a[0], &a[10]);

 // Поиск первого элемента в list1 равного 7:
 list<int>::iterator i = find(list1.begin(),
                              list1.end(),7);

 assert (*i == 7 && *(++i) == 11);

 cout << "Type some characters, including an 'x' followed\n"
   << "by at least one nonwhite-space character: " << flush;

 istream_iterator<char> in(cin);
 istream_iterator<char> eos;
 find(in, eos, 'x');
 cout << "The first nonwhite-space character following\n"
     << "the first 'x' was '" << *(++in) << "'." << endl;
 return 0;}
```

Вариант 2

Реализовать программу поиска всех анаграмм в слове, используя словарь, загруженный из файла

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
 cout << "Anagram finding program:\n"
  << "finds all words in a dictionary that can\n"
  << "be formed with the letters of a given word.\n" << endl;
 cout << "First, enter the name of the file containing\n"
     << "the dictionary: " << flush;
 string dictionary_name;
 cin >> dictionary_name;
 ifstream ifs(dictionary_name.c_str());
 if (!ifs.is_open()) {
  cout << "Eh? Could not open file named "
       << dictionary_name << endl;
  exit(1);
 }
 cout << "\nReading the dictionary ..." << flush;
 typedef istream_iterator<string> string_input;
 vector<string> dictionary;
 copy(string_input(ifs), string_input(),
     back_inserter(dictionary));
 cout << "\nThe dictionary contains "
  << dictionary.size() << " words.\n\n";
 cout << "Now type a word (or any string of letters),\n"
  << "and I'll see if it has any anagrams: " << flush;
 for (string_input j(cin); j != string_input(); ++j) {
```

```
    string word = *j;
    sort(word.begin(), word.end());
    bool found_one = false;
    do {
      if (binary_search(dictionary.begin(),
                        dictionary.end(),
                        word)) {
        cout << "  " << word << endl;
        found_one = true;
      }
    } while (next_permutation(word.begin(), word.end()));
    if (!found_one)
      cout << "  Sorry, none found.\n";
    cout << "\nType another word "
         << "(or the end-of-file char to stop): " << flush;
  }
  return 0;
}
```

Вариант 3

Найдите все группы анаграмм в словаре и напечатайте их, используя стандартный потоковый вывод

```
#include <algorithm>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <iterator>
using namespace std;
#include "ps.h" // Содержит объявления для PS, firstLess, firstEqual


int main() {
  cout << "Anagram group finding program:\n"
       << "finds all anagram groups in a dictionary.\n\n"
```

```
      << flush;

  cout << "First, enter the name of the file containing\n"
       << "the dictionary: " << flush;
  string dictionary_name;
  cin >> dictionary_name;
  ifstream ifs(dictionary_name.c_str());
  if (!ifs.is_open()) {
    cout << "Eh? Could not open file named "
         << dictionary_name << endl;
    exit(1);
  }
  cout << "\nReading the dictionary ..." << flush;
  typedef istream_iterator<string> string_input;

  vector<PS> word_pairs;
  copy(string_input(ifs), string_input(),
       back_inserter(word_pairs));
  cout << "\nSearching " << word_pairs.size()
    << " words for anagram groups..." << flush;
  sort(word_pairs.begin(), word_pairs.end(), firstLess);
  vector<PS>::const_iterator j = word_pairs.begin(),
                             finis = word_pairs.end(), k;
  cout << "\n\nThe anagram groups are:" << endl;
  while (true) {
    j = adjacent_find(j, finis, firstEqual);
    if (j == finis) break;
    k = find_if(j + 1, finis, not1(bind1st(firstEqual, *j)));
    cout << "  ";
    copy(j, k, ostream_iterator<string>(cout, " "));
    cout << endl;
    j = k;
  }
  return 0;
}
```

Вариант 4

Найдите все группы анаграмм в порядке увеличения размера

```
#include <algorithm>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <list>
#include <map>
#include <iterator>
using namespace std;
#include "ps.h" // Содержит объявления для PS, firstLess, firstEqual


typedef vector<PS>::const_iterator PSi;
typedef pair<PSi, PSi> PPS;

int main() {
  cout << "Anagram group finding program:\n"
       << "finds all anagram groups in a dictionary.\n\n"
       << flush;

  cout << "First, enter the name of the file containing\n"
       << "the dictionary: " << flush;
  string dictionary_name;
  cin >> dictionary_name;
  ifstream ifs(dictionary_name.c_str());
  if (!ifs.is_open()) {
    cout << "Eh? Could not open file named "
         << dictionary_name << endl;
    exit(1);
  }
  cout << "\nReading the dictionary ..." << flush;
  typedef istream_iterator<string> string_input;
```

```
  vector<PS> word_pairs;
  copy(string_input(ifs), string_input(),
       back_inserter(word_pairs));
  cout << "\nSearching " << word_pairs.size()
    << " words for anagram groups..." << flush;
  sort(word_pairs.begin(), word_pairs.end(), firstLess);

  typedef map<int, list<PPS>, greater<int> > map_1;
  map_1 groups;
  cout << "\n\nThe anagram groups are" << endl;
  PSi j = word_pairs.begin(), finis = word_pairs.end(), k;
  while (true) {
    j = adjacent_find(j, finis, firstEqual);
    if (j == finis) break;
    k = find_if(j + 1, finis, not1(bind1st(firstEqual, *j)));
    if (k-j > 1)
      // Save the positions j and k delimiting the anagram
      // group in the list of groups of size k-j:
      groups[k-j].push_back(PPS(j,k));
    j = k;
  }
  map_1::const_iterator m;
  for (m = groups.begin(); m != groups.end(); ++m) {
    cout << "\nAnagram groups of size " << m->first << ":\n";
    list<PPS>::const_iterator l;
    for (l = m->second.begin(); l != m->second.end(); ++l) {
      cout << "   ";
      j = l->first;  // beginning of the anagram group
      k = l->second; // end of the anagram group
      copy(j, k, ostream_iterator<string>(cout, " "));
      cout << endl;
    }
  }
  return 0;
}
```

Вариант 5

Найдите все группы анаграмм из слов словаря.

```
#include <algorithm>
#include <iostream>
#include <fstream>
#include <string>
#include <list>
#include <map>
#include <iterator>
#include <functional>
using namespace std;

typedef multimap<string, string> multimap_1;
typedef multimap_1::value_type PS;
typedef multimap_1::const_iterator PSi;

typedef pair<PSi, PSi> PPS;

int main() {
  cout << "Anagram group finding program:\n"
    << "finds all anagram groups in a dictionary.\n\n";

  cout << "First, enter the name of the file containing\n"
      << "the dictionary: " << flush;
  string dictionary_name;
  cin >> dictionary_name;
  ifstream ifs(dictionary_name.c_str());
  if (!ifs.is_open()) {
    cout << "Eh? Could not open file named "
        << dictionary_name << endl;
    exit(1);
  }

  cout << "\nReading the dictionary ..." << flush;

  // Copy words from dictionary file to
```

```
// a multimap:
typedef istream_iterator<string> string_input;
multimap_1 word_pairs;
for (string_input in(ifs); in != string_input(); ++in) {
  string word = *in;
  sort(word.begin(), word.end());
  word_pairs.insert(PS(word, *in));
}

cout << "\nSearching " << word_pairs.size()
  << " words for anagram groups..." << flush;

// Set up the map from anagram group sizes to lists of
// groups of that size:
typedef map<int, list<PPS>, greater<int> > map_1;
map_1 groups;

// Find all the anagram groups and save their
// positions in the groups map:
cout << "\n\nThe anagram groups are: " << endl;
PSi j = word_pairs.begin(), finis = word_pairs.end(), k;
while (true) {
  // Make j point to the next anagram
  // group, or to the end of the multimap:
  j = adjacent_find(j, finis,
                    not2(word_pairs.value_comp()));
  if (j == finis) break;

  k = find_if(j, finis,
              bind1st(word_pairs.value_comp(), *j));
  multimap_1::size_type n = distance(j, k);
  if (n > 1)
    // Save the positions j and k delimiting the anagram
    // group in the list of groups of size n:
    groups[n].push_back(PPS(j, k));
```

```
    // Prepare to continue search at position k:
    j = k;
  }

  // Iterate through the groups map to output the anagram
  // groups in order of decreasing size:
  map_1::const_iterator m;
  for (m = groups.begin(); m != groups.end(); ++m) {

    cout << "\nAnagram groups of size "
      << m->first << ":\n";

    list<PPS>::const_iterator l;
    for (l = m->second.begin();
        l != m->second.end(); ++l) {
      cout << "   ";
      j = l->first;  // Beginning of the anagram group
      k = l->second; // End of the anagram group
      for (; j != k; ++j)
        cout << j->second << " ";
      cout << endl;
    }
  }
  return 0;
}
```

Вариант 6

Реализовать использование обобщенного алгоритма STL find при прямом и обратном проходе последовательности.

```
#include <iostream>
#include <vector>
#include <algorithm> // для алгоритма find
#include <iterator>
using namespace std;
template <typename Container>
Container make(const char s[])
{
```

```
  return Container(&s[0], &s[strlen(s)]);
}

int main()
{
  cout << "Using find with normal and reverse iteration:\n";
  vector<char> vector1 =
     make< vector<char> >("now is the time");
  ostream_iterator<char> out(cout, " ");

  vector<char>::iterator i =
    find(vector1.begin(), vector1.end(), 't');
  cout << "chars from the first t to the end: ";
  copy(i, vector1.end(), out); cout << endl;

  cout << "chars from the last t to the beginning: ";
  vector<char>::reverse_iterator r =
    find(vector1.rbegin(), vector1.rend(), 't');
  copy(r, vector1.rend(), out); cout << endl;

  cout << "chars from the last t to the end: ";
  copy(r.base() - 1, vector1.end(), out); cout << endl;
  return 0;
}
```

Вариант 7

Продемонстрировать возможности прямого и обратного прохода последовательности.

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

template <typename Container>
void display(const Container& c)
{
```

```
  cout << "Elements in normal (forward) order:  ";
  typename Container::const_iterator i;
  for (i = c.begin(); i != c.end(); ++i)
     cout << *i << "  ";
  cout << endl;

  cout << "Elements in reverse order:           ";
  typename Container::const_reverse_iterator r;
  for (r = c.rbegin(); r != c.rend(); ++r)
     cout << *r << "  ";
  cout << endl;
}

int main()
{
  cout << "Normal and reverse iteration through a vector:\n";
  vector<int> vector1;
  vector1.push_back(2);
  vector1.push_back(3);
  vector1.push_back(5);
  vector1.push_back(7);
  vector1.push_back(11);

  display(vector1);

  cout << "Normal and reverse iteration through a list:\n";
  list<int> list1(vector1.begin(), vector1.end());

  display(list1);
  return 0;
}
```

Вариант 8

Реализовать использование обобщенного алгоритма STL accumulate с обратным итератором.

```
#include <iostream>
#include <vector>
#include <cassert>
#include <numeric>  // для алгоритма accumulate
using namespace std;

int main()
{cout << "Demonstrating generic accumulate algorithm with "
     << "a reverse iterator." << endl;
 float small = (float)1.0/(1 << 26);
 float x[5] = {1.0, 3*small, 2*small, small, small};
 // Инициализация vector1 элементами массива с x[0] по x[4]:
 vector<float> vector1(&x[0], &x[5]);

 cout << "Values to be added: " << endl;

 vector<float>::iterator i;
 cout.precision(10);
 for (i = vector1.begin(); i != vector1.end(); ++i)
  cout << *i << endl;
 cout << endl;

 float sum = accumulate(vector1.begin(), vector1.end(),
                        (float)0.0);

 cout << "Sum accumulated from left = " <<  sum << endl;

 float sum1 = accumulate(vector1.rbegin(), vector1.rend(),
                         (float)0.0);

 cout << "Sum accumulated from right = "
     << (double)sum1 << endl;
 return 0;}
```

Вариант 9

Реализовать использование контейнера STL vector и итераторов с произвольным доступом

```
#include <iostream>
#include <cassert>
#include <vector>
#include <algorithm>  // для алгоритма find
using namespace std;
template <typename Container>
Container make(const char s[])
{
 return Container(&s[0], &s[strlen(s)]);
}

int main()
{
 cout << "Demonstrating generic find algorithm with "
      << "a vector." << endl;

 vector<char> vector1 =
  make< vector<char> >("C++ is a better C");

 // Поиск первого вхождения символа e:
 vector<char>::iterator
  where = find(vector1.begin(), vector1.end(), 'e');

 assert (*where == 'e' && *(where + 1) == 't');
 cout << " --- Ok." << endl;
 return 0;
}
```

Вариант 10

Реализовать использование контейнера STL multiset с применением итераторов.

```
#include <iostream>
#include <cassert>
#include <list>
#include <string>
#include <set>
using namespace std;
template <typename Container>
Container make(const char s[])
{
 return Container(&s[0], &s[strlen(s)]);
}
#include <functional>
template <typename Container>
string make_string(const Container& c)
{
 string s;
 copy(c.begin(), c.end(), inserter(s, s.end()));
 return s;
}

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

 // Положить символы из list1 в multiset1:
 multiset<char> multiset1;
 copy(list1.begin(), list1.end(),
     inserter(multiset1, multiset1.end()));
 assert (make_string(multiset1) ==
    "       ATaaaaccccdeeeehiiiiiiilllmmnnnnnorrrsssstttvy");
```

```
  multiset1.erase('a');
  assert (make_string(multiset1) ==
        "       ATccccdeeeehiiiiiiilllmmnnnnnnorrrsssstttvy");

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

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

3 Обобщенные алгоритмы

STL предоставляет программистам богатый набор алгоритмов, работающих со структурами данных, определенными в рамках схемы STL. Алгоритмы STL являются обобщенными: каждый алгоритм может работать не с одной, а с разнообразными структурами данных.

Обобщенные алгоритмы STL разделяются на четыре большие категории в соответствии с их семантикой. **Неизменяющие алгоритмы над последовательностями** работают с контейнерами без модификации их содержимого, в то время как **изменяющие алгоритмы над последовательностями** обычно модифицируют содержимое контейнеров, с которыми они имеют дело. **Связанные с сортировкой алгоритмы** включают алгоритмы сортировки и слияния, алгоритмы бинарного поиска и операции над множествами, работающие с упорядоченными последовательностями. Наконец, имеется небольшой набор **обобщенных числовых алгоритмов**.

3.1 Задания для выполнения лабораторной работы № 5 «Обобщенные алгоритмы»

Вариант 1

Продемонстрируйте использование алгоритма сортировки «на месте».

```
#include <iostream>
#include <algorithm>
```

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

int main() {
  cout << "Using an in-place generic sort algorithm." << endl;
  int a[1000];
  int i;
  for (i = 0; i < 1000; ++i)
    a[i] = 1000 - i - 1;

  sort(&a[0], &a[1000]);

  for (i = 0; i < 1000; ++i)
    assert (a[i] == i);
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 2

Продемонстрируйте использование алгоритма reverse_copy, копирующей версии обобщенного алгоритма reverse.

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

int main() {
  cout << "Using reverse_copy, a copying version of the"
       << " generic reverse algorithm." << endl;
  int a[1000], b[1000];
  int i;
  for (i = 0; i < 1000; ++i)
    a[i] = i;

  reverse_copy(&a[0], &a[1000], &b[0]);

  for (i = 0; i < 1000; ++i)
```

```
    assert (a[i] == i && b[i] == 1000 - i - 1);
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 3

Продемонстрируйте использование обобщенного алгоритма sort с бинарным предикатом.

```
#include <iostream>
#include <algorithm>
#include <cassert>
#include <functional>
using namespace std;

int main() {
  cout << "Using the generic sort algorithm "
       << "with a binary predicate." << endl;
  int a[1000];
  int i;
  for (i = 0; i < 1000; ++i)
    a[i] = i;
  random_shuffle(&a[0], &a[1000]);
  // Сортировка по возрастанию:
  sort(&a[0], &a[1000]);

  for (i = 0; i < 1000; ++i)
    assert (a[i] == i);

  random_shuffle(&a[0], &a[1000]);

  // Сортировка по убыванию:
  sort(&a[0], &a[1000], greater<int>());

  for (i = 0; i < 1000; ++i)
    assert (a[i] == 1000 - i - 1);
  cout << " --- Ok." << endl;
  return 0;}
```

Вариант 4

Продемонстрируйте использование обобщенного алгоритма find_if.

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

// Определение унарного предиката в виде объекта:
class GreaterThan50 {
 public:
  bool operator()(int x) const { return x > 50; }
};

int main()
{
  cout << "Illustrating the generic find_if algorithm."
       << endl;
  // Создание вектора со значениями 0, 1, 4, 9, 16, ..., 144:
  vector<int> vector1;
  for (int i = 0; i < 13; ++i)
    vector1.push_back(i * i);

  vector<int>::iterator where;
  where = find_if(vector1.begin(), vector1.end(),
                  GreaterThan50());

  assert (*where == 64);
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 5

Продемонстрируйте использование обобщенного алгоритма adjacent_find.

```
#include <iostream>
#include <string>
#include <algorithm>
#include <cassert>
#include <functional>
#include <deque>
using namespace std;

int main()
{cout << "Illustrating the generic adjacent_find algorithm."  << endl;
 deque<string> player(5);
 deque<string>::iterator i;
 // Инициализация дека:
 player[0] = "Pele";
 player[1] = "Platini";
 player[2] = "Maradona";
 player[3] = "Maradona";
 player[4] = "Rossi";

 // Нахождение первой последовательной пары одинаковых имен:
 i = adjacent_find(player.begin(), player.end());

 assert (*i == "Maradona" && *(i+1) == "Maradona");

 // Нахождение первого имени, которое лексикографически
 // больше чем следующее имя:
 i = adjacent_find(player.begin(), player.end(),
                   greater<string>());

 assert (*i == "Platini" && *(i+1) == "Maradona");
 cout << " --- Ok." << endl;
 return 0;}
```

Вариант 6

Продемонстрируйте использование обобщенного алгоритма count.

```
#include <iostream>
#include <cassert>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{
  cout << "Illustrating the generic count algorithm." << endl;
  int a[] = {0, 0, 0, 1, 1, 1, 2, 2, 2};

  // Подсчет количества чисел в массиве a
  // больших или равных 1:
  int final_count = count(&a[0], &a[9], 1);

  assert (final_count == 3);

  // Определение чисел в массиве,
  // которые не равны 1:
  final_count = count_if(&a[0], &a[9],
                    bind2nd(not_equal_to<int>(), 1));

  // Проверка того, что 6 элементов не равны 1.
  assert (final_count == 6);
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 7

Продемонстрируйте использование обобщенного алгоритма for_each.

```
#include <iostream>
#include <cassert>
#include <algorithm>
#include <string>
#include <list>
#include <iostream>
using namespace std;

void print_list(string s)
{
  cout << s << endl;
}

int main()
{
  cout << "Illustrating the generic for_each algorithm."
       << endl;
  list<string> dlist;
  dlist.insert(dlist.end(), "Clark");
  dlist.insert(dlist.end(), "Rindt");
  dlist.insert(dlist.end(), "Senna");

  // Печать (вывод на экран) каждого элемента списка:
  for_each(dlist.begin(), dlist.end(), print_list);
  return 0;
}
```

Вариант 8

Продемонстрируйте использование обобщенного алгоритма equal.

```
#include <iostream>
#include <cassert>
#include <algorithm>
#include <string>
```

```
#include <list>
#include <deque>
#include <vector>
using namespace std;

int main()
{
  cout << "Illustrating the generic equal "
       << "and mismatch algorithms." << endl;
  list<string> driver_list;
  vector<string> vec;
  deque<string> deq;

  driver_list.insert(driver_list.end(), "Clark");
  driver_list.insert(driver_list.end(), "Rindt");
  driver_list.insert(driver_list.end(), "Senna");

  vec.insert(vec.end(), "Clark");
  vec.insert(vec.end(), "Rindt");
  vec.insert(vec.end(), "Senna");
  vec.insert(vec.end(), "Berger");

  deq.insert(deq.end(), "Clark");
  deq.insert(deq.end(), "Berger");

  // Показывает, что driver_list и первые 3 элемента
  // vec равны в соответствующих позициях:
  assert (equal(driver_list.begin(), driver_list.end(),
                vec.begin()));

  // Показывает, что deq и первые 2 элемента driver_list
  // не равны в соответствующих позициях:
  assert (!equal(deq.begin(), deq.end(),
                 driver_list.begin()));

  // Поиск соответствующих позиций в deq и driver_list,
```

```
  // в которых неодинаковые элементы впервые появляются:
  pair<deque<string>::iterator, list<string>::iterator>
    pair1 = mismatch(deq.begin(), deq.end(),
                     driver_list.begin());

  if (pair1.first != deq.end())
    cout << "First disagreement in deq and driver_list:\n  "
         << *(pair1.first) << " and " << *(pair1.second)
         << endl;
  return 0;
}
```

Вариант 9

Продемонстрируйте использование обобщенного алгоритма search.

```
#include <iostream>
#include <cassert>
#include <algorithm>
#include <vector>
#include <deque>
using namespace std;

int main()
{
  cout << "Illustrating the generic search algorithm." << endl;
  vector<int> vector1(20);
  deque<int>  deque1(5);
  // Инициализация vector1 значениями 0, 1, ..., 19:
  int i;
  for (i = 0; i < 20; ++i)
    vector1[i] = i;

  // Инициализация deque1 значениями 5, 6, 7, 8, 9:
  for (i = 0; i < 5; ++i)
    deque1[i] = i + 5;

  // Поиск первого вхождения в deque1
```

```
  // последовательности из vector1:
  vector<int>::iterator k =
    search(vector1.begin(), vector1.end(),
           deque1.begin(), deque1.end());

  // Проверка, что значения 5, 6, 7, 8, 9 входят в vector1,
  // начиная с позиции k:
  for (i = 0; i < 5; ++i)
    assert (*(k + i) == i + 5);
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 10

Продемонстрируйте использование обобщенных алгоритмов copy copy_backward.

```
#include <iostream>
#include <cassert>
#include <algorithm>
#include <vector>
#include <string>
#include <iostream>
using namespace std;

int main()
{
  cout << "Illustrating the generic copy "
       << "and copy_backward algorithms." << endl;
  string s("abcdefghihklmnopqrstuvwxyz");
  vector<char> vector1(s.begin(), s.end());

  vector<char> vector2(vector1.size());

  // Копирование vector1 в vector2:
  copy(vector1.begin(), vector1.end(),
       vector2.begin());
```

```
  assert (vector1 == vector2);

  // Сдвиг содержимого vector1 влево на 4 позиции:
  copy(vector1.begin() + 4, vector1.end(),
      vector1.begin());

  assert (string(vector1.begin(), vector1.end()) ==
        string("efghihklmnopqrstuvwxyzwxyz"));

  // Сдвиг вправо на 2 позиции:
  copy_backward(vector1.begin(), vector1.end() - 2,
              vector1.end());

  assert (string(vector1.begin(), vector1.end()) ==
        string("efefghihklmnopqrstuvwxyzwx"));
  cout << " --- Ok." << endl;
  return 0;
}
```

4 Расширение STL

4.1 Функциональные объекты

Большинство обобщенных алгоритмов STL (и некоторые классы контейнеров) принимает в качестве параметра функциональный объект, что делает возможным влиять на вычисления способом, отличным от управления итераторами. Функциональный объект представляет собой любую сущность, которая может быть применена к нулю или большему количеству элементов для получения значения и/или изменения состояния вычислений. В программировании на C++ любая обычная функция отвечает этому определению, но ему же отвечает и объект любого класса (или структуры), который перегружает оператор вызова функции operator().

STL предоставляет с десяток или около того функциональных объектов для наиболее распространенных случаев.

4.1.1 Задания для выполнения лабораторной работы № 6 «Функциональные объекты»

Вариант 1

Продемонстрируйте использование обобщенного алгоритма accumulate для вычисления произведения с применением multiplies.

```
#include <iostream>
#include <vector>
#include <cassert>
#include <numeric>  // Для алгоритма accumulate
#include <functional>  // Для функционального объекта multiplies
using namespace std;

int main()
{
  cout << "Using generic accumulate algorithm to "
       << "compute a product." << endl;

  int x[5] = {2, 3, 5, 7, 11};

  // Инициализация vector1 с x[0] по x[4]:
  vector<int> vector1(&x[0], &x[5]);

  int product = accumulate(vector1.begin(), vector1.end(),
                           1, multiplies<int>());

  assert (product == 2310);
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 2

Продемонстрируйте использование функционального объекта для подсчета операций, первая версия

```
#include <iostream>
#include <iomanip>
#include <cassert>
#include <vector>
```

```
#include <algorithm>
#include <functional>
using namespace std;
template <typename T>
class less_with_count : public binary_function<T, T, bool> {
public:
  less_with_count() { }
  bool operator()(const T& x, const T& y) {
        ++counter;
        return x < y;
  }
  long report() const {return counter;}
  static long counter;
};

template <typename T>
long less_with_count<T>::counter = 0;

int main()
{
  cout << "Using a function object for operation counting, "
       << "first version." << endl;
  const long N1 = 1000, N2 = 128000;
  for (long N = N1; N <= N2; N *= 2) {
    vector<int> vector1;
    for (int k = 0; k < N; ++k)
      vector1.push_back(k);
    random_shuffle(vector1.begin(), vector1.end());
    less_with_count<int> comp_counter;
    less_with_count<int>::counter = 0;
    sort(vector1.begin(), vector1.end(), comp_counter);
    cout << "Problem size " << setw(9) << N
         << ",  comparisons performed: "
         << setw(9) << comp_counter.report() << endl;
  }
  return 0;}
```

Вариант 3

Продемонстрируйте использование функционального объекта для подсчета операций, вторая версия

```
#include <iostream>
#include <iomanip>
#include <cassert>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
template <typename T>
class less_with_count : public binary_function<T, T, bool> {
public:
  less_with_count() : counter(0), progenitor(0) { }
  // Конструктор копирования:
  less_with_count(less_with_count<T>& x) : counter(0),
     progenitor(x.progenitor ? x.progenitor : &x) { }
  bool operator()(const T& x, const T& y) {
    ++counter;
    return x < y;
  }
  long report() const { return counter; }
  ~less_with_count() {  // Destructor
    if (progenitor) {
      progenitor->counter += counter;
    }
  }
private:
  long counter;
  less_with_count<T>* progenitor;
};
int main()
{
  cout << "Using a function object for operation counting, "
       << "second version." << endl;
  const long N1 = 1000, N2 = 128000;
```

```
  for (long N = N1; N <= N2; N *= 2) {
    vector<int> vector1;
    for (int k = 0; k < N; ++k)
      vector1.push_back(k);
    random_shuffle(vector1.begin(), vector1.end());
    less_with_count<int> comp_counter;
    sort(vector1.begin(), vector1.end(), comp_counter);
    cout << "Problem size " << setw(9) << N
         << ",  comparisons performed: "
         << setw(9) << comp_counter.report() << endl;
  }
  return 0;
}
```

Вариант 4

Продемонстрируйте использование обобщенного алгоритма accumulate для вычисления произведения с применением функционального объекта.

```
#include <iostream>
#include <vector>
#include <cassert>
#include <numeric>  // Для алгоритма accumulate
using namespace std;

class multiply {
 public:
  int operator()(int x, int y) const { return x * y; }
};

int main()
{
  cout << “Using generic accumulate algorithm to “
       << “compute a product.” << endl;

  int x[5] = {2, 3, 5, 7, 11};

  // Инициализация vector1 с x[0] по x[4]:
```

```
  vector<int> vector1(&x[0], &x[5]);

  int product = accumulate(vector1.begin(), vector1.end(),
                           1, multiply());

  assert (product == 2310);
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 5

Сортировка вектора в возрастающем порядке членов id.

```
#include <iostream>
#include <cassert>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

class U : public binary_function<U, U, bool> {
public:
  int id;
  bool operator()(const U& x, const U& y) const {
     return x.id >= y.id;
  }
  friend ostream& operator<<(ostream& o, const U& x) {
    o << x.id;
    return o;
  }
};

int main()
{
  vector<U> vector1(1000);
  for (int i = 0; i != 1000; ++i)
    vector1[i].id = 1000 - i - 1;
  sort(vector1.begin(), vector1.end(), not2(U()));
```

```
  for (int k = 0; k != 1000; ++k)
    assert (vector1[k].id == k);
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 6

Продемонстрируйте использование обобщенного алгоритма sort с бинарным предикатом.

```
#include <iostream>
#include <algorithm>
#include <cassert>
#include <functional>
using namespace std;

int main() {
  cout << "Using the generic sort algorithm "
       << "with a binary predicate." << endl;
  int a[1000];
  int i;
  for (i = 0; i < 1000; ++i)
    a[i] = i;
  random_shuffle(&a[0], &a[1000]);
  // Сортировка по возрастанию:
  sort(&a[0], &a[1000]);
  for (i = 0; i < 1000; ++i)
    assert (a[i] == i);

  random_shuffle(&a[0], &a[1000]);

  // Сортировка по убыванию:
  sort(&a[0], &a[1000], greater<int>());

  for (i = 0; i < 1000; ++i)
     assert (a[i] == 1000 - i - 1);
  cout << " --- Ok." << endl;
  return 0;}
```

4.2 Адаптеры

Адаптеры представляют собой компоненты STL, которые могут использоваться для изменения интерфейса другого компонента. Они определены как шаблоны классов, получающие тип компонента в качестве параметра. В STL имеются адаптеры контейнеров, адаптеры итераторов и функциональные адаптеры.

Адаптеры контейнеров. Адаптер контейнера stack может быть применен к vector, list или deque:

- stack<T> – представляет собой стек элементов типа T с реализацией по умолчанию, использующей deque;
- stack<T, vector<T>> – представляет собой стек элементов типа T с реализацией, использующей vector;
- stack<T, list<T>> – представляет собой стек элементов типа T с реализацией, использующей list;
- stack<T, deque<T>> – представляет собой стек элементов типа T с реализацией, использующей deque (идентичен stack<T>).

Очередь отличается от стека тем, что элементы в нее вставляются с одного конца, а извлекаются с другого. О ней часто говорят как о буфере «первый вошел, первый вышел» – first-in, first-out (FIFO), в отличие от «последний вошел, первый вышел» – last-in, first-out (LIFO) для стека. Очередь может создаваться адаптацией контейнеров list и deque:

- queue<T> – представляет собой очередь элементов типа T с реализацией по умолчанию, использующей deque;
- queue<T, list<T>> – представляет собой очередь элементов типа T с реализацией, использующей list;
- queue<T, deque<T>> – представляет собой очередь элементов типа T с реализацией, использующей deque (идентична queue<T>).

Очередь с приоритетами представляет собой тип последовательности, в которой элемент, доступный для выборки, является наибольшим в последовательности при некотором способе упорядочения. Как vector<T>, так и deque<T> предоставляют итераторы с произвольным доступом и все необходимые операции для очереди с приоритетами:

- priority_queue<int> – хранит значения типа int в реализации контейнера по умолчанию (vector) и использует функциональный объект сравнения по умолчанию (less<int>);
- priority_queue<int, vector<int>, greater<int>> – хранит значения типа int в контейнере типа vector и использует встроенный оператор >, определенный для int;
- priority_queue<float, deque<float>, greater<float>> – хранит значения типа float в контейнере deque и использует встроенный оператор >, определенный для float.

Адаптеры итераторов. Адаптеры итераторов представляют собой компоненты STL, которые могут использоваться для изменения интерфейса итератора. В STL предопределен адаптер итераторов только одного вида – адаптер обратного итератора, который преобразует данный двунаправленный итератор или итератор с произвольным доступом в итератор с обратным направлением обхода. Преобразованный итератор имеет тот же интерфейс, что и исходный. Этот адаптер определен как шаблон класса, который получает тип итератора в качестве параметра.

Функциональные адаптеры. Функциональные адаптеры помогают создавать более широкий набор функциональных объектов. Использование функциональных адаптеров часто оказывается более простым, чем непосредственное создание нового типа функционального объекта при помощи определения структуры или класса. STL предоставляет три категории функциональных адаптеров: связыватели (binders), инверторы (negators) и адаптеры для указателей на функции (adaptors for pointer to functions). Эти адаптеры определены как шаблоны классов, но сопровождаются шаблонами функций, которые делают применение классов более удобными.

Связыватель представляет собой разновидность функционального адаптера, используемого для преобразования бинарных функциональных объектов в унарные путем связывания аргумента с некоторым конкретным значением.

Инвертор представляет собой разновидность функционального адаптера, используемую для обращения смысла функционального объекта предиката. В STL имеется два инвертора – not1 и not2.

Адаптеры указателей на функции предоставляются для того, чтобы позволить указателям на унарные и бинарные функции работать вместе с другими функциональными адаптерами библиотеки.

4.2.1 Задания для выполнения лабораторной работы № 7 «Адаптеры»

Вариант 1

Продемонстрируйте использование адаптера стека.

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

int main()
{
 cout << "Illustrating the stack adaptor." << endl;
 int thedata[] = {45, 34, 56, 27, 71, 50, 62};
 stack<int> s;
 cout << "The stack size is now " << s.size() << endl;
 int i;
 cout << "Pushing 4 elements " << endl;
 for (i = 0; i < 4; ++i)
  s.push(thedata[i]);
 cout << "The stack size is now " << s.size() << endl;
 cout << "Popping 3 elements " << endl;
 for (i = 0; i < 3; ++i) {
  cout << s.top() << endl;
  s.pop();
 }
 cout << "The stack size is now " << s.size() << endl;
 cout << "Pushing 3 elements " << endl;
 for(i = 4; i < 7; ++i)
  s.push(thedata[i]);
 cout << "The stack size is now " << s.size() << endl;
 cout << "Popping all elements" << endl;
```

```
while (!s.empty()) {
   cout << s.top() << endl;
   s.pop();
  }
  cout << "The stack size is now " << s.size() << endl;
  return 0;
}
```

Вариант 2

Продемонстрируйте использование адаптера очереди.

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

int main()
{
  cout << "Illustrating the queue adaptor." << endl;
  int thedata[] = {45, 34, 56, 27, 71, 50, 62};
  queue<int, list<int> > q;
  cout << "The queue size is now " << q.size() << endl;
  int i;
  cout << "Pushing 4 elements " << endl;
  for (i = 0; i < 4; ++i)
    q.push(thedata[i]);
  cout << "The queue size is now " << q.size() << endl;
  cout << "Popping 3 elements " << endl;
  for (i = 0; i < 3; ++i) {
    cout << q.front() << endl;
    q.pop();
  }
  cout << "The queue size is now " << q.size() << endl;
  cout << "Pushing 3 elements " << endl;
  for(i = 4; i < 7; ++i)
    q.push(thedata[i]);
  cout << "The queue size is now " << q.size() << endl;
```

```
  cout << "Popping all elements" << endl;
  while (!q.empty()) {
    cout << q.front() << endl;
    q.pop();
  }
  cout << "The queue size is now " << q.size() << endl;
  return 0;
}
```

Вариант 3

Продемонстрируйте использование адаптера очереди с приоритетами.

```
#include <iostream>
#include <queue> // Определены как queue, так и priority_queue
using namespace std;

int main()
{
  cout << "Illustrating the priority_queue adaptor." << endl;
  int thedata[] = {45, 34, 56, 27, 71, 50, 62};
  priority_queue<int> pq;
  cout << "The priority_queue size is now " << pq.size()
       << endl;
  int i;
  cout << "Pushing 4 elements " << endl;
  for (i = 0; i < 4; ++i)
    pq.push(thedata[i]);
  cout << "The priority_queue size is now " << pq.size()
       << endl;
  cout << "Popping 3 elements " << endl;
  for (i = 0; i < 3; ++i) {
    cout << pq.top() << endl;
    pq.pop();
  }
  cout << "The priority_queue size is now " << pq.size()
       << endl;
  cout << "Pushing 3 elements " << endl;
```

```
  for(i = 4; i < 7; ++i)
    pq.push(thedata[i]);
  cout << "The priority_queue size is now " << pq.size()
       << endl;
  cout << "Popping all elements" << endl;
  while (!pq.empty()) {
    cout << pq.top() << endl;
    pq.pop();
  }
  cout << "The priority_queue size is now " << pq.size()
       << endl;
  return 0;
}
```

Вариант 4

Продемонстрируйте использование прямого и обратного обхода.

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

template <typename Container>
void display(const Container& c)
{

  cout << "Elements in normal (forward) order:  ";
  typename Container::const_iterator i;
  for (i = c.begin(); i != c.end(); ++i)
    cout << *i << "  ";
  cout << endl;

  cout << "Elements in reverse order:           ";
  typename Container::const_reverse_iterator r;
  for (r = c.rbegin(); r != c.rend(); ++r)
    cout << *r << "  ";
  cout << endl;
```

```
}

int main()
{
  cout << "Normal and reverse iteration through a vector:\n";
  vector<int> vector1;
  vector1.push_back(2);
  vector1.push_back(3);
  vector1.push_back(5);
  vector1.push_back(7);
  vector1.push_back(11);

  display(vector1);

  cout << "Normal and reverse iteration through a list:\n";
  list<int> list1(vector1.begin(), vector1.end());

  display(list1);
  return 0;
}
```

Вариант 5

Продемонстрируйте использование алгоритма find с обычными и обратными итераторами.

```
#include <iostream>
#include <vector>
#include <algorithm> // Для алгоритма find
#include <iterator>
using namespace std;
template <typename Container>
Container make(const char s[])
{
  return Container(&s[0], &s[strlen(s)]);
}

int main()
{
```

```
  cout << "Using find with normal and reverse iteration:\n";
  vector<char> vector1 =
     make< vector<char> >("now is the time");
  ostream_iterator<char> out(cout, " ");

  vector<char>::iterator i =
   find(vector1.begin(), vector1.end(), 't');
  cout << "chars from the first t to the end: ";
  copy(i, vector1.end(), out); cout << endl;

  cout << "chars from the last t to the beginning: ";
  vector<char>::reverse_iterator r =
   find(vector1.rbegin(), vector1.rend(), 't');
  copy(r, vector1.rend(), out); cout << endl;

  cout << "chars from the last t to the end: ";
  copy(r.base() - 1, vector1.end(), out); cout << endl;
  return 0;
}
```

Вариант 6

Продемонстрируйте сортировку вектора в возрастающем порядке членов id.

```
#include <iostream>
#include <cassert>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

class U : public binary_function<U, U, bool> {
public:
  int id;
  bool operator()(const U& x, const U& y) const {
    return x.id >= y.id;
  }
  friend ostream& operator<<(ostream& o, const U& x) {
```

```
    o << x.id;
    return o;
  }
};

int main()
{
  vector<U> vector1(1000);
  for (int i = 0; i != 1000; ++i)
    vector1[i].id = 1000 - i - 1;
  sort(vector1.begin(), vector1.end(), not2(U()));
  for (int k = 0; k != 1000; ++k)
    assert (vector1[k].id == k);
  cout << " --- Ok." << endl;
  return 0;
}
```

Вариант 7

Продемонстрируйте применение адаптера для указателей на функции

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

bool less1(const string& x, const string& y)
{
  return x < y;
}

bool greater1(const string& x, const string& y)
{
  return x > y;
}

int main()
{
```

```
  cout << "Illustrating the use of an adaptor"
       << " for pointers to functions." << endl;

  typedef
    set<string,
        pointer_to_binary_function<const string&,
                                   const string&, bool> >
    set_type1;

  set_type1 set1(ptr_fun(less1));

  set1.insert("the");
  set1.insert("quick");
  set1.insert("brown");
  set1.insert("fox");

  set_type1::iterator i;
  for (i = set1.begin(); i != set1.end(); ++i)
    cout << *i << " ";
  cout << endl;

  set_type1 set2(ptr_fun(greater1));

  set2.insert("the");
  set2.insert("quick");
  set2.insert("brown");
  set2.insert("fox");

  for (i = set2.begin(); i != set2.end(); ++i)
    cout << *i << " ";
  cout << endl;
  return 0;
}
```

Библиографический список

1 Мюссер, Д. И. C++ и STL : справочное руководство / Д. И. Мюссер, Ж. Д. Дердж, А. Сейни. – 2-е изд. (серия C++ in Depth). – Москва : ООО «И.Д. Вильямс», 2010. – 432 с.

2 Аммерааль, Л. STL для программистов на C++ / Л. Аммерааль. – Москва : ДМК, 1999. – 240 с.

3 Мейерс, С. Эффективное использование STL / С. Мейерс. – Санкт-Петербург : Питер, 2002. – 224 с.

4 Остерн, М. Г. Обобщенное программирование и STL: Использование и наращивание стандартной библиотеки шаблонов C++ / М. Г. Остерн. – Санкт-Петербург : Невский диалект, 2004. – 544 с.

5 Джосьютис, Н. C++ Стандартная библиотека. Для профессионалов / Н. Джосьютис. – Санкт-Петербург : Питер, 2004. – 730 с.

6 Уилсон, М. Расширение библиотеки STL для C++. Наборы и итераторы / М. Уилсон. – Москва : ДМК Пресс, 2008. – 608 с.

7 Халперн, П. Стандартная библиотека C++ на примерах / П. Халперн. – Москва : Издательский Дом «Вильямс», 2001. – 336 с.

Учебное издание

Маер Алексей Владимирович

Введение в стандартную библиотеку шаблонов (STL)

Учебное пособие

Редактор Л. П. Чукомина

Подписано в печать 19.06.20	Формат 60x84 1/16	Бумага 80 г/м^2
Печать цифровая	Усл. печ. л. 5,44	Уч.–изд. л. 5,44
Заказ № 38	Тираж 100	

Библиотечно-издательский центр КГУ.
640020, г. Курган, ул. Советская, 63/4.
Курганский государственный университет.