Министерство образования Московской области
Государственный университет «Дубна»
Филиал «Протвино»
Кафедра «Информационные технологии»

Т. Н. Кульман

Использование стандартной библиотеки шаблонов STL в программировании структур и алгоритмов обработки данных

УЧЕБНОЕ ПОСОБИЕ

Рекомендовано учебно-методическим советом
университета «Дубна» в качестве учебного пособия
для студентов, обучающихся по направлению подготовки
«Информатика и вычислительная техника»

Дубна
2021

УДК 004.05

ББК 32.973я73

К906

Рецензент:

Д.В. Тараканов, к.т.н., доцент кафедры автоматики и компьютерных систем,
БУ ВО «Сургутский государственный университет»

Кульман Т.Н.

К906 «Использование стандартной библиотеки шаблонов STL в программировании структур и алгоритмов обработки данных». Учебное пособие. – Дубна: Гос. университет «Дубна», 2021. – 83 с.

ISBN 978-5-89847-639-7

Стандартная библиотека шаблонов STL является современным инструментом программирования, содержащим реализацию многих структур и алгоритмов, а также повышающим эффективность труда программистов. Изучение использования библиотеки STL при разработке программ является важной составной частью обучения программированию, – при этом повышается надёжность программ, их переносимость, уменьшается время разработки.

В пособии приводится теоретический материал по шаблонам, контейнерам, алгоритмам, итераторам, функциональным объектам и другим базовым понятиям STL. Рассматриваются решения задач с указанием методических рекомендаций и комментариев, даются задания для самостоятельного выполнения и контрольные вопросы. Библиотека шаблонов применяется при изучении дисциплины «Структуры и алгоритмы обработки данных».

Пособие способствует пониманию сложных понятий, связанных с алгоритмами и структурами, получению новых знаний и приобретению опыта в программировании студентами на современном уровне.

Учебное пособие «Эффективное использование стандартной библиотеки шаблонов STL в программировании структур и алгоритмов обработки данных» предназначено для студентов очной и заочной форм обучения направления 09.03.01 «Информатика и вычислительная техника» (профиль подготовки «Программное обеспечение вычислительной техники и автоматизированных систем»).

УДК 004.05

ББК 32.97я73

ISBN 978-5-89847-639-7

Оглавление

Введение

В учебном пособии описывается применение стандартной библиотеки шаблонов (Standard Template Library – STL). Изучение использования библиотеки STL при разработке программ является важной составной частью обучения программированию, – при этом повышается надёжность программ, их переносимость, а также уменьшаются расходы на их создание.

Материал, изложенный в пособии, соответствует программе дисциплины «Структуры и алгоритмы обработки данных», которая посвящена изучению структур данных, применяемых в программировании, и современных методов построения алгоритмов. Библиотека STL, фактически, является стандартной частью языка программирования C++, на котором ведётся преподавание дисциплины «Структуры и алгоритмы обработки данных».

Базовыми понятиями библиотеки STL являются шаблоны, контейнеры, итераторы алгоритмы, функциональные объекты, адаптеры и др. В пособии излагается теоретический материал по всем этим понятиям. Прежде чем приступать к изучению библиотеки STL, необходимо рассмотреть шаблоны функций и классов, которые позволяют обрабатывать разнотипные данные. Рассматриваются отличия шаблонов функций от механизмов перегрузки функций.

Далее изучаются разные виды контейнеров, которые обладают большой гибкостью и функциональностью. Они поддерживают динамическое размещение элементов. В последовательных контейнерах элементы хранятся в том порядке, в котором они были введены. В ассоциативных контейнерах элементы упорядочены не зависимо от того, в каком порядке они были добавлены, и это позволяет реализовать быстрый поиск элементов. Существуют ещё адаптеры-контейнеры, которые построены на основе последовательных контейнеров.

Итераторы обеспечивают доступ к содержимому контейнеров, поэтому играют важную роль в библиотеке. С их помощью можно выполнять операции над элементами контейнеров. Существует несколько разновидностей итераторов, имеющих разные свойства. Именно итераторы являются связующим звеном между контейнерами и алгоритмами. В пособии рассматривается множество алгоритмов, позволяющих решать самые разные задачи.

Некоторые алгоритмы можно модифицировать, включая в их параметры предикаты или функциональные объекты. Функциональными объектами являются классы, в которых присутствует специальный оператор вызова, позволяющий удобно описывать обработку данных.

В рассмотрение включены битовые множества, предназначенные для обработки длинной последовательности битов, применяемой для компактного хранения флагов в наборах элементов или условий.

Алгоритмы библиотеки STL работают не только со структурами данных самой STL, но также и со структурами данных, созданными пользователями. Кроме того, структуры данных библиотеки шаблонов могут взаимодействовать с алгоритмами, разработанными пользователями.

Учебное пособие предназначено для студентов 2 и 3 курсов очной и заочной форм обучения направления «Информатика и вычислительная техника». Теоретический материал пособия иллюстрируется практическими работами для изучения и освоения основных средств и методов программирования с использованием библиотеки шаблонов.

Самую подробную информацию по структурам и алгоритмам на C++ можно найти в следующих книгах [1 – 6], по STL – в книгах и Интернет-источниках [7 – 16]. В дополнении к задачам для самостоятельного решения, приведённым в учебном пособии, много задач представлено в работе [17].

Пособие написано на основе более чем 10-ти летнего опыта преподавания

дисциплины «Структуры и алгоритмы обработки данных» с включением теоретической информации и практических работ по STL.

Цель и задачи пособия

Целью учебного пособия «Использование стандартной библиотеки шаблонов STL в программировании структур и алгоритмов обработки данных» является подготовка специалистов в важном направлении информационных технологий, заключающемся в обучении студентов программированию на современном уровне с применением библиотеки STL. Знакомство с таким подходом и закрепление практических навыков позволяет сформировать соответствующие компетенции у студентов для решения практических задач.

Задачами пособия являются:

- изложение основных положений теории структур и алгоритмов обработки данных, заложенных в библиотеке шаблонов;
- применение теоретических знаний при создании программ и приложений;
- демонстрация применения библиотеки STL на примерах;
- обучение использованию библиотеки шаблонов при разработке программ.

Каждый раздел пособия содержит теоретический и практический материалы. В теоретической части даются основные определения, рассматриваются базовые понятия, способы применения алгоритмов и структур библиотеки STL. В практической части приводится множество программ на C++, фрагменты готовых проектов, которые сопровождаются подробными методическими указаниями, объяснениями и комментариями. Также сформулированы задачи для самостоятельного решения и приведены контрольные вопросы.

1. Перегрузка функций. Шаблоны функций и классов

В данном разделе изучаются основные понятия перегрузки функций, шаблонов функций и классов, проводится обучение созданию шаблонов функций и классов.

Основные понятия:

- перегрузка функций,
- шаблонные функции,
- шаблонные классы,
- пространство имён.

Перегрузка функций

Рассмотрим пример: нужно запрограммировать функцию, которая выводила бы на экран элементы массива. Чтобы написать такую функцию, необходимо знать тип данных массива, который будем выводить на экран. Но типы данных могут быть разными: `int, double, float, char` и др.

В данном случае нужно запрограммировать 4 функции, которые выполняют одни и те же действия, но для различных типов данных.

Чтобы решить эту задачу, воспользуемся понятием *перегрузки функций.*

```
oid printArray (const int *array, int count) // Формальные параметры
{
      // Перегрузка функции printArray для вывода массива на экран
      // Массив содержит данные типа int
      for (int i = 0; i < count; ++i)
            cout << array[i] << "   ";
      cout << endl;
}

      // Массив содержит данные типа double
void printArray (const double *array, int count) // Формальные параметры
{
      for (int i = 0; i < count; ++i)
            cout << array[i] << "   ";
      cout << endl;
}
```

Эти две функции одинаковы, отличие состоит лишь в типе параметров. Аналогично напишем функции для типов данных `float` и `char.`

Для вызова этих функций составим код:

```
const int count = 7;
int Ar[count] = {1, 6, 8, 15, 37, 82, 45};
double Ar1[count] = {3.4, 12.7, 34.9, 18.5, 3.02, 78.2, 93.7};
printArray (Ar, count);  // Фактические параметры
printArray (Ar1, count); // Фактические параметры
```

Таким образом, перегрузка функций представляет собой создание нескольких функций с одним и тем же именем, но с разными параметрами (по количеству и/или по типам). При вызове функции компилятор по количеству и типам аргументов определяет, какую версию функции следует вызвать.

Здесь уместно подчеркнуть разницу между формальными и фактическими параметрами (аргументами). Формальные параметры указываются при описании функции, а фактические – при вызове функции.

Рассмотрим другой подход к решению данной задачи.

Он основывается на понятии *шаблона*. Определены два вида шаблонов: шаблоны

функций и шаблоны классов (шаблонные классы). Шаблоны функций и шаблонные классы – это инструкции, согласно которым создаются локальные версии функций и классов для определенного набора параметров и типов данных.

Шаблонные функции

Цель введения *шаблонов функций* – автоматизация создания функций, которые могут обрабатывать разнотипные данные. В отличие от механизма перегрузки, когда для каждого набора формальных параметров определяется своя функция, шаблон семейства функций определяется один раз, но это определение параметризуется. Параметризовать в шаблоне функций можно тип возвращаемого функцией значения и типы любых параметров, количество и порядок размещения которых должны быть фиксированы. Для параметризации используется список параметров шаблона.

Напишем функцию печати массива с использованием шаблона.

В определении шаблона семейства функций применяется служебное слово **template** (шаблон). Для параметризации используется список формальных параметров шаблона, который заключается в угловые скобки <>.

```
// Шаблон функции printArray для вывода массива на экран
template <typename T>
// T – тип данных, задаваемый аргументом при вызове функции printArray
void printArray (const T *array, int count)
{
      for (int i = 0; i < count; ++i)
            cout << array[i] << "   ";
      cout << endl;
}
```

Вызов шаблонной функции:

```
const int count = 7;
int Ar[count] = {1, 6, 8, 15, 37, 82, 45};
double Ar1[count] = {3.4, 12.7, 34.9, 18.5, 3.02, 78.2, 93.7};
printArray<int> (Ar, count);
printArray<double> (Ar1, count);
```

В качестве еще одного примера рассмотрим шаблон семейства функций для обмена значениями двух передаваемых им параметров.

```
template <class T> void swap(T *x, T *y)
{
      T z = *x;
      *x = *y;
      *y = z;
}
```

Здесь параметр `T` используется не только в заголовке для спецификации формальных параметров, но и в теле функции для определения переменной `z`.

Вызов шаблонной функции:

```
int x = 57;
int y = 83;
float x1 = 4.6;
float y1 = 7.2;
cout << x << " " << y << endl;
swap <int> (x, y);
cout << x << " " << y;
cout << endl;
```

```
cout << x1 << " " << y1 << endl;
swap <float> (x1, y1);
cout << x1 << " " << y1 << endl;
```

Вывод программы:
```
57   83
83   57
4.6  7.2
7.2  4.6
```

Если в программе присутствует шаблон `swap()`, значения параметров `long k=4, d=8;` вызов `swap(&k, &d);` то компилятор сформирует определение функции:

```
void swap(long *x, long *y)
{
      long x = *x;
      *x = *y;
      *y = x;
}
```

Затем будет выполнено обращение именно к этой функции и значения переменных `k, d` поменяются местами.

Если в той же программе фактические параметры `double a = 2.44, b = 66.3;` вызов `swap(&a, &b);` то сформируется и выполнится функция:

```
void swap(double *x, double *y)
{
      double x = *x;
      *x = *y;
      *y = x;
}
```

Шаблонные классы

Пусть нам нужен класс `Pair`, чтобы хранить пары значений. В первом случае оба значения пары принадлежат типу `double` (класс `PairDouble`), во втором – типу `int` (класс `PairInt`). Рассмотрим класс `PairDouble`.

```
class PairDouble
{
public:
      PairDouble(double x1, double y1) : x(x1), y(y1) {}
      // Начальная инициализация x(x1), y(y1)
      void showQ();    // Определение метода showQ()
private:
      double x, y;
};

void PairDouble::showQ()  // Описание метода showQ()
{
      cout << x / y << endl;
}
```

Создадим экземпляр класса n:

```
PairDouble n(3.6, 4.9);
n.showQ();
```

Вывод программы:
```
0.7346
```

Аналогично создаётся класс `PairInt` для обработки данных типа `int`. Вместо двух классов `PairDouble` и `PairInt` создадим один шаблонный класс `Pair`. Параметр `T` описывает различные типы данных.

```
template <class T>
class Pair
{
	public:
		Pair(T x1, T y1) : x(x1), y(y1) {}
		// Начальная инициализация x(x1), y(y1)
		void showQ();
	private:
		T x, y;
};

template <class T>
void Pair<T>::showQ()
{
	cout << x / y << endl;
}
```

Воспользуемся этим классом, создав два экземпляра **c** и **d**:

```
Pair<double> c(25.7, 12.3);
Pair<int> d(25, 12);
c.showQ();
d.showQ();
```

Вывод программы:

```
2.08943
2
```

Шаблонные функции и классы STL доступны в виде файлов заголовков, которые можно использовать, не вдаваясь в подробности их программирования. Процесс создания конкретной версии шаблонной функции или класса называется *инстанцированием шаблона* или *созданием экземпляра.*

Пространство имён

Если программа имеет большой объём или состоит из многих файлов, нужно принять меры во избежание *конфликта имен*. Конфликт имен возникает, когда два одинаковых идентификатора находятся в одной области видимости, и компилятор не может понять, какой из идентификаторов нужно использовать в конкретной ситуации.

Концепция *пространства имен* может быть хорошим способом решения этой задачи. Пространство имен определяет область кода, в которой гарантируется уникальность всех идентификаторов. Чтобы избежать конфликта имён, необходимо объявлять собственные пространства имён, используя ключевое слово **namespace.** Для получения доступа к пространству имён необходимо применять оператор разрешения области видимости (::).

Рассмотрим примеры.

```
// Пространство имён
	namespace A
	{
		int i = 10;
	}
	namespace B
	{
		int i = 20;
	}
```

```
int nameSP()
{
	fA();
	fB();
	cout << "In nameSP(): " << A::i << " " << B::i << endl;
	//  cout << i << endl;  Здесь это недопустимо
	using A::i;
	cout << i << endl;  // Разрешено
	return 0;
}

void fA()
{
	using namespace A;
	cout << "In fA(): " <<
		A::i << " " << B::i << " " << i << endl;
}

void fB()
{
	using namespace B;
	cout << "In fB(): " <<
		A::i << " " << B::i << " " << i << endl;
}
```

Результат работы программы:

```
In  fA(): 10 20 10
In  fB(): 10 20 20
In nameSP(): 10 20
10
Для продолжения нажмите любую клавишу . . .
```

Благодаря идентификаторам `A` и `B` можно впоследствии ссылаться на эти пространства имен. Для пространства имен A можем написать одно из выражений:

```
A:: …
using namespace A;
using A::i;
```

Ранее был описан класс `PairDouble`, в котором *определён* (declare) метод `showQ()`. Однако сам метод *описывается* (definition) отдельно от класса и, чтобы подчёркнуть, что метод `showQ()` будет использоваться в классе `PairDouble`, применяется понятие пространства имён `PairDouble::showQ()`.

```
class  PairDouble
{
public:
	PairDouble(double xl, double yl) : x(xl), y(yl) {}
	// Начальная инициализация x(xl), y(yl)
	void showQ();
private:
	double x, y;
};
void PairDouble::showQ()
{
	cout << x / y << endl;
}
```

Задания для самостоятельного выполнения

1. Написать шаблонный класс для вычисления суммы одномерного массива чисел (с разными типами данных).
2. Написать шаблонный класс `rectangle` (прямоугольник) и определить его периметр и площадь.
3. Написать шаблонную функцию для нахождения максимального элемента в двумерных массивах для различных арифметических типов.

Контрольные вопросы

1. Что такое перегрузка функций?
2. Чем понятие «перегрузка функций» отличается от понятия «шаблон» в STL?
3. Какие виды шаблонов вы знаете?
4. Что означают параметры шаблонов и как они записываются?
5. Назовите основные возможности шаблонных функций.
6. Назовите основные возможности шаблонных классов.
7. Что такое инстанцирование шаблона?
8. Какие преимущества даёт использование перегрузки по сравнению с применением шаблонов и наоборот?
9. Что такое «пространство имён» и для какх целей оно используется?
10. Объясните разницу в понятиях определён (declare) и описан (definition) при создании методов класса.

2. Последовательные контейнеры. Операции с последовательными контейнерами

В STL существуют разные виды контейнеров.

Цель данного раздела состоит в том, чтобы изучить основные понятия последовательных контейнеров, познакомиться с итераторами, а также научиться создавать последовательные контейнеры, применять к ним операции и алгоритмы.

Основные понятия:

- последовательные контейнеры,
- итераторы,
- операции с последовательными контейнерами,
- алгоритмы.

В библиотеке STL содержится 5 основных компонент:

- контейнеры (последовательные и ассоциативные),
- итераторы,
- алгоритмы,
- функциональные объекты,
- адаптеры.

Все эти компоненты мы и будем далее изучать.

Тип контейнера следует выбирать в зависимости от задачи.

Последовательные контейнеры

Контейнеры – это объекты, предназначенные для хранения элементов одинакового типа. Контейнеры являются шаблонными классами. Последовательные контейнеры организуют конечную последовательность элементов в линейном порядке. Существует три вида последовательных контейнеров:

– **`vector`** (вектор), – элементы последовательности сохраняются в массиве переменной длины, поддерживает вставки и удаления в конце последовательности;

– **`list`** **(**список), – двунаправленный связный список, используется, когда производятся частые вставки и удаления в/из середины последовательности;

– **`deque`** (двусторонняя очередь), – применяется, когда вставки и удаления производятся в начале или конце последовательности.

– Довольно часто к последовательным контейнерам относят и обыкновенный **`массив.`**

Чтобы использовать данные контейнеры, необходимо включать в проект заголовочные файлы:

- **`#include <vector>`**
- **`#include <list>`**
- **`#include <deque>`**

Итераторы

Итераторы – объекты, обеспечивающий доступ к содержимому контейнера. Первоначально можно представлять итератор как аналог указателя. Однако дальнейшее изучение покажет, что итераторы обладают многими другими свойствами.

Поскольку все контейнеры являются шаблонными классами, они содержат некоторые стандартные методы, присутствующие в контейнерах. Рассмотрим наиболее часто применяемые:

- `begin()` – возвращает итератор, указывающий на начало контейнера.
- `end()` – возвращает итератор, указывающий на элемент, следующий за последним элементом контейнера.
- `empty()` – определяет, является ли контейнер пустым.

– `size()` – определяет размер контейнера.
– `insert()` – вставляет элемент в контейнер.
– `erase()` – удаляет элемент или несколько элементов из контейнера.

При дальнейшем изложении будут описаны и другие методы.

Чтобы понять работу с итераторами, необходимо знать, что метод `begin()` устанавливает итератор на первый элемент контейнера, а метод `end()` – за последний элемент контейнера.

```
1     3     5     6     7
↑                       ↑
v.begin()               v.end()
```

Рассмотрим программу, которая читает с клавиатуры 5 целых чисел и выводит их на экран.

```
int vector_STL()
{
   // Чтение и вывод 5 целых чисел
   // Используется вектор
      vector<int> v;
      int x, i;
      cout << "Введите 5 целых чисел: \n";
      for (i = 0; i != 5; ++i)
      {
            cin >> x;
            v.push_back(x); // Метод ввода данных в вектор
      }
      vector<int>::iterator iv; // Создание итератора
      for (iv = v.begin(); iv != v.end(); ++iv)
            cout << *iv << " ";
      cout << endl;
      return 0;
}
```

Результат:

```
Введите 5 целых чисел:
2
6
10
-4
7
2 6 10 -4 7
Для продолжения нажмите любую клавишу . . .
```

Шаблон **`vector`** можно рассматривать как массив переменной длины. Сначала его длина равна `0`.

`vector<int> v;` – описание класса *вектор* **`v`** с целым типом данных.

`v.push_back(x);` – метод добавления элемента в конец вектора.

`vector<int>::iterator iv;` – определение *итератора* `iv`, соответствующего типу контейнера `vector<int>`.

В данном примере итератор используется аналогично указателю.

Рассмотрим вывод элементов массива `int a[N]` с использованием указателя:

```
int a[N], *p;  // &a[0] и a эквивалентны.
for (p = a; p != a + N; ++p)
      cout << *p << " ";
```

Вывод элементов контейнера с помощью итератора выглядит так:

```
for (i = v.begin(); i != v.end(); ++i)
      cout << *i << " ";
```

Для итератора `i` определены также операторы инкремента/декремента ++ и --, как в префиксном, так и в постфиксном вариантах.

В приведенном цикле лучше не заменять знак **`!=`** на знак **`<`**. В данном случае знак **`<`** будет работать, но в других контейнерах этого не произойдёт. Однако, оператор **`!=`** работает во всех случаях.

В программе `vector_STL()` 3 раза встречается слово `vector`.

```
#include <vector>  // Подключение библиотеки
. . .
vector<int> v;     // Создание вектора v
. . .
vector<int>::iterator iv;  // Определение итератора iv
```

При определении итератора используется понятие пространства имён.

С помощью STL мы можем использовать (двойные) связные списки, не программируя их самостоятельно. Все, что нам требуется для программы `list_STL()`, – всюду заменить слово `vector` на `list`, как показано в следующей программе:

```
int list_STL()
{
   // Чтение и вывод 5 целых чисел
   // Используется список
      list<int> v;
      int x, i;
      cout << "Введите 5 целых чисел: \n";
      for (i = 0; i != 5; ++i)
      {
            cin >> x;
            v.push_back(x);
      }
      list<int>::iterator iv;
      for (iv = v.begin(); iv != v.end(); ++iv)
            cout << *iv << " ";
      cout << endl;
      return 0;
}
```

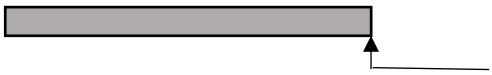
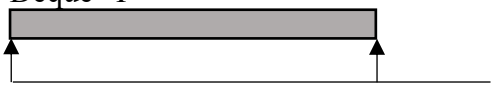
Результат:

```
Введите 5 целых чисел:
12
-5
8
4
9
12 -5 8 4 9
Для продолжения нажмите любую клавишу . . .
```

Программа чтения и вывода 5 целых чисел также будет правильно выполняться, если заменить слово `list` на `deque` (двусторонняя очередь), что дает нам третье решение. Пользователь не заметит никаких различий в поведении этих трех версий программы, но внутреннее представление данных будет различаться. Это скажется на наборе доступных операций, которые смогут выполняться эффективно.

Свойства трех последовательных контейнеров

Для данного типа `T` последовательные контейнеры `vector<T>, deque<T>` и `list<T>` обладают следующими свойствами [7]:

Vector<T> 	Вставка и удаление в конце вектора. Произвольный доступ.
Deque<T> 	Вставка и удаление в начале и в конце двусторонней очереди. Произвольный доступ.
List<T> 	Вставка и удаление в любом месте. Нет произвольного доступа.

Операции для работы с последовательными контейнерами

Операция	Функция	vector	deque	list
Вставить в конце	***push back()***	✓	✓	✓
Удалить в конце	***pop_back()***	✓	✓	✓
Вставить в начале	***push_front()***	-	✓	✓
Удалить в начале	***pop_front()***	-	✓	✓
Вставить в любом месте	***insert()***	**(✓)**	**(✓)**	✓
Удалить в любом месте	***erase()***	**(✓)**	**(✓)**	✓
Отсортировать	***sort()*** **(алгоритм)**	✓	✓	-

Галочка (✓) означает, что функции `insert()` и `erase()` для вектора `vector` и двусторонней очереди `deque` выполняются гораздо медленнее, чем для списков, т.е. выполнение функций `insert()` и `erase()` занимает линейное время для векторов и двусторонних очередей, а это означает, что время их выполнения пропорционально длине последовательности, хранящейся в контейнере. В противоположность этому все операции, помеченные галочкой ✓(без скобок), выполняются за постоянное время, то есть время, необходимое для их выполнения, не зависит от длины последовательности.

Рассмотрим программу использования функций-членов вставки и удаления для контейнера `vector` – `vector_insdel()`, для вывода на экран содержимого вектора – программу `show_Vector()` и маленькую программку `vec_ins()`, которая позволяет вводить в указанную позицию вектора заданное число.

Прежде чем выполнять какую-либо операцию над числом из вектора, *необходимо установить итератор в соответствующую позицию,* т.е. определить положение используемого числа.

Задание для программы:

1. Создать вектор. Ввести переменное количество ненулевых целых чисел (ввод завершается нулем) и вывести их на экран.
2. Используя введённые в п.1 данные, применить функции вставки и удаления для выполнения следующих действий (после каждого действия следует вывод на экран):
   a. Вставить число 20 во 2-ю позицию.

b. Удалить первый элемент.
c. Добавить число 80 в конец.
d. Вставить число 25 в 5-ю позицию.
e. Перед последним элементом вставить число 63.
f. Отсортировать.
g. Вставить число 22 в позицию 4.

```
void show_Vector(const char* str, const vector<int>& v)
{
   // Печать содержимого вектора
   // Первый параметр – массив типа char для комментирования
   // выполняемых действий с вектором
      vector<int>::const_iterator i;  // Определение итератора i
      cout << str << endl;
      for (i = v.begin(); i != v.end(); ++i) // Вывод содержимого вектора
            cout << *i << " ";
      cout << endl;
}

void vec_ins(int pos, int val, vector<int>& v)
{
   // Ввод в указанную позицию вектора заданного числа
      vector<int>::iterator i = v.begin() + pos;
      v.insert(i, val);
}

void vector_insdel()
{
      vector<int> v;
      int x;
      cout << "Введите серию чисел, заканчивающуюся 0:\n";
      while (cin >> x, x != 0)
            v.push_back(x);
      vector<int>::iterator i = v.begin(); // Определение итератора i и
      // установка его на начало вектора
      show_Vector("Начальное значение вектора: ", v);
      i = i + 2;          // Модификация итератора
      v.insert(i, 20);    // Вставка во 2-ю позицию вектора
      show_Vector("Вставить число 20 во 2-ю позицию: ", v);
      i = v.begin() + 1;
      v.erase(i);
      show_Vector("Удалить первый элемент: ", v);
      v.push_back(80);
      show_Vector("Добавить число 80 в конец: ", v);
      i = v.begin() + 5;
      v.insert(i, 25);
      show_Vector("Вставить число 25 в 5-ю позицию:", v);
      i = v.begin();
      v.erase(i);
      show_Vector("Удалить нулевой элемент:", v);
      i = v.end() - 1;
      v.insert(i, 63);
      show_Vector("Вставить число 63 перед последним элементом: ", v);
      cout << endl;
      sort(v.begin(), v.end());
```

```
        show_Vector("Отсортировать вектор: ", v);
        cout << endl;
        cout << "Вставка с использованием функции vec_ins(pos, val, v)" <<
endl;
        int pos = 4;
        int val = 22;
        vec_ins(pos, val, v);
        show_Vector("Вставить число 22 в позицию 4: ", v);
}
```

Необходимо напомнить, что нумерация элементов в векторе начинается с `0`.

Результат работы программы:

```
Введите серию чисел, заканчивающуюся 0:
4 -5 10 37 81 -8 10 0
Начальное значение вектора:
4 -5 10 37 81 -8 10
Вставить число 20 во 2-ю позицию:
4 -5 20 10 37 81 -8 10
Удалить первый элемент:
4 20 10 37 81 -8 10
Добавить число 80 в конец:
4 20 10 37 81 -8 10 80
Вставить число 25 в 5-ю позицию:
4 20 10 37 81 25 -8 10 80
Удалить нулевой элемент:
20 10 37 81 25 -8 10 80
Вставить число 63 перед последним элементом:
20 10 37 81 25 -8 10 63 80

Отсортировать вектор:
-8 10 10 20 25 37 63 80 81

Вставка с использованием функции vec_ins(pos, val, v);
Вставить число 22 в позицию 4:
-8 10 10 20 22 25 37 63 80 81
Для продолжения нажмите любую клавишу . . .
```

Приращение итератора можно выполнять с помощь функции **`advance (i, n)`**, где `i` – итератор, `n` – приращение. Например, `advance (i, 3)`.

Замечания

Рассмотрим употребление спецификатора `const` в функции `show_Vector()`:

```
void show_Vector(const char *str, const vector<int> &v)
{     vector <int>::const_iterator i; . . .
```

Добавление спецификатора `const` к параметрам типа указатель или ссылка, как это сделано выше, является хорошей практикой, если такие параметры *не используются для модификации объектов*, на которые они указывают. Поскольку функция `show_Vector()` не модифицирует ни строку `str`, ни вектор `v`, поэтому применим к ним модификатор `const` (отсюда – два употребления слова `const` в первой строке).

Во второй строке объявляем переменную `i` типа `const_iterator`, чтобы иметь возможность использовать ее вместе с `v`. Это похоже на применение модификатора `const` к указателям: если хотим присвоить вышеобъявленный параметр `str` указателю `p`, можно сделать это, только использовав `const` при объявлении указателя `p`:

```
const char *str;
const char *p;   // const необходим при описании указателя p, поскольку str
p = str;         // описан как  const char *str
```

Алгоритмы

Алгоритмы выполняют обработку содержимого контейнеров. Существуют разные виды алгоритмов: немодифицирующие (поиска, подсчёта количества значений, удовлетворяющих определённым критериям и др.), модифицирующие (копирования, обмен местами двух элементов, удаление элементов с заданными значениями и т.д.), сортировки, перестановки, работа с множествами и др.

Для работы алгоритмов нужно подключить заголовочный файл `algorithm`, – **`#include <algorithm>.`**

Алгоритм *sort*

Отсортируем введенные пользователем числа в восходящем порядке (по умолчанию), используя контейнер `vector.`

```
#include <vector>
#include <algorithm>
int vector_sortl()
{
      // сортировка вектора
      vector<int>  v;
      int  x;
      cout << "Введите серию положительных чисел, заканчивающихся 0:\n";
      while (cin >> x, x != 0)  v.push_back(x);
      sort(v.begin(), v.end());
      cout << "После сортировки:  \n";
      vector<int>::iterator i;
      for (i = v.begin(); i != v.end(); ++i)
            cout << *i << " ";
      cout << endl;
      return 0;
}
```

Результат работы:

```
Введите серию положительных чисел, заканчивающихся 0:
24 38 5 19 87 62 35 17 6 14 0
После сортировки:
5 6 14 17 19 24 35 38 62 87
Для продолжения нажмите любую клавишу . . .
```

Вышеприведенный вызов алгоритма `sort` отличается от вызовов методов `push(), insert(), begin()` и др. Мы пишем не `v.sort(),` а просто `sort,` поскольку `sort` не является функцией-членом класса `vector,` а является шаблонной функцией. Технический термин, обозначающий такую шаблонную функцию в STL, – обобщенный (generic) алгоритм, или просто алгоритм.

Строчка `#include <algorithm>` необходима, поскольку используется алгоритм `sort,` описание которого находится в `<algorithm>.`

Алгоритм `sort` требует произвольного доступа к контейнерам, для которых он применяется. Такой доступ обеспечивают векторы, массивы и двусторонние очереди. Вызов `sort(v.begin(), v.end());` также будет работать, если в программе `vector_sortl()` заменим всюду `vector` на `deque,` *но не на `list.`* Список не обеспечивает произвольного доступа, поэтому к нему не применим алгоритм `sort.` Для сортировки списка используется метод `sort()` самого класса `list.` Например, `L.sort();`

Алгоритм *find*

Алгоритм `find` применяется для нахождения в контейнере первого вхождения искомого элемента. В следующем примере используется алгоритм `find` для поиска определенного элемента в двусторонней очереди `deque` (можно также применить рассматриваемую программу к вектору или списку).

При поиске искомого элемента могут возникнуть 3 ситуации: элемент не найден, найден внутри очереди или найден как первый элемент (для первого элемента необходимо откорректировать значение итератора).

В качестве комментария к алгоритму `find` необходимо заметить, что нам важен любой результат (найден или не найден элемент), поэтому после вызова `find` нужно проанализировать полученные данные, что и показано в приведенной программе.

```
1. #include <deque>
2. #include <algorithm>
3. int deque_find()
4. {
5. // Найти заданное значение в очереди
6. deque<int> Q;
7. int x, n = 0;
8. cout << "Введите не более 10 положительных чисел, заканчивающихся 0:\n";
9. for (n = 0; n < 10; n++)
10.      {
11.          cin >> x;
12.          if (x == 0)
13.          {
14.               if (n == 0) return 0;
15.               break;
16.           }
17.           Q.push_back(x);
18.      }
19.      cout << "Введите искомый элемент:  ";
20.      cin >> x;
21.      deque<int>::iterator iQ;
22.      cout << "\ndeque:  ";
23.      for (iQ = Q.begin(); iQ != Q.end(); ++iQ)
24.           cout << *iQ << " ";
25.      cout << endl;
26.      iQ = find(Q.begin(), Q.end(), x);
27.      if (iQ != Q.end())
28.          cout << "Позиция в очереди: " << iQ - Q.begin() <<
29.          " Значение: " << *iQ << "\n";
30.      if (iQ == Q.end())
31.          cout << "Не найдено\n";
32.      else
33.      {
34.          cout << "Найдено";
35.          if (iQ == Q.begin())
36.               cout << " на месте первого элемента ";
37.          else cout << " за элементом со значением: " << *--iQ;
38.      }
39.      cout << endl;
40.      return  0;
41.}
```

Результат (в случае, если искомый элемент найден):

```
Введите не более 10 положительных чисел, заканчивающихся 0:
2 56 71 35 90 82 37 4 10 3
Введите искомый элемент:  82

deque:  2 56 71 35 90 82 37 4 10 3
Позиция в очереди: 5 Значение: 82
Найдено за элементом со значением: 90
Для продолжения нажмите любую клавишу . . .
```

Результат (в случае, если искомый элемент не найден):

```
Введите не более 10 положительных чисел, заканчивающихся 0:
32 54 90 88 6 1 24 82 0
Введите искомый элемент:  12

deque:  32 54 90 88 6 1 24 82
Не найдено

Для продолжения нажмите любую клавишу . . .
```

Результат (в случае, если искомый элемент – первый):

```
Введите не более 10 положительных чисел, заканчивающихся 0:
11 23 90 56 71 45 0
Введите искомый элемент:  11

deque:  11 23 90 56 71 45
Позиция в очереди: 0 Значение: 11
Найдено на месте первого элемента
Для продолжения нажмите любую клавишу . . .
```

Комментарии

(6)	**`deque<int> Q;`** – создаём двустороннюю очередь Q, которая будет содержать данные целого типа.
(9-18)	Вводим не более 10 целых чисел. Последовательность чисел, если их количество меньше 10, заканчивается `0`. Для ввода данных, наряду с циклом **`while`** можно использовать и цикл **`for`**, – данная программа это иллюстрирует.
(20)	В переменную `x` вводим искомое число. **`cin >> x;`**
(21)	**`deque<int>::iterator iQ;`** – определяем итератор **`iQ.`**
(23)	С помощью разыменования итератора выводим данные очереди. **`for (iQ = Q.begin(); iQ != Q.end(); ++iQ)`** **`cout << *iQ << " ";`**
(26)	Ищем искомый элемент **`iQ = find(Q.begin(), Q.end(), x);`** Функция **`find()`** возвращает значение итератора, указывающего на первый найденный элемент или на **`Q.end(),`** если элемент не найден.
(28)	Чтобы было понятно, в какой позиции (по номеру) находится найденный элемент, вычисляем разность **`iQ - Q.begin()`** и выводим значение найденного элемента. Номер элементов в очереди начинается с `0.`
(30)	Если искомый элемент не найден (для этого сравнивается полученное значение **`iQ`** и значение, соответствующее функции **`Q.end(),`**

	if (iQ == Q.end()), то выдаётся соответствующее сообщение – Не найдено.
(35)	Если элемент найден, – выводится сообщение Найдено и производится дополнительный анализ для нахождения предыдущего элемента. Если найденный элемент является первым в очереди, то выводится сообщение "на месте первого элемента". Если он не первый, то выводится предыдущий элемент. Это достигается с помощью модификации итератора: ***--iQ.**

Рассмотрим ещё одну программу, в которой с помощью алгоритма find находим *не только первое вхождение* искомого элемента, но все его остальные вхождения. Будут указаны все позиции в последовательности, на которых находятся найденные элементы.

```
#include <vector>
#include <algorithm>
int vector_find()
{
      // Найти все вхождения заданного значения в векторе
       int A[] = {1, 49, 2, 3, 4, 49, 100, 49, 43, 49};
       vector <int> v(A, A + 10);
       vector <int>::iterator it = v.begin();
       while ((it = find(it, v.end(), 49)) != v.end())
       {
             cout << it - v.begin() << '\t' << *it << endl;
// Передвигаем указатель на следующую позицию и снова выполняем поиск
              ++it;
       }
       return 0;
}
```

Результат:

```
1       49
5       49
7       49
9       49
Для продолжения нажмите любую клавишу . . .
```

Алгоритм *find_if*

Алгоритм find_if является более общим по сравнению с алгоритмом find, поскольку одним из параметов find_if является предикат. Этот алгоритм ищет *первое вхождение* элемента, обращающего предикат в true.

Рассмотрим пример. Необходимо в списке найти элемент a, входящий в диапазон 10 < a < 20 и являющийся чётным числом.

```
#include <list>
#include <algorithm>
bool range_find_if(int x)
{
      // Предикат применяется в алгоритме list_find_if
      return (10 < x && x < 20) && (x % 2 == 0);
}
int list_find_if()
{
      // Найти в списке значение, принадлежащее диапазону чисел
```

```
    // 10 < a < 20 и являющееся чётным. Используется предикат range_find_if
    int a[6] = { 5, 18, 12, 14, 19, 24 };

        int j;
        list<int> lst(a, a + 6);
        list<int>::iterator i;
        for (i = lst.begin(); i != lst.end(); ++i)
              cout << *i << " ";
        cout << endl;
        i = find_if(lst.begin(), lst.end(), range_find_if);
        if (i != lst.end())
              cout << "Запрашиваемый элемент найден: " << *i << endl;
        else
              cout << "Запрашиваемый элемент не найден: " << endl;
        return  0;
}
```

Результат:

```
5 18 12 14 19 24
Запрашиваемый элемент найден: 18
Для продолжения нажмите любую клавишу . . .
```

В STL существует несколько алгоритмов, имеющих название, включающее _if, такие же как `find и find_if`, а значит, использующие предикаты. Например, `remove` и `remove_if`, `replace` и `replace_if`, `count` и `count_if` и другие.

Алгоритм *copy*

При копировании рассмотрим режимы замещения и вставки. В режиме вставки нам понадобится *итератор вставки.* Для его использования необходимо в проект добавить заголовочный файл **`#include <iterator>.`**

Чтобы ввести данные в контейнер, не обязательно применять методы вставки, можно воспользоваться инициализацией контейнеров. Хорошей основой для инициализации контейнеров `vector, deque` и `list` может послужить массив. Например:

`int a[3] = {5, 9, 17};` – инициализация массива
`vector<int> v(a, a+3);` – инициализация вектора
`deque<int> w(a, a+3);` – инициализация очереди
`list<int> x(a, a+3);` – инициализация списка

Этот приём будет использоваться в программах, приведенных ниже.

Но не только массив, а также `vector, deque` и `list` могут служить основой для инициализации контейнера того же типа. Например, проинициализируем вектор `v1` с помощью данных вектора `v`:

```
vector<int> v1(v.begin(),  v.end());
```

Рассмотрим две программы с использованием алгоритма `copy:` в режиме замещения и в режиме вставки.

```
#include <deque>
#include <list>
#include <algorithm>
#include <iterator>
int copy_deque_list1()
{   //  Копируем очередь в список:  режим замещения.
      int a[5] = { 5, 8, 12, 36, 9 };
      deque<int> deq (a, a + 5);
      deque<int>::iterator iq;
      cout << "Значения в очереди: ";
      for (iq = deq.begin(); iq != deq.end(); ++iq)

            cout << *iq << "  ";
        cout << endl;
        int b[5] = { 2, 3, 4, 5, 6 };
        list<int> lst(b, b + 5); // Список из 5 элементов
        list<int>::iterator il;
        cout << "Значения в списке:  ";
        for (il = lst.begin(); il != lst.end(); ++il)
              cout << *il << "  ";
        cout << endl;
        cout << "Копируем в режиме замещения: очередь -> список" << endl;
        copy(deq.begin(), deq.end(), lst.begin());
        cout << "Значения в списке:  ";
        for (il = lst.begin(); il != lst.end(); ++il)
              cout << *il << "  ";
        cout << endl;
        return  0;
  }
```

Результат:

```
Значения в очереди: 5  8  12  36  9
Значения в списке:  2  3  4  5  6
Копируем в режиме замещения: очередь -> список
Значения в списке:  5  8  12  36  9
Для продолжения нажмите любую клавишу . . .
```

При копировании в режиме вставки заменим вызов алгоритма `copy`: `copy(deq.begin(), deq.end(), lst.begin());` на следующий:
`copy(deq.begin(), deq.end(), inserter(lst, il));`

В данном случае применяется специальный вид адаптеров итераторов `inserter` – итератор вставки. Существует 3 вида итераторов вставки `front_inserter, back_inserter и inserter.` Первый добавляет элементы в начало контейнера, второй – в конец и третий в произвольное место, которое указывается параметрами итератора.

В рассматриваемом случае `inserter(lst, il)` – вставка производится в список `lst`, начиная с `il`-й позиции, считая от `0.`

```
int copy_deque_list2()
{   //  Копируем очередь в список: режим вставки.
      int a[5] = { 5, 8, 12, 36, 9 };
      deque<int> deq(a, a + 5);
      deque<int>::iterator iq;
      cout << "Значения в очереди: ";
      for (iq = deq.begin(); iq != deq.end(); ++iq)
            cout << *iq << "  ";
      cout << endl;
      int b[5] = { 1, 3, 6, 7, 11 };
      list<int> lst(b, b + 5); // Список из 5 элементов
      list<int>::iterator il;
      cout << "Значения в списке:  ";
      for (il = lst.begin(); il != lst.end(); ++il)
            cout << *il << "  ";
      cout << endl;
      cout << "Копируем в режиме вставки: очередь -> список" << endl;
      il = lst.begin();
      advance(il, 3); // приращение итератора
      copy(deq.begin(), deq.end(), inserter(lst, il));
      cout << "Значения в списке:  ";

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

Результат (il = 3):

```
Значения в очереди: 5  8  12  36  9
Значения в списке:  1  3  6  7  11
Копируем в режиме вставки: очередь -> список
Значения в списке:  1  3  6  5  8  12  36  9  7  11
Для продолжения нажмите любую клавишу . . .
```

Задания для самостоятельного выполнения

1. Создать очередь. Ввести переменное количество ненулевых целых (ввод завершается нулем) и вывести их на экран. К данным применить функции вставки и удаления для выполнения следующих действий (после каждого действия следует вывод на экран):
    a. Удалить 4-й элемент.
    b. Вставить числа 11 и 22 в начало очереди.
    c. Перед последним элементом очереди вставить 99.
    d. Вставить в конец очереди число 88.
    e. Отсортировать.
    f. Задать искомый элемент и найти его в очереди.
2. Создать вторую очередь, ввести в неё 5 любых чисел. Вывести их на экран.
3. Используя итератор вставки, скопировать вторую очередь в первую после 4-го элемента. Вывести полученный результат на экран.
4. Написать программу нахождения заданного значения в массиве, используя алгоритм find.
5. Написать программу, формирующую по заданному списку целых чисел вектор, состоящий из элементов списка с чётными значениями. Список и вектор вывести на экран.

Контрольные вопросы

1. Дайте определение последовательному контейнеру.
2. Какие виды последовательных контейнеров вы знаете?
3. Какие заголовочные файлы нужно включить в проект для работы с последовательными контейнерами?
4. Что такое итератор?
5. Какие вы знаете методы, применяемые в контейнерах?
6. Объясните особенности методов `begin()` и `end()`.
7. Для каких целей в программах используются следующие конструкции: `#include <list>, list <int> L; list <int>::iterator iL;`?
8. В какие позиции производится вставка и удаление в контейнерах `vector, deque` и `list` по умолчанию?
9. Чем отличаются методы шаблонных классов от алгоритмов?
10. Алгоритм `sort`. Его использование и особенности.
11. Алгоритм `find`. Его использование и особенности.
12. Как с помощью алгоритма `find` найти все вхождения искомого элемента?
13. Алгоритм `copy`. Режимы замещения и вставки.
14. В каких случаях используются итераторы вставки?
15. Каким образом предикат в вызове `find_if` влияет на результат поиска?

3. Операция объединения. Итераторы

Для объединения данных применяется алгоритм `merge`. Данные могут быть расположены в контейнерах разных типов, особенностью алгоритма является упорядоченность данных в исходных контейнерах.

Алгоритм merge

Алгоритм `merge` можно использовать для каждого из трёх типов последовательных контейнеров (векторы, двусторонние очереди, списки), также и для массивов. Контейнеры не обязательно должны принадлежать к одному и тому же контейнерному типу.

Каждая из объединяемых последовательностей должна быть упорядочена, иначе алгоритм не будет работать правильно.

Рассмотрим программу объединения вектора и списка в очередь. Пусть тип данных будет double. Вводим данные в контейнеры в произвольном порядке, но перед объединением их необходимо отсортировать. Для вектора используем алгоритм sort, а для списка – метод sort(). Для вывода данных каждого контейнера создаём итератор, соответствующий контейнеру. Для алгоритма `merge` применяем итератор вставки `inserter`.

```
#include <vector>
#include <list>
#include <queue>
#include <algorithm>
#include <iterator>
double vector_and_list_to_deque()
{
	vector <double> vec;
	double x;
	cout << "Введите веществ. числа, за которыми следует 0.0: \n";
	while (cin >> x, x != 0.0)  // Ввод данных
		vec.push_back(x);
	vector<double>::iterator i;    // Итератор для вектора
	sort(vec.begin(), vec.end());  // Сортировка вектора
	cout << "Отсортированный вектор: " << endl;
	for (i = vec.begin(); i != vec.end(); ++i)
		cout << *i << " ";
	cout << endl << endl;

	list <double> lst;
	double y;
	int count = 0;
	cout << "Введите 5 чисел: \n";
	while (cin >> y)
	{
		lst.push_back(y);
		count++;
		if (count == 5)
			break;
	}
	lst.sort();    // Сортировка списка
	cout << "Отсортированный список: " << endl;
	list<double>::iterator il;    // Итератор для списка
	for (il = lst.begin(); il != lst.end(); ++il)
		cout << *il << " ";
	cout << endl;
```

```
        deque <double> Q;
        cout << endl << "Объединение вектора и списка в очередь: " << endl;
        merge(vec.begin(), vec.end(), lst.begin(), lst.end(),
              inserter(Q, Q.begin()));
        deque <double> ::iterator iQ;    // Итератор для очереди
        for (iQ = Q.begin(); iQ != Q.end(); ++iQ)
             cout << *iQ << " ";
        cout << endl;
        return 0.0;
  }
```

Результат:

```
Введите веществ. числа, за которыми следует 0.0:
4.9 2.3 9.1 -18.3 7.4 16.5 0.0
Отсортированный вектор:
-18.3 2.3 4.9 7.4 9.1 16.5

Введите 5 чисел:
2.4 5.6 6.2 -10.1 43.7
Отсортированный список:
-10.1 2.4 5.6 6.2 43.7

Объединение вектора и списка в очередь:
-18.3 -10.1 2.3 2.4 4.9 5.6 6.2 7.4 9.1 16.5 43.7
Для продолжения нажмите любую клавишу . . .
```

В качестве альтернативы использования итератора вставки можно выделить достаточно места для очереди `Q` (в этом случае нужно знать, сколько элементов будет введено в каждый контейнер):

`deque <double> Q (11)` // принимает `6 + 5 = 11` элементов
`merge(vec.begin(), vec.end(), lst.begin(), lst.end(), Q.begin());`
Итератор вставки уже встречался нам при описании алгоритма `copy`.

Типы, определенные пользователем. Оператор сравнения operator<

Чтобы применить другие возможности алгоритма `merge`, рассмотрим типы данных, определённые пользователем.

Так как реализация `merge(...)` в программе `vector_and_list_to_deque()` основана на операции сравнения «меньше чем» **<** (в данном случае сравнивали вещественные числа в двух контейнерах), вызов `merge(...)` для новых типов возможен, только если для этих типов определен **`operator<`**, *который будет сравнивать значения заданных типов в двух контейнерах.*

Рассмотрим структуру, содержащую 3 поля: год, месяц и название. В структуру входит `operator<`, в котором описывается упорядочивание записей по двум полям: по году и месяцу. Алгоритм этого упорядочивания (от меньшего к большему) несложный:

пусть `y1` – год в первой записи, `m1` – месяц в первой записи,
`y2` – год во второй записи, `m2` – месяц во второй записи:

`if (y1 < y2) return true;` – первая запись впереди,
`else if (y1 > y2) return false;` – вторая запись впереди,
`else return m1 < m2;` – если годы совпадают, то сравниваем месяцы, первая запись впереди.

```
struct item
{
      long year;
      long month;
```

```
        char name[60];
        bool operator< (const item &b) const
        {
              // Сравнение двух записей по году и месяцу
              // для программы merge_item()
              {
                    if (year < b.year) return true;
                    else if (year > b.year) return false;
                    else return month < b.month;
              }
        }
  };
```

Функция, принимающая логические значения, называется предикатом. Функция **`bool operator<()`** принимает значения `true` или `false` в зависимости от тела функции. В данном примере – это упорядочение по году и месяцу при объединении записей.

Создадим 3 массива структур `a[3], b[2]` и `f[3],` заполним их данными. Данные в каждом массиве должны быть упорядочены по году и месяцу. Поскольку алгоритм `merge` объединяет два контейнера, будем сначала объединять `a[3]` и `b[2]` в `c[5],` а затем `f[3]` и `c[5]` в `g[8].` Так как работаем с массивами, определим указатель `*p` типа `item` (название структуры). С помощью этого указателя будем выводить результат объединения на экран.

```
int merge_item()
{
// Объединяем записи, используя год и месяц в качестве ключей
// Для сравнения применяется предикат bool operator< (const item &b) const
      item a[3] =
      {
            { 2011,5,"GPU"},
            { 2013,9,"Keyboard"},
            { 2019,12,"Mouse"}
      },
            b[2] =
      {
            { 2013,10,"Motherboard"},
            { 2014,12,"PowerSupply"}
      },
            f[3] =
      {
            { 2013,5,"Flash_drive"},
            { 2014,9,"Monitor"},
            { 2020,6,"Mousepad"}
      },
            c[5], g[8], *p;
      merge(a, a + 3, b, b + 2, c);
      cout << "By year + month: \n";
      for (p = c; p != c + 5; ++p)
            cout << p->year << " " << p->month << " " << p->name << endl;
      cout << endl;
      merge(f, f + 3, c, c + 5, g);
      for (p = g; p != g + 8; ++p)
            cout << p->year << " " << p->month << " " << p->name << endl;
      cout << endl;

      return 0;
```

Результат:
Первое объединение ((a и b) в c):

```
By year + month:
2011 5 GPU
2013 9 Keyboard
2013 10 Motherboard
2014 12 PowerSupply
2019 12 Mouse
```

Второе объединение ((f и c) в g):

```
2011 5 GPU
2013 5 Flash_drive
2013 9 Keyboard
2013 10 Motherboard
2014 9 Monitor
2014 12 PowerSupply
2019 12 Mouse
2020 6 Mousepad
```

В программе `merge_item()` в качестве ключей объединения применялить годы и месяцы. Решим другую задачу: объединить записи массивов структуры `item` по названию (`name`) в качестве ключа. Для этого нужно определить `operator<` заново, но программа `merge_item()` *останется прежней.* Структура `item` описывается следующим образом (сравниваются строки).

```
struct item
{
	long year;
	long month;
	char name[60];
	bool operator< (const item& b) const
	{
		return strcmp(name, b.name) < 0;
	}
};
```

Функция `strcmp(string1, string2)` сравнивает строки `string1` и `string2` лексикографически и возвращает значение:

`<0` – `string1` меньше, чем `string2`,
`0` – `string1` совпадает с `string2`,
`>0` – `string1` больше, чем `string2`.

Результат:
Первое объединение ((a и b) в c):

```
By name:
2011 5 GPU
2013 9 Keyboard
2013 10 Motherboard
2019 12 Mouse
2014 12 PowerSupply
```

Второе объединение ((f и c) в g):

```
2013 5 Flash_drive
2011 5 GPU
2013 9 Keyboard
2014 9 Monitor
2013 10 Motherboard
2019 12 Mouse
2020 6 Mousepad
2014 12 PowerSupply
```

Из последних двух примеров видно, что упорядочение при объединении с помощью алгоритма `merge` зависимосит от того, как был запрограммирован предикат `operator<` в структуре `item.`

Категории итераторов

Для того чтобы глубже понять работу библиотеки STL, необходимо познакомиться с различными категориями итераторов. Если рассмотреть разные алгоритмы, то станет понятно, что, например, для последовательного поиска достаточно только просмотра «вперёд», а для сортировки необходимы различные перестановки, т.е. движения как «вперёд» по последовательности, так и «назад». Все эти передвижения по последовательности или по множеству осуществляются с помощью различных категорий итераторов.

В случае сортировки (алгоритм `sort`) или последовательного поиска (алгоритм `find`) используются итераторы, но алгоритм `sort` требует «более мощных» итераторов, нежели алгоритм `find.` Итераторы можно поделить на пять категорий, в соответствии с теми операциями, которые для них определены.

Пусть `i` и `j` – итераторы одного вида.

Тогда, с любым итератором могут быть выполнены следующие основные операции:

- Разыменование итератора; если `i` – итератор, то `*i` – значение объекта, на который он ссылается.
- Присваивание одного итератора другому: `i = j.`
- Сравнение итераторов: `i == j, i != j.`
- Перемещение итератора по всем элементам контейнера с помощью префиксного `(++i)` или постфиксного `(i++)` инкремента.

Реализация итератора различна для каждого контейнера, поэтому при объявлении итераторов указывается тип контейнера, например:

```
vector<int> v;
vector<int>::iterator iv;
```

Другим универсальным способом описания любой переменной в C++ служит описание переменной через присваивание ей начального значения. Ключевое слово **`auto`** *позволяет не описывать тип переменной явно* (в конкретном случае `iv`), а брать её тип из значения справа: `auto iv = v.begin();`

В дальнейшем будем широко пользоваться этим способом.

Существуют 5 основных категорий итераторов:

- входные `(input)` итераторы используются для чтения значений из контейнера (например, алгоритм `find);`
- выходные `(output)` итераторы используются для записи значений в контейнер (например, вывод данных в поток `cout` – алгоитм `copy`);
- прямые `(forward)` итераторы осуществляют передвижения по контейнеру только в прямом направлении (например, алгоритм `replace`, заменяющий одно определенное значение на другое);

– двунаправленные (bidirectional) итераторы поддерживают навигацию как в прямом, так в обратном направлениях (например, алгоритм reverse, выполняющий реверсию данных контейнера);
– итераторы произвольного доступа (random access) – самые «умные» из всех основных итераторов, используются для сложных алгоритмов (например, в алгоритме sort).

Контейнеры vector и deque основываются на итераторах произвольного доступа.

Так как список не предоставляет итераторов произвольного доступа, нельзя применить к списку алгоритм sort. В классе list реализован собственный метод sort().

Потоковые итераторы

Алгоритм copy можно применять для вывода на экран и ввода данных с клавиатуры. При этом используются потоковые итераторы ostream_iterator и istream_iterator.

Например:

```
#include <iterator>
int mas[5] = { 8, 3, 15, 29, 1 };
copy(mas, mas + 5, ostream_iterator<int>(cout, " "));
```

Это можно сделать и таким образом:

```
ostream_iterator <int> ocout (cout, " ");
copy (mas, mas +5, ocout); // потоковый итератор – 3-й аргумент.
```

В поток стандартного вывода cout будут выведены числа 8, 3, 15, 29, 1, как если бы написали:

```
for (int* p= mas; p!= mas +5; ++p)
cout << *p << " ";
```

Например, для очереди потоковый итератор можно записать следующим образом:

```
deque<int> Q;
copy(Q.begin(),Q.end(), ostream_iterator<int>(cout, " "));
```

Для ввода используется аналогичный прием:

```
istream_iterator<int> icin (cin);
```

Рассмотрим небольшую программу, которая читает из стандартного потока cin числа, вводимые пользователем, и дублирует их на экране. Работа программы заканчивается, при вводе числа 77.

```
#include <iterator>
void io_iterator()
{
   // Использование потоковых итераторов
      istream_iterator<int> icin (cin);
      ostream_iterator<int> ocout (cout, "  ");
      int k;
      while ((k = *icin) != 77)
      {
            *ocout++ = k;
            icin++;
      }
      cout << endl;
}
```

Результат:

```
15 32 65 91 30 77
15  32  65  91  30
Для продолжения нажмите любую клавишу . . .
```

Первая строка – ввод с клавиатуры, вторая – вывод на экран.

Дополнительный материал по итераторам и алгоритмам предоставляется студентам в электронном виде, также можно посмотреть в приведённой литературе.

Задания для самостоятельного выполнения

1. Написать программу объединения очереди и вектора в списк. Пусть тип данных будет `int`. Для инициализации очереди и вектора используйте массивы.
2. Создать структуру, описывающую покупку товара, поля структуры:
   - год покупки – целое число,
   - месяц покупки – целое число,
   - стоимость – вещественное число,
   - название товара – строка.

   Ввести данные в три массива структур с различными именами. Количество записей в каждом массиве 3-4. Объединить записи, используя:
   a. названия товаров в качестве ключа,
   b. год и месяц в качестве ключа с двумя полями.
   c. стоимость товара в качестве ключа.

   Результаты вывести на экран.
3. Написать программу, добавляющую в конец списка вещественных чисел элемент, значение которого равно среднему арифметическому всех его элементов. Результат вывести на экран.
4. Написать программу, формирующую по заданному вектору целых чисел список из элементов вектора с четными числами и вывести список на экран.

Контрольные вопросы

1. Объясните работу алгоритма `merge.`
2. Для каких контейнеров может применяться алгоритм `merge?`
3. Для каких целей в заголовочные файлы включается строка `#include <iterator>?`
4. Каким образом используется алгоритм `merge` в типах, определённых пользователем?
5. Что означает `operator<?`
6. Как можно записать `operator<` для упорядочивания строк / чисел / двух полей (например, год и месяц)?
7. Какие значения возвращает функция `strcmp(string1, string2)?`
8. Какие вы знаете категории итераторов?
9. Приведите примеры алгоритмов с соответствующими категориями итераторов.
10. Объясните работу потоковых итераторов.

4. Функциональные объекты. Оператор вызова operator()

Функциональный объект (функтор) обобщает понятие функции. Целью данного раздела является знакомство с понятием функционального объекта, изучение функциональных объектов на приведённых примерах и применение полученных знаний.

Определение функционального объекта и его возможности

На примере работы алгоритма `merge` было показано, что объединение записей в соответствии с этим алгоритмом полностью зависит от того, по каким ключам задаётся упорядочение в операторе сравнения `operator<.`

Теперь рассмотрим функциональные объекты, т.е. объекты, для которых определён оператор вызова **`operator().`** Все алгоритмы в библиотеке STL выполняют заданные определённые действия. Однако, в алгоритмы можно добавить свои способы сравнения объектов или их обработки, используя `operator().`

Функциональным объектом называется класс, в котором определен оператор вызова. От класса не требуется наличия каких-либо других членов.

При работе с функторами применяется заголовочный файл:
`#include <functional>,` но иногда он не требует явного указания, поскольку входит в другие заголовочные файлы.

Функциональные объекты, которые возвращают булевский тип, называются предикатами.

Рассмотрим алгоритм `sort,` работающий по умолчанию, т.е. стандартным образом, а именно – упорядочивание чисел в восходящем порядке (от меньшего к большему). Чтобы упорядочить числа в нисходящем порядке, можно:

1. написать предикат и использовать его как параметр при обращении к алгоритму sort;
2. написать функциональный объект `compare` и применять его в качестве параметра в алгоритме `sort.`

`1.` Напишем функцию сравнения и применим её в алгоритме `sort.`

```
// dsort1.cpp: Сортировка в нисходящем порядке
// с использованием функции сравнения
bool comp1(int x, int y)   //Функции сравнения
{
      return x > y;
}

int dsort1()
{
      const  int N = 8;
      int a[N] = { 21, 37, 44, 13, 96, 58, 20, 73 };
      cout << "Перед сортировкой: \n";
      //Вывод производится через потоковые итераторы
      copy(a, a + N, ostream_iterator<int>(cout, " "));
      cout << endl;
      sort(a, a + N, comp1);  //В sort включили функцию сравнения
      cout << "После сортировки в нисходящем порядке:\n";
      copy(a, a + N, ostream_iterator<int>(cout, " "));
      cout << endl;
      return  0;
}
```

Результат:

```
Перед сортировкой:
21 37 44 13 96 58 20 73
После сортировки в нисходящем порядке:
96 73 58 44 37 21 20 13
Для продолжения нажмите любую клавишу . . .
```

2. Напишем функтор и применим его к алгоритму `sort`.

```
//  dsort2.cpp: Сортировка в нисходящем порядке с использованием
//  определённого функционального объекта
class  compare   // Функциональный объект compare, содержащий operator()
{
public:
      bool operator()(int x, int y)const
      {
            return x > y;
      }
};

int dsort2()
{
      const int N = 8;
      int a[N] = { 21, 37, 44, 13, 96, 58, 20, 73 };
      cout << "Перед сортировкой:\n";
      // Вывод производится через потоковые итераторы
      copy(a, a + N, ostream_iterator<int>(cout, " "));
      cout << endl;
      sort(a, a + N, compare()); // В sort включили функциональный объект
      cout << "После сортировки в нисходящем порядке:\n";
      copy(a, a + N, ostream_iterator<int>(cout, " "));
      cout << endl;
      return 0;
}
```

Результат:

```
Перед сортировкой:
21 37 44 13 96 58 20 73
После сортировки в нисходящем порядке:
96 73 58 44 37 21 20 13
Для продолжения нажмите любую клавишу . . .
```

На примерах программ `dsort1()` и `dsort2()` видно, что можно модифицировать алгоритм (в частности `sort`), применяя третий параметр, который указывает либо на функцию сравнения, либо – на функциональный объект.

Напишем более сложную программу – целочисленный калькулятор для операций: `+, -, *, /, %` и для выхода `#`. Для решения создадим функциональный объект `calc`. В классе `calc` используется `operator()`, содержащий 4 параметра.

`operator()(int x, char oper, int y, int& ok)`:

– первый операнд `int x`,
– знак операции `char oper`,
– второй операнд `int y`,

– параметр, который передаёт состояние при выполнении операций (передача параметра по ссылке) `int& ok`:
  - `1` – нормальный выход, все операции работают правильно,
  - `-1` – деление на `0`,
  - `-2` – неверная операция.

Выход из программы – только при вводе `#`.

```
class calc  // Функциональный объект
{
public:
      int operator()(int x, char oper, int y, int& ok) const
      {
            int res;
            switch (oper) // Анализ знака операции
            {
            ok = 1;
            case '+': {res = x + y; break; }
            case '-': {res = x - y; break; }
            case '*': {res = x * y; break; }
            case '/':
                  if (y) res = x / y; // Знаменатель y != 0
                  else
                  {
                        ok = -1; res = 0;
                  }
                  break;
            case '%': {res = x % y; break; }
            default:
                  ok = -2;
                  cout << "Неверная операция" << endl;
                  res = 0;
            }
            return res;
      }
};
```

Функциональный объект применим в программе `calculate()`.

```
void calculate()
{
      int a, b, c;
      char oper;
      int ok(1); // Первоначально ok = 1
      cout << "            Целочисленный калькулятор";
      while (ok) // Цикл будет работать, пока ok = 1
      {
            cout << "\n  Введите знак операции (+, -, *, /, %, для окончания #): ";
            cin >> oper;
            if (oper == '#') break; // Выход из программы при вводе #
            // При неверной операции возвращаемся на начало цикла - continue
            if (oper != '+' && oper != '-' && oper != '*' && oper != '/' &&
                  oper != '%' )
                  { cout << "Error operation" << endl; ok = 1; continue; }
            cout << "Введите 1-й операнд: ";
            cin >> a;
```

```
            cout << "Введите 2-й операнд: ";
            cin >> b;
            calc example; // Создаём экземпляр функционального объекта
            c = example(a, oper, b, ok);
        // Анализируем значение ok: 1, -2, -1 и возвращаемся на начало цикла
            if (ok == 1)
                cout << "     " << a << oper << b << "=" << c << endl;
            else
            {
               if (ok == -2) {cout << "Error operation" << endl; ok = 1;}
               else
               {
                  if (ok == -1) {cout << "Error divide" << endl; ok = 1;}
               }
            }
        }
   }
```

Результат:

```
         Целочисленный калькулятор
  Введите знак операции (+, -, *, /, %, для окончания #): +
Введите 1-й операнд: 6
Введите 2-й операнд: 7
     6+7=13

  Введите знак операции (+, -, *, /, %, для окончания #): /
Введите 1-й операнд: 8
Введите 2-й операнд: 0
Error divide
  Введите знак операции (+, -, *, /, %, для окончания #): =
Error operation

  Введите знак операции (+, -, *, /, %, для окончания #): +
Введите 1-й операнд: 12
Введите 2-й операнд: 5
     12+5=17

  Введите знак операции (+, -, *, /, %, для окончания #): %
Введите 1-й операнд: 8
Введите 2-й операнд: 5
     8%5=3
  Введите знак операции (+, -, *, /, %, для окончания #): -
Введите 1-й операнд: 9
Введите 2-й операнд: 3
     9-3=6

  Введите знак операции (+, -, *, /, %, для окончания #): #
Для продолжения нажмите любую клавишу . . .
```

Функциональные объекты, определенные в STL

В библиотеке STL определены несколько встроенных функциональных объектов, соответствующих арифметическим и логическим операциям. Доступ к этим объектам возможен при включении заголовочного файла: **`#include <functional>.`** Смысл операции соответствует названию.

Арифметические операции			
plus	x+y	minus	x-y
multiplies	x*y	divides	x/y
modulus	x%y	negate	-x
Сравнения			
equal_to	x==y	not_equal_to	x!=y
greater	x>y	less	x<y
greater_equal	x>=y	less_equal	x<=y
Логические операции			
logical_and	x&&y	logical_or	x||y
logical_not	!x		

Эти функциональные объекты можно использовать в различных алгоритмах, например:

```
sort (a, a+N, greater<int> ());
```

Способы модификации алгоритмов

Подведём итог использования алгоритмов в STL с включением предикатов и функциональных объектов. Будем рассматривать алгоритм `sort` для vector<int> v.

1. Применяем стандартный алгоритм `sort`:
   `sort (v.begin (), v.end());` – сортировка от меньшего к большему.
2. Затем используем этот алгоритм с предикатом `comparefun()`, для сортировки от большего к меньшему:

```
bool comparefun(int x, int y)
{   return x > y;
}
sort (v.begin (), v.end(), comparefun);
```

3. Используем алгоритм `sort` с функциональным объектом `compare()` – сортировка в убывающем порядке:

```
class compare
{  public:
bool operator() (int x, int y) const
      {   return x > y; }
};
sort (v.begin (), v.end(), compare());
```

4. Используем алгоритм `sort` с функциональным объектом `greater<int>()`, определённым в STL – сортировка в убывающем порядке:

```
sort (v.begin (), v.end(), greater<int>());
```

В дальнейшем также будем применять указанные способы и для других алгоритмов, тем самым усиливая их возможности.

Задания для самостоятельного выполнения

1. Задан вектор из 10 элементов, проинициализируйте его с помощью данных массива `int a[] = {11, 26, 79, 11, 55, 31, 59, 18, 20, 143};`
   a. С помощью алгоритма `find` найдите элемент `55`;
   b. С помощью алгоритма `find_if` найти *первый элемент*, удовлетворяющий условию `10<x<50`;
   c. С помощью алгоритма `find_if` найти *все элементы*, удовлетворяющие условию `10<x<50`;

Для заданий b и c нужно написать функциональный объект, определяющий, входит ли очередной элемент вектора в диапазон (10, 50). Результат всех заданий вывести на экран.

Алгоритмы find и find_if по умолчанию находят только первый элемент.

2. Заполнить массив размером 10 случайными числами от 0 до 10, отсортировать его по убыванию и вывести на экран. Используйте функциональный объект STL greater<int>().
3. Создать очередь, внести в неё 10 целых чисел (как положительных, так и отрицательных). Написать функциональный объект, который, на основе алгоритма sort, будет упорядочивать данные из очереди по возрастанию их абсолютных величин.

Контрольные вопросы

1. Что такое функциональный объект?
2. Для каких целей служит функциональный объект?
3. Какой заголовочный файл нужно подключить, чтобы использовать функциональные объекты, определённые в STL?
4. Что такое предикат?
5. Каким образом в программе задаётся передача параметра по ссылке?
6. Объясните на примере программы калькулятора как можно работать с параметром, который передаётся по ссылке.
7. Какие вы знаете функциональные объекты, определённые в STL? Приведите примеры использования.
8. Чем отличается применение в алгоритмах предикатов и функциональных объектов?
9. Какие вы знаете способы модифицирования алгоритмов в STL?
10. Чем отличаются операторы operator< и operator()?

5. Работа со строками и численный алгоритм accumulate

Рассмотрим возможности некоторых алгоритмов работы со строками и вычислительного алгоритма `accumulate`. Работа алгоритмов иллюстрируется примерами, обучающими применению полученных знаний.

При работе со строками используется заголовочный файл **<algorithm>,** для алгоритма `accumulate` применяется заголовочный файл **<numeric>.**

Работа со строками

Рассмотрим несколько алгоритмов работы со строками:

- `replace` – замена символов в строке;
- `reverse` – замена строки на обратную ей;
- `count` и `count_if` – подсчёт количества символов строки, в том числе при заданном условии;

Разберём их работу.

Алгоритмы *replace, reverse, count и count_if*

Рассмотрим примеры использования указанных алгоритмов.

```
#include <algorithm>
#include <string>
```

Для работы алгоритма `count_if` нам потребуются предикаты:

```
bool found(char ch) // Предикат для работы алгоритма count_if
{
	return ch == 'а' || ch == 'д';
}
bool found1(char ch) // Предикат для работы алгоритма count_if,
// для подсчёта гласных русских букв
{
	return ch == 'а' || ch == 'е' || ch == 'и' || ch == 'о' || ch == 'у' ||
		ch == 'ы' || ch == 'э' || ch == 'э' || ch == 'я';
}
```

Демонстрация работы со строками:

```
void sequences()
{
	setlocale(LC_ALL, "Russian");
	char str[] = "абвгдавгвбда";
	// Заменить символы в строке
	int n = strlen(str);  // Определение количества символов в строке
	cout << str << endl;
	replace(str, str + n, 'в', '1');
	cout << "replace в -> 1: " << str << endl << endl;
```

```
Результат:
абвгдавгвбда
replace в -> 1: аб1гда1г1бда
```

```
	// Заменить строку на обратную ей
	cout << str << endl;
	reverse(str, str + strlen(str));
	cout << "reverse: " << str << endl << endl;
```

```
Результат:
аб1гда1г1бда
reverse: адб1г1адг1ба
```

```
        // Подсчитать количество элементов 'а'
        cout << str << endl;
        int k = count(str, str + n, 'а');
        cout << "count 'а' : " << k << endl << endl;
```

Результат:

```
адб1г1адг1ба
count 'а' : 3
```

```
        // Подсчитать количество элементов 'а' и 'д'
        // Воспользуемся предикатом found(char ch)
        cout << str << endl;
        k = count_if(str, str + n, found);
        cout << "count_if 'а' и 'д' : " << k << endl << endl;
```

Результат:

```
адб1г1адг1ба
count_if 'а' и 'д' : 5
```

```
        // Подсчитать количество всех русских гласных букв
        // (а, е, и, о, у, ы, э, ю, я) в предложении (1 вариант)
        char* p =
        "STL была первоначально разработана сотрудниками Hewlett-Packard",
              * q = p + strlen(p);
        n = count(p, q, 'а') + count(p, q, 'е') +
            count(p, q, 'и') + count(p, q, 'о') + count(p, q, 'у') +
            count(p, q, 'ы') + count(p, q, 'э') + count(p, q, 'ю') +
            count(p, q, 'я');
cout << n << " - количество найденных гласных (1 вариант). " << endl << endl;
```

Результат:

```
17 - количество найденных гласных (1 вариант).
```

```
        // Подсчитать количество всех русских гласных букв
        // (а, е, и, о, у, ы, э, ю, я) в предложении (2 вариант)
        // Воспользуемся предикатом found1(char ch)
         n = count_if(p, q, found1);
         cout << n << " - количество найденных гласных (2 вариант). " << endl
```

Результат:

```
17 - количество найденных гласных (2 вариант).
}
```

Использование предиката `found1` в алгоритме `count_if` позволило просматривать тестируемое предложение всего лишь один раз, в отличие от алгоритма `count`, который для той же самой задачи пришлось вызывать много раз.

Рассмотрим ещё одну задачу (2 решения):

Ввести в массив `str[]` произвольную последовательность из `10` символов, содержащую 3 буквы `‘а’`, в массив `str1[]` передать строку `str[]` в обратном порядке, а в массив `str2[]` – строку из `str[]`, в которой буквы `‘а’` заменены на `‘1’`.

При работе с символьными строками конец символьной строки представляется с помощью символа `NULL`, который в `C++` изображается как символ `‘\0’`. Из этого следует, что, для размещения символа конца строки, длина массива должна быть увеличена на `1`.

1-е решение:

```
void massiv_str()
{
	setlocale(LC_ALL, "Russian");
	char str[] = "asdafgsdas";  // В строке 10 символов
	//Строка заканчивается ‘\0’, поэтому нужно задавать
	//размер массивов str1[11], str2[11]
	char str1[11], str2[11];
	cout << "Строка str:  " << str << endl;
	for (int i = 0; i < 11; ++i)
	{
		str1[i] = str[i];
		str2[i] = str[i];
	}
	cout << "Строка str1: " << str1 << endl;
	cout << "Строка str2: " << str2 << endl << endl;
	reverse(str1, str1 + strlen(str1));
	replace(str2, str2 + strlen(str2), 'a', '1');
	cout << "reverse str1: " << str1 << endl;
	cout << "replace str2: " << str2 << endl;
}
```

Результат:

```
Строка str:  asdafgsdas
Строка str1: asdafgsdas
Строка str2: asdafgsdas

reverse str1: sadsgfadsa
replace str2: 1sd1fgsd1s
Для продолжения нажмите любую клавишу . . .
```

2-е решение:

```
void massiv1_str()
{	setlocale(LC_ALL, "Russian");
	char str[] = "asdafgsdas";
	// Применим функцию size() для определения длины массива
	cout << "Длина  массива str: " << size(str) << endl;
	// Применим функцию strlen(str) для определения длины строки
	int n = strlen(str);  // Длина строки
	cout << "Длина строки str: " << n << endl;
	// Под массивы str1 и str2 выделяем память динамически
	char* str1 = new char[n+1];
	char* str2 = new char[n+1];
	strcpy_s(str1, n+1, str);
	strcpy_s(str2, n+1, str);
	cout << "Строка str1: " << str1 << endl;
	cout << "Строка str2: " << str2 << endl << endl;
	reverse(str1, str1 + strlen(str1));
	replace(str2, str2 + strlen(str2), 'a', '1');
	cout << "reverse str1: " << str1 << endl;
	cout << "replace str2: " << str2 << endl;
	// Освобождаем память
	delete[] str1;
	delete[] str2;
}
```

Результат:

```
Длина  массива str: 11
Длина строки str: 10
Строка str1: asdafgsdas
Строка str2: asdafgsdas

reverse str1: sadsgfadsa
replace str2: 1sd1fgsd1s
Для продолжения нажмите любую клавишу . . .
```

Строки размещаются в массивах, длина которых на 1 больше, чем длина строки. В этом дополнительном (последнем) элементе массива располагается NULL ('\0'). Во втором решении применяется функция strlen(str) для определения длины строки str, как и следовало ожидать, длина строки равна 10. Для определения длины массива используем функцию size(str), которая выдаст длину массива, равную 11. Под массивы str1 и str2 выделяем память динамически с помощью оператора new, длина этих массивов будет равна 11. Соответственно, в конце программы освобождаем эту память с помощью оператора delete[].

Алгоритм accumulate

С помощью алгоритма accumulate можно выполнять различную обработку числовых элементов, например, просуммировать элементы, найти их произведение, применить функциональные объекты и т.д.

В данной задаче для заполнения вектора воспользуемся генератором случайных чисел – srand(time(0));

```
#include <vector>
#include <numeric>
#include <functional>
void accum()
{
	// Числовой алгоритм
	// Подсчитать сумму элементов вектора
	srand(time(0)); // Генератор случайных чисел
	// time(0) - в качестве старт. числа устанавливается сист. время
	const int n = 10;
	int i, ac[n] = { 0 }; // Первоначальное заполнение
	for ( i = 0; i < n; ++i)
		cout << (ac[i] = rand() % 10) << ' '; // Заполнение массива
	cout << endl;
	vector<int> v(ac, ac + 10);   // Инициализация вектора
	int sum = 20;
	sum = accumulate(v.begin(), v.end(), sum);
	cout << "sum (включая число 20: ) " << sum << endl << endl;
	// Подсчитать произведение элементов вектора
	// Воспользуемся функциональным объектом STL multiplies<int>()
	int product1 = 1;
	product1 = accumulate(v.begin(), v.end(), product1, multiplies<int>());
	cout << "product: " << product1 << endl << endl;
}
```

Результат:

```
5 4 8 1 6 5 7 9 1 6
sum (включая число 20: ) 72

product: 1814400

Для продолжения нажмите любую клавишу . . .
```

Рассмотрим ещё одну задачу с применением алгоритма `accumulate`. Найти среднее арифметическое введенных с клавиатуры вещественных чисел. Последовательность чисел заканчивается вводом числа `0.0`.

Для того, чтобы воспользоваться манипуляторами при выводе вещественных чисел воспользуется стандартным заголовком **<iomanip>**.

```
1. #include <vector>
2. #include <numeric>
3. #include <functional>
4. #include <iomanip>
5. void accum2()
6. {
7. // Подсчитать среднее арифметическое введенных чисел
8. cout << "Введите вещественные числа с клавиатуры, ";
9. cout <<     "ввод заканчивается 0.0: ";
10.      istream_iterator<double> is(cin); // Читаем данные с клавиатуры
11.      cout << fixed;  // Настройка вывода вещественных чисел
12.      cout << setprecision(2);   // 2 цифры после запятой
13.      vector<double> v;
14.      double ar(0.0), dn;   // ar - среднее арифметическое
15.      int k(0);   // Количество введённых чисел
16.      do
17.      {
18.          dn = *is++;
19.          k++;
20.          v.push_back(dn);
21.          cout << dn << "  ";
22.      } while (*is != 0.0);
23.      cout << endl;
24.      ar = accumulate(v.begin(), v.end(), ar) / k;
25.      cout << "Количество введённых чисел = " << k << endl;
26.      cout << "Среднее арифметическое = " << ar <<endl;
27. }
```

Результат:

```
Введите вещественные числа с клавиатуры, ввод заканчивается 0.0:
3.3 5.5 9.9 7.7 0.0
3.30  5.50  9.90  7.70
Количество введённых чисел = 4
Среднее арифметическое = 6.60
Для продолжения нажмите любую клавишу . . .
```

Комментарии

(10)	**istream_iterator<double> is(cin);** – ввод вещественных чисел с клавиатуры с помощью потокового итератора.
(11-12)	Используется заголовочный файл **<iomanip>** для подключения манипуляторов при работе с потоковыми операциями. **cout << fixed;** – настройка вывода вещественных чисел. **cout << setprecision(2);** – вывод 2 цифр после запятой.
(14-15)	**double ar(0.0), dn;** **int k(0);** ar – переменная для среднего арифметического, dn – введённые числа, k – переменная для подсчёта количества введённых чисел.
(16-22)	**do** **{** **dn = *is++;** **k++;** **v.push_back(dn);** **cout << dn << " ";** **} while (*is != 0.0);** Цикл для считывания из потока чисел до числа **0.0**. Подсчитывается количество введённых чисел, числа передаются в вектор и выводятся на экран.
(24)	**ar = accumulate(v.begin(), v.end(), ar) / k;** С помощью алгоритма **accumulate** подсчитывается среднее арифметическое введённых чисел.

Задания для самостоятельного выполнения

1. Заполнить двумерный вектор таблицей умножения и вывести его на экран.
   Двумерный вектор описывается следующим образом:
   ```
   const int n = 10;
   vector < vector <int>  > v(n);
   ```
2. Заполнить вектор длиной 10 квадратами целых чисел (от 1 до 10) и вывести его на экран с помощью потокового итератора.

Контрольные вопросы

1. Какой заголовочный файл используется при работе с последовательностями?
2. Какой заголовочный файл применяется при работе с вычислительным алгоритмом?
3. Объясните работу алгоритма replace.
4. Объясните работу алгоритма reverse.
5. Каким образом можно определить длину строки и длину массива?
6. С помощью каких операторов память под массивы выделяется динамически и затем освобождается?
7. Чем отличаются алгоритмы count и count_if?
8. Какие возможности вы знаете при работе вычислительного алгоритма accumulate?
9. Каким целям служит заголовочный файл <iomanip>?
10. Какие манипуляторы применяются для настройки вывода вещественных чисел?

6. Различные алгоритмы сортировок в STL

Различные алгоритмы сортировок, отличные от стандартного алгоритма `sort`, в некоторых случаях ускоряют решение задач, поэтому познакомиться с ними будет весьма интересно и полезно. В разделе рассматривается пример с использованием разных сортировок, применяются полученные ранее знания.

Сортировки в STL

Полноценная сортировка требуется далеко не всегда, поэтому рассмотрим различные виды сортировок, отличных от стандартного алгоритма `sort`:

- `partial_sort` – частичная сортировка. Сортируется заданное число элементов, остальные остаются неотсортированными.
- `stable_sort` – стабильная сортировка, не меняет относительный порядок сортируемых элементов, имеющих одинаковые ключи.
- `nth_element` – сортировка переупорядочивает последовательность так, что все элементы, меньшие, чем тот, на который указывает итератор `nth`, оказываются перед ним, а все большие элементы – после.

Также во всех этих алгоритмах может быть применена операция сравнения, заданная программистом.

Частичная сортировка

Пусть имеется вектор объектов и нужно отобрать 20 объектов с максимальным значением. Можно выявить 20 нужных объектов и остальные оставить неотсортированными. Задача называется частичной сортировкой и для её решения существует специальный алгоритм `partial_sort`.

Частичная сортировка экономит время в тех случаях, когда нас интересуют только несколько самых больших или самых маленьких значений. `partial_sort` сортирует часть последовательности, укладывающуюся в диапазон `[first,k)`. Элементы в диапазоне `[k,last)` остаются неотсортированными, k – количество элементов, которые нужно отсортировать.

```
int sort_partial()
{
	const int m = 10;
	int d[m] = { 3, 1, 2, 34, 8, 10, 20, 2, 30, 5 };
	int i;
	partial_sort(d, d + 5, d + m, greater<int>());
	for (i = 0; i < m; i++)
		cout << d[i] << " ";
	cout << endl;
	return 0;
}
Результат:
```

```
Результат:
34 30 20 10 8 1 2 2 3 5
Для продолжения нажмите любую клавишу . . .
```

Рассмотрим вызов `partial_sort(d, d + 5, d + m, greater<int>());`

В массиве `d` всего `10` элементов, `d + 5` – указывает на шестой элемент, следовательно, необходимо отсортировать `5` элементов. Кроме того, используется функциональный объект STL `greater<int>()`, сортирующий данные в нисходящем порядке.

Стабильная сортировка

Возникает вопрос, что делают алгоритмы сортировок с одинаковыми по значению элементами. При стабильной сортировке два одинаковых элемента сохраняют свои относительные позиции после сортировки. Например, `A` предшествует `B` в неотсортированном векторе, при этом `A = B.` Стабильные алгоритмы сортировки гарантируют, что после сортировки `A,` по-прежнему, будет предшествовать `B.`

Алгоритм `partial_sort` (частичная сортировка) стабильным не является. Алгоритм `sort` также нестабильный.

Существует специальный алгоритм `stable_sort,` который не меняет относительный порядок сортируемых элементов, имеющих одинаковые ключи. `stable_sort` сортирует объекты в диапазоне `[first, last).` Если два объекта равны, их относительный порядок не изменится.

Применение стабильной сортировки будет рассмотрено в демонстрационном примере.

Сортировка *nth_element* (n-й элемент)

Алгоритм `nth_element` переупорядочивает последовательность, ограниченную диапазоном `[first, last),` так, что все элементы, меньшие, чем тот, на который указывает итератор `nth,` оказываются перед ним, а все большие элементы – после него.

Например, если есть массив:

```
int a[] = {12,18,45,31,14,76,10,13,49,8};
```

то вызов `nth_element,` в котором `nth` указывает на третий элемент (его значение равно `31,` счёт элементов начинается с `0`): `nth_element (&a[0], &a[3], &a[10]);`
создаёт последовательность, в которой все элементы, меньшие `31,` оказываются слева от `31`, а элементы, большие `31` – справа:

```
{12,18,14,10,13,8,31,45,76,49}
```

Элементы, расположенные по обе стороны от `nth,` могут быть не упорядочены.

Демонстрационный пример на сортировки

Рассмотрим задачу, в которой будут применяться описанные выше сортировки. Описать класс, содержащий информацию о собаках: порода, имя, возраст, вес. Предусмотреть конструктор. Далее выполнить следующие действия:

- Переопределить у этого класса оператор вывода в поток.
- Написать функцию заполнения вектора, в соответствии с полями класса, произвольными данными (6-9 записей).
- Написать функцию печати содержимого вектора.
- Отсортировать вектор по именам собак по возрастанию.
- Отсортировать вектор по породам собак по убыванию.
- Поставить в первые пять элементов вектора самых старших собак по возрасту в убывающем порядке.
- Отсортировать следующим образом: первые 4 элемента вектора – собаки с самым большим весом, в рамках одного веса упорядочиваются по возрасту.
- После каждой операции выводить список в выходной поток.

Приведём проект практически полностью.

```
1. // Dogs_sort.cpp : This file contains the 'main' function.
2. // Program execution begins and ends there.

3. #include <iostream>
4. #include <string>
5. #include <vector>
```

```
#include <iomanip>
#include <functional>
#include <algorithm>

using namespace std;

//=========================================
class Dogs
{
public:
string Breed;  // Порода
string Name;
int Age;
int Weight;
Dogs(string ABreed, string AName, int AAge, int AWeight) :
    Breed(ABreed), Name(AName), Age(AAge), Weight(AWeight) {};
// Конструктор
friend ostream& operator << (ostream& s, const Dogs& ss)
    {
        // Переопределение оператора <<
        s.setf(ios::left);
        return s << setw(12) << ss.Breed
            << setw(10) << ss.Name
            << setw(8) << ss.Age
            << setw(8) << ss.Weight
            << endl;
    }
};
//=========================================

// Описание прототипов функций
void FillVector(vector <Dogs>&a);
bool LessName(const Dogs & s1, const Dogs & s2);
bool GreaterBreed(const Dogs & s1, const Dogs & s2);
bool GreaterAges(const Dogs & s1, const Dogs & s2);
bool AgeWeight(const Dogs & s1, const Dogs & s2);
int SM();

int main()
{
system("color F0");  // Установка белого фона и черного текста
setlocale(LC_ALL, "rus");
SM();
system("pause");
}
//=========================================
template <class D>
// Шаблонная функция для вывода на экран содержимого контейнера
void print(D& d)
{
    typename D::iterator it;
    for (it = d.begin(); it != d.end(); it++)
        cout << *it;
    cout << endl;
}
```

```
57.  //========================================
58.  void FillVector(vector<Dogs>& d)   // Ввод данных
59.  {
60.  d.push_back(Dogs("Pug", "Gary", 2, 16));
61.  d.push_back(Dogs("Poodle", "Jack", 3, 17));
62.  d.push_back(Dogs("Boxer", "Casper", 6, 17));
63.  d.push_back(Dogs("Labrador", "Ben", 4, 15));
64.  d.push_back(Dogs("PitBull", "Lary", 5, 17));
65.  d.push_back(Dogs("Rottweiler", "Skay", 3, 10));
66.  d.push_back(Dogs("Basenji", "Frank", 4, 8));
67.  d.push_back(Dogs("Labrador", "Dany", 7, 12));
68.  d.push_back(Dogs("Poodle", "Lars", 3, 14));
69.  }
70.  //========================================
71.  // Напишем предикаты для разных сортировок
72.  // По имени по возрастанию
73.  bool LessName(const Dogs& s1, const Dogs& s2)
74.  {
75.      return s1.Name < s2.Name;
76.  }

77.  // По породе по убыванию
78.  bool GreaterBreed(const Dogs& s1, const Dogs& s2)
79.  {
80.      return s1.Breed > s2.Breed;
81.  }

82.  // Поместить в вектор
83.  // самых младших собак по возрасту, по убыванию
84.  bool GreaterAges(const Dogs& s1, const Dogs& s2)
85.  {
86.      return s1.Age > s2.Age;
87.  }

88.  // Поместить в вектор собак с самым большим весом,
89.  // в рамках одного веса упорядочить по возрасту
90.  bool AgeWeight(const Dogs& s1, const Dogs& s2)
91.  {
92.      if (s1.Weight > s2.Weight) return true;
93.      else if (s1.Weight < s2.Weight) return false;
94.      else return s1.Age > s2.Age;
95.  }
96.  //========================================
97.  int SM()  // Основная функция
98.  {
99.  vector <Dogs> d;
100. FillVector(d);
101. vector <Dogs> f(d); //Создается копия вектора "d" в вектор "f"
102. cout << "Начальный вектор: " << endl;
103. cout << "Порода      Имя       Возраст Вес " << endl;
104. print(d);

105. // По имени в возрастающем порядке
106. d = f;
107. sort(d.begin(), d.end(), LessName);
```

```
108.  cout << "Сортировка (sort) по имени в возр. порядке: " << endl;
109.  cout << "      (sort(d.begin(), d.end(), LessName))" << endl;
110.  cout << "Порода      Имя       Возраст Вес " << endl;
111.  print(d);

112.  // По породам по убыванию
113.  d = f;
114.  stable sort(d.begin(), d.end(), GreaterBreed);
115.  cout << "Стабильная сорт.(stable sort) по породам в убыв. порядке: " <<
116.  endl;
117.  cout << "      (stable_sort(d.begin(), d.end(), GreaterBreed))" <<
118.  endl;
119.  cout << "Порода      Имя       Возраст Вес " << endl;
120.  print(d);

121.  // Поместить в первые пять элементов вектора - самых старших по
122.  // возрасту собак, по убыванию
123.  d = f;
124.  partial sort(d.begin(), d.begin() + 5, d.end(), GreaterAges);
125.  cout << "Частичная сорт., первые 5 элементов - самые старшие собаки по
   убыванию: " << endl;
126.  cout << "      (partial_sort(d.begin(), d.begin() + 5, d.end(),
   GreaterAges))" << endl;
127.  cout << "Порода      Имя       Возраст Вес " << endl;
128.  print(d);

129.  // Поместить в первые 4 элемента вектора
130.  // собак с самым большим весом, упорядочив по возрасту.
131.  d = f;
132.  partial sort(d.begin(), d.begin() + 4, d.end(), AgeWeight);
133.  cout << "Частичная сорт., первые 4 элемента – собаки с самым большим
   весом, упорядочиваются по возрасту: " << endl;
134.  cout << "      (partial_sort(d.begin(), d.begin() + 4, d.end(),
   AgeWeight))" << endl;
135.  cout << "Порода      Имя       Возраст Вес " << endl;
136.  print(d);

137.  cout << endl << endl;
138.  return 0;
139.  }
```

Результат:

```
Начальный вектор:
Порода      Имя       Возраст Вес
Pug         Gary      2       16
Poodle      Jack      3       17
Boxer       Casper    6       17
Labrador    Ben       4       15
PitBull     Lary      5       17
Rottweiler  Skay      3       10
Basenji     Frank     4       8
Labrador    Dany      7       12
Poodle      Lars      3       14
```

```
Сортировка (sort) по имени в возр. порядке:
      (sort(d.begin(), d.end(), LessName))
Порода      Имя       Возраст Вес
Labrador    Ben       4       15
Boxer       Casper    6       17
Labrador    Dany      7       12
Basenji     Frank     4       8
Pug         Gary      2       16
Poodle      Jack      3       17
Poodle      Lars      3       14
PitBull     Lary      5       17
Rottweiler  Skay      3       10
```

```
Стабильная сорт.(stable sort) по породам в убыв.   порядке:
      (stable_sort(d.begin(), d.end(), GreaterBreed))
Порода      Имя       Возраст Вес
Rottweiler  Skay      3       10
Pug         Gary      2       16
Poodle      Jack      3       17
Poodle      Lars      3       14
PitBull     Lary      5       17
Labrador    Ben       4       15
Labrador    Dany      7       12
Boxer       Casper    6       17
Basenji     Frank     4       8
```

```
Частичная сорт., первые 5 элементов - самые старшие собаки по убыванию:
      (partial_sort(d.begin(), d.begin() + 5, d.end(), GreaterAges))
Порода      Имя       Возраст Вес
Labrador    Dany      7       12
Boxer       Casper    6       17
PitBull     Lary      5       17
Basenji     Frank     4       8
Labrador    Ben       4       15
Pug         Gary      2       16
Poodle      Jack      3       17
Rottweiler  Skay      3       10
Poodle      Lars      3       14
```

```
Частичная сорт., первые 4 элемента - с самым большим весом по возрасту:
      (partial_sort(d.begin(), d.begin() + 4, d.end(), AgeWeight))
Порода      Имя       Возраст Вес
Boxer       Casper    6       17
PitBull     Lary      5       17
Poodle      Jack      3       17
Pug         Gary      2       16
Labrador    Ben       4       15
Poodle      Lars      3       14
Basenji     Frank     4       8
Rottweiler  Skay      3       10
Labrador    Dany      7       12
```

Комментарии

(6)	Заголовочный файл **<iomanip>** необходим для подключения манипуляторов при работе с потоковыми операциями. Манипуляторы могут использоваться в составе выражений ввода/вывода. В частности, неоднократно применялся манипулятор endl – переход на новую строку. В пункте 23-29 будем рассматривать другие манипуляторы для форматирования вывода.
(11-31)	Описывается класс Dogs: **class Dogs** **{** **public:** // Класс содержит 4 поля: порода, имя, возраст и вес собаки. **string Breed;** **string Name;** **int Age;** **int Weight;...** Далее идёт описание конструктора, с помощью которого инициализируются переменные класса. **Dogs(string ABreed, string AName, int AAge, int AWeight) :** **Breed(ABreed), Name(AName), Age(AAge), Weight(AWeight) {};**
(21)	Производится перегрузка оператора вывода в поток (**operator <<**) **friend ostream& operator << (ostream& s, const Dogs& ss)**
(23-29)	Применяются манипуляторы, чтобы при выводе данные принимали вид таблицы. Функция **setf(ios::left)** входит в состав класса ostream для вывода, в данном случае запрашивается выравнивание влево. Как правило, функция setf() применяется к объекту cout. Но поскольку оператор вывода был перегружен (см п. 21), теперь все манипуляторы применяются к объектам s и ss. Манипулятор setw(n) устанавливает ширину каждого поля. **//переопределение оператора <<** **s.setf(ios::left);** **return s << setw(12) << ss. Breed** **<< setw(10) << ss.Name** **<< setw(8) << ss.Age** **<< setw(8) << ss.Weight** **<< endl;**
(33-34)	Описываются прототипы функций, применяемые далее. Прототипом функции в C++ называется *объявление* функции, не содержащее тела функции, но указывающее имя функции, количество аргументов, типы аргументов и возвращаемый тип данных. *Определение* функции описывает, что именно делает функция. Прототип функции можно рассматривать как описание её интерфейса. По правилам языка C++ каждую функцию перед использованием необходимо объявить. Прототипы позволяют компилятору выполнить проверку количества аргументов, типов и их преобразования. **void FillVector(vector <Dogs>&a); –** заполение вектора данными.
(34-38)	Следующие четыре функции – предикаты, выдают булевские значения и будут применяться при вызове алгоритмов сортировки для настройки сравнения полей класса Dogs по имени, породе, возрасту и комбинации возраста и веса. **bool LessName(const Dogs & s1, const Dogs & s2);** **bool GreaterBreed(const Dogs & s1, const Dogs & s2);** **bool GreaterAges(const Dogs & s1, const Dogs & s2);** **bool AgeWeight(const Dogs & s1, const Dogs & s2);**
(39)	Функция **int SM();** является основной функцией, которая запускается в этом проекте.
(42)	**system("color F0");** – установка белого фона и черного текста при выводе результата.
(48-56)	Описывается шаблонная функция для вывода на экран содержимого контейнера, в данном случае – это будет **vector <Dogs> d;** В дальнейшем вывод на печать будет производиться при вызове функции **print(d);**
(71-95)	Определяются предикаты для разных сортировок, в них сравниваются поля класса **Dogs** по различным параметрам.

(99)	Создается вектор на основе класса **Dogs**, т.е. типом вектора будет класс: **vector <Dogs> d;**
(101)	**vector <Dogs> f(d); –** создается копия вектора **d** в вектор **f.** Копия создаётся для того, чтобы при вызове каждой новой сортировки можно было работать с начальными данными. Начальные данные находятся в векторе **d,** затем производится изменение вектора **d** и его печать **print(d);** Для восстановления данных используется оператор **d = f;**
(107)	**sort(d.begin(), d.end(), LessName);** Обычная сортировка по имени в возрастающем порядке. Применяется предикат **LessName.**
(114)	**stable_sort(d.begin(), d.end(), GreaterBreed);** Стабильная сортировка по породам в убывающем порядке. Применяется предикат **GreaterBreed.**
(124)	**partial_sort(d.begin(), d.begin() + 5, d.end(), GreaterAges);** Частичная сортировка: выбираются 5 самых старших собак, упорядочиваются в убывающем порядке. Применяется предикат **GreaterAges.**
(132)	**partial_sort(d.begin(), d.begin() + 4, d.end(), AgeWeight);** Частичная сортировка: выбираются 4 собаки с наибольшим весом, в рамках одного веса упорядочиваются по возрасту. Применяется предикат **AgeWeight.**

Задания для самостоятельного выполнения

Описать класс «студент» с полями: имя, фамилия, курс. Выполнить следующие действия:

- Переопределить у этого класса оператор вывода в поток.
- Написать функцию заполнения вектора из класса «студент» произвольными данными (10 записей).
- Написать функцию печати содержимого вектора.
- Отсортировать вектор по именам студентов по возрастанию (стабильная сортировка).
- Отсортировать вектор по фамилиям студентов по убыванию.
- Поместить в первые семь элементов вектора студентов самых младших курсов, в рамках одного курса упорядочить по фамилии в возрастающем порядке. После каждой операции выводить список студентов в выходной поток.

Контрольные вопросы

1. Какие различные виды сортировок в STL вы знаете?
2. Что такое частичная сортировка?
3. Какие особенности у стабильной сортировки?
4. В чём особенность сортировки `nth_element`?
5. Какие манипуляторы заголовочного файла `<iomanip>` вы узнали в этой практической работе?
6. Что означает перегрузка оператора вывода?
7. Чем объявление функции отличается от определения функции?
8. Что такое прототип функции и зачем он используется?
9. Каким образом можно написать предикат для упорядочения по двум ключам (на примере `AgeWeight())?`
10. Объясните полученные результаты различных сортировок в приведённом примере.

7. Ассоциативные контейнеры

Ассоциативные контейнеры – это упорядоченные структуры, которые позволяют реализовать быстрый поиск элементов. Ассоциативные контейнеры могут служить хорошей базой при создании информационных систем. В разделе рассматривается теоретический материал, несколько программ, приводятся задачи для самостоятельного решения.

Типы ассоциативных контейнеров

Переходим к изучению второго типа контейнеров – ассоциативных. Обсудим кратко физическую реализацию контейнеров, поскольку именно от неё зависит эффективность хранения и поиска данных.

Для реализации последовательных контейнеров (векторов, двусторонних очередей и списков) используются, в основном, массивы и списки. Ассоциативные контейнеры обычно реализованы в виде сбалансированных деревьев.

Дадим определение сбалансированного дерева.

Бинарное дерево называется сбалансированным или АВЛ-деревом, если для любой вершины дерева, высоты левого и правого поддеревьев отличаются не более чем на единицу. М. Адельсон-Вельский и Е.М. Ландис доказали, что при таком определении можно написать программы добавления/удаления и поиска, имеющие логарифмическую сложность, а также сохраняющие дерево сбалансированным.

Всего существует 4 типа ассоциативных контейнеров:

- множества (`set`),
- множества с дубликатами (`multiset`),
- словари (`map`),
- словари с дубликатами (`multimap`),

Как последовательные контейнеры, так и ассоциативные, имеют операции с одинаковым названием и смыслом, например, `begin()`, `end()`, `size()`, `empty()` и др. К ассоциативным контейнерам также применяются итераторы, которые являются двунаправленными.

В отличие от последовательных контейнеров ассоциативные контейнеры хранят свои элементы отсортированными, не зависимо от того, в каком порядке они были введены.

Необходимо заметить, что кроме упорядоченных ассоциативных контейнеров: `set`, `multiset`, `map`, `multimap`, существуют ещё и хешированные ассоциативные контейнеры: `hash_set`, `hash_multiset`, `hash_map`, `hash_multimap`. Это – неупорядоченные контейнеры, реализованные на основе хэш-таблиц. Работа с хэш-таблицей начинается с вычисления хэш-функции от ключа. Более подробно о хешированных ассоциативных контейнерах можно прочитать в приведенной литературе.

Множества и множества с дубликатами

При работе, как с множествами, так и с множествами с дубликатами применяется один и тот же заголовочный файл:

```
#include <set>.
```

Каждый элемент множества является собственным ключом, и эти ключи уникальны. Поэтому два различных элемента множества не могут совпадать. Например, множество может состоять из следующих элементов:

```
51   193    279   765.
```

Множество с дубликатами отличается от простого множества только тем, что содержит несколько совпадающих элементов.

```
51   193    279   279.
```

Поскольку ассоциативные контейнеры хранятся в отсортированном виде, то при их описании должна присутствовать функция упорядочивания, в соответствии с которой эти множества сортируются, несмотря на то, как они были созданы.

```
#include <set>
#include <algorithm>
int set1()
{
	set<int, less<int> > S_1, S_2; // Множества
	S_1.insert(1);  S_1.insert(2);  S_1.insert(3); S_1.insert(1);
	S_2.insert(2);  S_2.insert(3);  S_2.insert(1);
	cout << "Множество S_1: ";
	copy(S_1.begin(), S_1.end(), ostream_iterator<int>(cout, " "));
	cout << endl;
	cout << "Множество S_2: ";
	copy(S_2.begin(), S_2.end(), ostream_iterator<int>(cout, " "));
	cout << endl;
	return 0;
}
```

Результат:

```
Множество S_1: 1 2 3
Множество S_2: 1 2 3
Для продолжения нажмите любую клавишу . . .
```

Определение множеств `S_1 и S_2: set<int, less<int> > S_1, S_2;` Обратите внимание, что символы **> >** разделены пробелом, в отличие от **>>** – ввода данных в поток. В этом определении два параметра: первый – тип данных (в данном случае `int`), второй – способ упорядочивания (в рассматриваемом примере применяется предикат `less<int>,` упорядочивающий в возрастающем порядке, т.е. `k1 < k2` , где `k1` и `k2` являются ключами). Порядок, в котором данные вводились в `S_2` не существенен, поскольку эти данные упорядочиваются. Ввод второй раз в `S_1` числа `1` не меняет множество, поскольку ключи в множестве уникальны.

Рассмотрим тот же пример, но с множеством с дубликатами.

```
#include <set>
#include <algorithm>
int multiset1()
{
	multiset<int, less<int> > MS_1, MS_2;  // Мультимножества
	MS_1.insert(1);  MS_1.insert(2);  MS_1.insert(3); MS_1.insert(1);
	MS_2.insert(2);  MS_2.insert(3);  MS_2.insert(1);
	cout << "Мультимножество MS_1:  ";
	copy(MS_1.begin(), MS_1.end(), ostream_iterator<int>(cout, " "));
	cout << endl;
	cout << "Мультимножество MS_2:  ";
	copy(MS_2.begin(), MS_2.end(), ostream_iterator<int>(cout, " "));
	cout << endl;
	return 0;
}
```

Результат:

```
Мультимножество MS_1:  1 1 2 3
Мультимножество MS_2:  1 2 3
Для продолжения нажмите любую клавишу . . .
```

Множества с дубликатами определяются также (используется два параметра): `multiset<int, less<int> > MS_1, MS_2;` Однако одинаковые данные могут вводиться многократно. В приведенных программах `set1()` и `multiset1()` для вывода данных применяются потоковые итераторы.

Словари и словари с дубликатами

Примеры работы со словарями

Для работы со словарями и со словарями с дубликатами подключается заголовочный файл **`<map>: #include <map>.`**

Каждый элемент словаря и словаря с дубликатами представляет собой пару «ключ – значение». Существенно, что ключ и значение могут принадлежать разным типам.

В словаре `map` не может быть двух одинаковых ключей, но в словаре с дубликатами `multimap` – могут. В описании как `map,` так и `multimap` содержится 3 параметра, например: `map< char*, long, compare> D;` где:

- `char*` – тип ключа,
- `long` – тип сопутствующих данных,
- `compare` – третье поле, здесь определяется операция сравнения, возможно, она представлена с помощью функционального объекта, например, так:

```
class compare // Функциональный объект для словаря (сравнение по ключу)
{
public:
   bool operator()(const char *s, const char *t) const
   {
         return strcmp(s, t) <  0;
   }
};
```

Доступ к данным осуществляется с помощью итераторов, но при разыменовании итератора получаем экземпляр объекта «пара» («ключ – значение»), поэтому, чтобы получить доступ к ключу или к сопутствующему значению, используются операторы « `->` » или « `.` » для выделения ключа или значения. Пусть `it` – итератор, тогда к содержимому пары получаем доступ таким образом: `it -> first, it -> second,` либо таким: `(*it).first, (*it).second.`

Рассмотрим простой пример: создадим словарь, введём в него данные и выведем на экран. Ключём является поле `string,` сопутствующее поле – `double.` Упорядочение производится по умолчанию, от меньшего к большему.

В словаре все ключевые элементы являются уникальными. Рассмотрим пример работы со словарём:

```
1.    #include <iomanip>
2.    #include <map>
3.    #include <string>
4.    int map1()
5.    {
```

```
6.        map <string, double> D_1 =
7.        {
8.            { "Mouse", 389.4 },
9.            { "Monitor", 20000.3 },
10.           { "Keyboard", 900.5 },
11.           { "Mousepad", 310.8 }
12.        };
13.      cout.setf(ios::right);
14.      for (auto i = D_1.begin(); i != D_1.end(); ++i)
15.        {
16.   cout << setw(10) << i->first << setw(10) << i->second << endl;
17.       }
18.   return 0;
19.    }
```

Результат:

```
  Keyboard     900.5
   Monitor   20000.3
     Mouse     389.4
  Mousepad     310.8
Для продолжения нажмите любую клавишу . . .
```

Комментарии

(1)	Заголовочный файл **<iomanip>** необходим для подключения манипуляторов при работе с потоковыми операциями. Применение рассматривается в п. 13 и п. 16.
(6)	Описывается словарь с двумя полями **map <string, double> D_1,** первое поле – ключевое, второе – сопутствующее. Упорядочение производится по умолчанию. Обратите внимание, что при выводе значения ключевого поля упорядочены по алфавиту. Инициализация контейнера производилась явно, строки программы 8–11.
(13)	Применяются манипуляторы, чтобы при выводе данные принимали вид таблицы. Функция **setf(ios::right)** входит в состав класса ostream для вывода, в данном случае запрашивается выравнивание вправо, функция setf() применяется к объекту cout.
(14)	Подготавливается цикл для вывода на печать. Определяем итератор **i** с помощью спецификатора **auto,** который позволяет не указывать тип переменной явно, тип определяется на основе инициируемого значения. В данном случае **i** – итератор типа **map.**
(16)	Для вывода на экран применяется манипулятор **setw(10),** который означает, что на каждое поле отводится **10** позиций. Поскольку в словаре два поля, то значение одной записи представляет собой пару и доступ к каждому из полей осуществляется следующим образом: **i->first и i->second.**

Если заменить 13 строку на **cout.setf(ios::left);** то данные полей при выводе будут выравниваться влево.

Результат:

```
Keyboard  900.5
Monitor   20000.3
Mouse     389.4
Mousepad  310.8
Для продолжения нажмите любую клавишу . . .
```

Для следующего примера определим словарь D_2 таким образом:
map <char, int> D_2; Ключевое поле – char, сопутствующее поле – int.

```
int map2()
{
	char c;
	map <char, int> D_2;
	for (int i = 0, c = 'a'; i < 5; ++i, ++c)
	{
		D_2.insert(pair<char, int>(c, i));
	}
	for (auto i = D_2.begin(); i != D_2.end(); ++i)
	{
		cout << (*i).first << " : " << (*i).second << endl;
	}
	return 0;
}
```

Результат:

```
a : 0
b : 1
c : 2
d : 3
e : 4
Для продолжения нажмите любую клавишу . . .
```

В этом примере инициализация производится неявно, с помощью функции `insert()`. Применена другая форма вывода полей пары по ссылке: `(*i).first, (*i).second`.

В цикле при каждой новой итерации производится наращивание двух переменных `c (char)` и `i (int)`, отвечающих за первое и второе поля в словаре.

Поскольку значения ключа в словаре уникальны, то для них применим оператор индексации [ ]. Этот оператор постоянно используется с массивами, когда нужно ввести или найти элемент по индексу. В словарях тоже можно использовать индексацию. Если рассмотреть функцию `insert()` – функцию вставки нового элемента: `D1.insert(make_pair (num_1, num_2));` `num_1` – ключ, `num_2` – значение, то тоже самое можно сделать, применяя оператор индексации `D1[num_1] = num_2;`

Рассмотрим пример словаря, в котором данные вводятся с помощью индексации, применяются функции `size()` – определение количества записей в словаре, `find()` – поиск по ключу и `erase(it)` – удаление записи.

```
1.  #include <iomanip>
2.  #include <map>
3.  #include <string>
4.  #include <algorithm>
5.  int map3()   // Телефонный справочник (работа со словарём)
6.  {
7.        map <string, long> phone_book;
8.        phone_book["Petrov"] = 12345;
9.        phone_book["Sokolov"] = 45678;
10.       phone_book["Denisov"] = 98765;

11.       cout << "Number of records: " << phone_book.size() << endl;

12.       map <string, long> ::iterator it;
13.       for (it = phone_book.begin(); it != phone_book.end(); ++it)
```

```
14.       {
15.       cout << setw(8) << it->first << setw(6) << it->second << endl;
16.       }
17.       cout << endl;
18.       it = phone_book.begin();
19.       it = phone_book.find("Denisov"); // Ищем по фамилии
20.       cout << "search record Denisov: " << endl;
21.       cout << setw(8) << it->first << setw(6) << it->second << endl << endl;
22.       cout << "delete found record " << endl << endl;
23.       phone_book.erase(it);   // Удаляем найденную запись
24.       cout << "Number of records: " << phone_book.size() << endl;
25.       for (it = phone_book.begin(); it != phone_book.end(); ++it)
26.       {
27.       cout << setw(8) << it->first << setw(6) << it->second << endl;
28.       }
29.       return 0;
26. }
```

Результат:

```
Number of records: 3
 Denisov 98765
  Petrov 12345
 Sokolov 45678

search record Denisov:
 Denisov 98765

delete found record

Number of records: 2
  Petrov 12345
 Sokolov 45678
Для продолжения нажмите любую клавишу . . .
```

Комментарии

(7)	Описывается словарь с двумя полями **map <string, long> phone_book,** первое поле – ключевое (фамилия), второе – сопутствующее (номер телефона). Упорядочение производится по умолчанию. Обратите внимание, что при выводе значения ключевого поля упорядочены по алфавиту. Инициализация контейнера производилась явно, строки программы 6–8.
(11)	Используется метод **size()** – определение количества записей в словаре, **phone_book.size().**
(12)	Определяется итератор **map <string, long>::iterator it;** соответствующий заданному словарю.
(13-16)	В цикле выводятся все данные контейнера. Используются манипуляторы **setw(8)** и **setw(6),** чтобы при выводе данные принимали вид таблицы. Доступ к фамилиям и номерам телефонов осуществляется с помощью **it->first** и **it->second.**
(18)	Заданный итератор устанавливается на начало словаря **it = phone_book.begin();**
(19)	Для запроса на поиск по фамилии используется метод **find()** – поиск по ключу. **it = phone_book.find("Denisov");** Номер найденной записи хранится в итераторе.
(21)	Вывод на экран записи, соответствующей итератору, производится аналогично тому, как описано выше.
(23)	Для удаления записи, соответствующей итератору, применяется метод **erase(it). phone_book.erase(it);**

Пример работы со словарями с дубликатами

Рассмотрим работу со словарями с дубликатами. *Оператор доступа по индексу [ ] не определен для словарей с дубликатами,* поскольку одному ключу может соответствовать несколько сопутствующих значений.

Разберём пример, аналогичный предыдущему, но для словарей с дубликатами. В словаре с дубликатами может быть несколько записей с одинаковым значением ключа. Для ввода данных будем применять метод `insert()`. Рассмотрим также методы `size()`, `find()` и `erase()`,– для записей с одинаковыми ключами они имеют свои нюансы.

```
1. #include <iomanip>
2. #include <map>
3. #include <string>
4. #include <algorithm>
5. typedef multimap<string, long> MD; // Определение типа MD
6. MD phone_book;

7. void DeleteRecord(string name, long phonenumber)
8. {
9.     //Функция для удаления записи по фамилии и номеру телефона
10.    auto it = phone_book.begin();
11.    for (it; it != phone_book.end(); ++it)
12.      {
13.         if (it->first == name) if (it->second == phonenumber)
14.         {
15.             cout << "Erase found record:" << setw(8) << it->first <<
16.             setw(6) << it->second << endl;
17.             phone_book.erase(it);  // удаляем найденную запись
18.             break;
19.          }
20.       }
21.    }

22.    int Dmap1()  // Телефонный справочник для словаря с дубликатами
23.    {
24.    phone_book.insert(MD::value_type("Petrov", 78900));
25.    phone_book.insert(MD::value_type("Petrov", 54321));
26.    phone_book.insert(MD::value_type("Petrov", 12345));
27.    phone_book.insert(MD::value_type("Sokolov", 45678));
28.    phone_book.insert(MD::value_type("Denisov", 98765));
29.    cout << "Number of records: " << phone_book.size() << endl;
30.    multimap <string, long> ::iterator it;
31.    for (it = phone_book.begin(); it != phone_book.end(); ++it)
32.    {
33.         cout << setw(8) << it->first << setw(6) << it->second << endl;
34.    }
35.    cout << endl;
36.    it = phone_book.begin();
37.    it = phone_book.find("Denisov"); // Находим по фамилии
38.    cout << "Search record Denisov: " << endl;
39.    cout << setw(8) << it->first << setw(6) << it->second << endl << endl;
40.    // Находим по фамилии всех Петровых
41.    cout << "Search records Petrov: " << endl;
42.    for (it = phone_book.begin();  it != phone_book.end(); ++it)
```

```
43.   {
44.       if (it->first == "Petrov")
45.       cout << setw(8) << it->first << setw(6) << it->second << endl;
46.   }
47.   // Удаление
48.   cout << endl;
49.   DeleteRecord("Petrov", 12345);
50.   cout << endl;
51.   cout << "Number of records: " << phone_book.size() << endl;
52.   for (it = phone_book.begin(); it != phone_book.end(); ++it)
53.     {
54.       cout << setw(8) << it->first << setw(6) << it->second << endl;
55.     }
56.   return  0;
57.   }
```

Результат:

```
Number of records: 5
 Denisov 98765
  Petrov 78900
  Petrov 54321
  Petrov 12345
 Sokolov 45678

Search record Denisov:
 Denisov 98765

Search records Petrov:
  Petrov 78900
  Petrov 54321
  Petrov 12345

Erase found record:  Petrov 12345

Number of records: 4
 Denisov 98765
  Petrov 78900
  Petrov 54321
 Sokolov 45678
Для продолжения нажмите любую клавишу . . .
```

Комментарии

(5-6)	**typedef multimap<string, long> MD;** **MD phone_book;** **typedef –** оператор определения новых типов, который позволяет создать собственное имя типа. Так в программе оператор **typedef multimap<string, long> MD;** означает, что создан новый тип **MD,** имеющий определение как **multimap<string, long>.** Теперь можно создавать объекты этого типа, например: **MD phone_book;**
(7)	Функция для удаления записи по фамилии и номеру телефона **void DeleteRecord(string name, long phonenumber).** Она нам пригодится, когда нужно будет удалить конкретную запись, с определёнными значениями фамилии и номера телефона (при наличии записей с одинаковыми значениями ключей).
(10)	Определяется итератор, соответствующий заданному словарю, с помощью оператора **auto: auto it = phone_book.begin();**
(11-21)	Операция удаления разбивается на две части: сначала находим искомую запись, а потом её удаляем.

	В цикле ищется необходимая запись по введённым параметрам: **if (it->first == name) if (it->second == phonenumber),** выводится на экран найденная запись, итератор зафиксирован и теперь (по значению итератора) можно её удалить **phone_book.erase(it);**
(22)	Рассмотрим работу программы **int Dmap1().**
(24-28)	Для ввода данных используется метод **insert()** со следующим синтаксисом: **phone_book.insert(MD::value_type("Petrov", 78900));** **value_type** – определение типа элементов в контейнере. **value_type("Petrov", 78900)** – это упорядоченная пара, в которой первый элемент эквивалентен значению ключа, а второй элемент – данные, соответствующие ключу. Перед **value_type** требуется написать префикс **MD::** (тип контейнера). Обращаю ваше внимание на то, что в словаре *три записи* с фамилией **Petrov.** В связи с этим и возникают вопросы, какую именно запись удалить и как найти все записи с одинаковым ключом.
(29)	Используется метод **size()** для определения количества записей в словаре, **phone_book.size().**
(30)	Определяется итератор **multimap <string, long>::iterator it;** и далее данные словаря выводятся на печать.
(37-39)	Итератор устанавливается на начало и вызывается метод **find():** **it = phone_book.find("Denisov");** находящий первое вхождение заданного параметра.
(42-46)	Поставим задачу найти в словаре с дубликатами всех людей по фамилии **Petrov**. В цикле выполняем оператор **if (it->first == "Petrov")** и, если он даёт результат **true,** выводим на экран.
(49)	Для удаления записи вызывается функция **DeleteRecord("Petrov", 12345);** в ней указываются необходимые параметры.
(52-55)	Вывод на экран оставшихся записей.

Выводы:

1. Элементы, помещаемые в упорядоченные ассоциативные контейнеры, размещены не в порядке их ввода, как это было с последовательными контейнерами, а в отсортированном, по значению ключа, виде.
2. В соответствии с п.1 для ключа должна существовать операция сравнения. В некоторых случаях эта операция используется по умолчанию, иногда описывается явно, также может быть создана пользователем в виде, например, функционального объекта.
3. Поиск в ассоциативных контейнерах выполняется эффективно и быстро, – это основано на их физической реализации.

Алгоритмы работы с множествами для упорядоченных контейнеров

Поскольку ассоциативные контейнеры являются множествами, то для них можно выполнять алгоритмы работы с множествами (выходные последовательности упорядочены):

- `includes` – выполняет проверку *включения* одной последовательности в другую. Результат равен `true` в том случае, когда каждый элемент второй последовательности содержится в первой последовательности.
- `set_intersection` – создаёт отсортированное *пересечение множеств*, то есть множество, содержащее только те элементы, которые одновременно входят и в первое, и во второе множества.
- `set_difference` – формирует отсортированную последовательность из элементов, входящих в первую последовательность, но отсутствующих во второй из двух последовательностей.

– `set_union` – создаёт отсортированное *объединение множеств,* то есть множество, содержащее элементы первого и второго множеств без повторяющихся элементов.
– И др.

Необходимо заметить, что критерий упорядочивания элементов данных операций может быть указан предикатом.

На примере следующей программы рассмотрим работу некоторых их этих алгоритмов. Задача формулируется таким образом: есть контейнеры разных видов (`list, vector, deque`), содержащие целые числа. Необходимо разделить каждый из них на две, возможно, неравные части. Создать два контейнера `<set>` и перенести данные из каждой части в отдельный контейнер, а затем между этими двумя контейнерами выполнить операции объединения, включения и пересечения.

```
#include <list>
#include <vector>
#include <deque>
#include <set>
#include <iterator>
#include <algorithm>
//  -------------- Работа с ассоциативными контейнерами-----
template <class A> void showT(A& cont)
    {   // Шаблонная функция для вывода данных
   typename A::const_iterator i = cont.begin();
   if (cont.empty()) cout << "Container is empty";
   for (i; i != cont.end(); ++i) cout << *i << ' ';
   cout << endl;
   }

   void my_example()
   {
   // set_union (объединение множеств)
   setlocale(LC_ALL, "Russian");
   cout << "==== объединение ====" << endl;
   list <int> L{ 4, 2, 3, 4, 8, 1, 9, 3, 4, 5, 7, 7 }; // 12 чисел
   cout << "Исходный список: "; showT(L);
   set<int> s1, s2, res;
   list <int>::iterator l1 = L.begin(), l2 = L.begin();
   int dis = L.size() * 2 / 4;
   // L.size() * 2 / 4 = 12 * 2 / 4 = 6,  dis = 6
   advance(l2, dis);
   copy(l1, l2, inserter(s1, s1.begin()));
   copy(l2, L.end(), inserter(s2, s2.begin()));
   set_union(begin(s1), end(s1), begin(s2), end(s2),
       inserter(res, res.begin()));
   cout << "Первое множество: "; showT(s1);
   cout << "Второе множество: "; showT(s2);
   cout << "set_union: "; showT(res);

   // includes (включение одной последовательности в другую)
   cout << endl << "==== включение ====" << endl;
   vector<int> v{ 7, 1, 8, 3, 1, 2, 3, 4, 5, 7, 8, 0 };  //12 чисел
   cout << "Исходный вектор: "; showT(v);
   set<int> s3, s4;
   vector <int>::iterator v1 = v.begin(), v2 = v.begin();
   dis = v.size() * 2 / 6;
```

```
41.   // v.size() * 2 / 6 = 12 * 2 / 6 = 4,  dis = 4
42.   advance(v2, dis);
43.   copy(v1, v2, inserter(s3, s3.begin())); // В s3 4 элемента
44.   copy(v2, v.end(), inserter(s4, s4.begin())); // В s4 8 элементов
45.   cout << "Первое множество: "; showT(s4);
46.   cout << "Второе множество: "; showT(s3);
47.   bool r = includes(begin(s4), end(s4), begin(s3), end(s3));
48.   cout << "Вхождение 2-го множества в 1-е: " << endl;
49.   cout << (r ? "true" : "false") << endl << endl;

50.   // set_intersection (пересечение множеств)
51.   cout << endl << "==== пересечение ====" << endl;
52.   deque<int> d{ 2, 3, 7, 8, 0, 1, 2, 0, 5, 7, 6, 3, 4, 4 }; //14
53.   cout << "Исходная очередь: "; showT(d);
54.   set<int> s5, s6, res1;
55.   deque<int>::iterator d1 = d.begin(), d2 = d.begin();
56.   dis = d.size() * 2 / 7;
57.   // d.size() * 2 / 7 = 14 * 2 / 7 = 4,  dis = 4
58.   advance(d2, dis);
59.   copy(d1, d2, inserter(s5, s5.begin()));  // В s5 4 элемента
60.   copy(d2, d.end(), inserter(s6, s6.begin())); // В s6 10 эл-ов
61.   cout << "Первое множество: "; showT(s6);
62.   cout << "Второе множество: "; showT(s5);
63.   set_intersection(s6.begin(), s6.end(), s5.begin(), s5.end(),
63.       inserter(res1, res1.begin()));
64.   cout << "Наличие во 1-м множестве элементов из 2-го мн-ва: " << endl;
65.   cout << (res1.empty() ? "false" : "true") << endl << endl;
66.   if (!res1.empty())
67.       {
67.           cout << "Результат пересечения множеств:"; showT(res1);
68.       }
69.    }
```

Результат (обратите внимание, что во множествах `set` данные упорядочиваются автоматически):

```
==== объединение ====
Исходный список: 4 2 3 4 8 1 9 3 4 5 7 7
Первое множество: 1 2 3 4 8
Второе множество: 3 4 5 7 9
set_union: 1 2 3 4 5 7 8 9
```

```
==== включение ====
Исходный вектор: 7 1 8 3 1 2 3 4 5 7 8 0
Первое множество: 0 1 2 3 4 5 7 8
Второе множество: 1 3 7 8
Вхождение 2-го множества в 1-е:
true
```

```
==== пересечение ====
Исходная очередь: 2 3 7 8 0 1 2 0 5 7 6 3 4 4
Первое множество: 0 1 2 3 4 5 6 7
Второе множество: 2 3 7 8
Наличие в 1-м множестве элементов из 2-го множества:
true

Результат пересечения множеств:2 3 7
Для продолжения нажмите любую клавишу . . .
```

Комментарии

(8-14)	Описывается шаблонная функция **template <class A> void showT(A& cont)** для вывода данных контейнера в поток. Параметр функции – выводимый контейнер.
(23)	Создаются два итератора, которые указывают на начало списка: **l1 = l.begin(), l2 = l.begin();** Это нужно для последующего алгоритма **copy,** в котором первый параметр указывает на начало списка **(l1),** второй – на элемент в середине списка **(l2)**. Итератор **l2** модифицируется с помощью функции **advance()**.
(24)	С помощью метода **size()** определяется, сколько элементов содержится в контейнере (либо в списке, либо в векторе, либо в очереди). Далее произвольным образом делим контейнеры на 2 части, определяя смещение. Например, смещение **dis** для итератора **l2** вычисляется по формуле: **l.size() * 2 / 4 = 12 * 2 / 4 = 6, dis = 6.** Из начального списка выбираем 6 элементов. Необходимо заметить, что вы можете выбрать другие смещения и разделить контейнеры по-другому.
(26)	Применяется функция **advance(it, dis);** которая смещает итератор **it** на несколько позиций вправо **(dis > 0)** или влево **(dis < 0).**
(27-28)	Важно рассмотреть, как выполняется алгоритм **copy.** Стандартный вызов, например, такой: **copy(v.begin(), v.end(), s.begin());** В нашем случае необходимо разделить данные на две части и поместить их в разные множества **s1** и **s2**. **copy(l1, l2, inserter(s1, s1.begin()));** Копируется первая половина списка в **s1:** 4 2 3 4 8 1 – 6 элементов, но т к. **s1** – множество без дубликатов (не допускается повторение значений элементов), то будет скопировано 5 элементов и они будут упорядочены: 1 2 3 4 8. **copy(l2, L.end(), inserter(s2, s2.begin()));** Копируется вторая половина списка в **s2:** 9 3 4 5 7 7 – 6 элементов, но т к. **s2** (аналогично **s1**) – множество без дубликатов, то будет скопировано 5 элементов и они также будут упорядочены: 3 4 5 7 9.
(29-30)	**set_union(begin(s1), end(s1), begin(s2), end(s2), inserter(res, res.begin()));** – операция объединения во множество **res.**
(31-33)	С помощью функции **showT()** выводим данные на экран.
(34)	Операция **includes** выполняется аналогично п. 19 – 32, но здесь используется вектор. **includes** выдаёт логическое значение: входит второе множество в первое или нет.
(50)	Выполняется операция пересечения множеств **set_intersection,** результат помещается в **res1,** кроме этого, анализируется, не пусто ли это множество. Исходный контейнер – очередь, остальное выполняется аналогично п. 19 – 32.

Задания для самостоятельного выполнения

1. Дан вектор `V` с четным количеством элементов. Если какие-либо значения, содержащиеся во второй половине вектора, входят хотя бы один раз в его первую половину, то вывести `true`, иначе вывести `false`. Использовать алгоритм `set_intersection`, применив его к двум *множествам* (контейнерам типа `set`), созданным на основе вектора `V`.

2. Создать словарь с дубликатами для описания группы студентов (фамилия и год рождения). Фамилия – это ключевое поле. Ввести `7-10` записей, так, чтобы были одинаковые фамилии. Создать операции поиска, добавления и удаления в это множество.
3. Создать словарь Food_products, в котором ключом является название товара, а значением – цена за килограмм. Цена – целое число.
   a. Заполнить его несколькими значениями (5-6).
   b. Создать меню, в котором будут решаться следующие задачи:
      - Ввести название продукта и по нему найти цену.
      - Вывести на экран полный список продуктов и их цен.
      - Ввести название продукта, если такой есть, то вывести его цену, если нет – ввести в словарь информацию о продукте. После этого вывести на экран пополненный список продуктов с их ценами.

Контрольные вопросы

1. Какие контейнеры называются ассоциативными?
2. Какие структуры лежат в основе физической реализации ассоциативных контейнеров?
3. Как описываются ассоциативные контейнеры?
4. С какой целью в описании указывается способ упорядочивания элементов?
5. Назовите методы и свойства ассоциативных контейнеров.
6. Перечислите особенности множеств и словарей.
7. Перечислите особенности множеств и словарей с дубликатами.
8. Как определяется пара в STL?
9. Каким образом можно получить доступ к отдельным элементам пары?
10. Какие существуют алгоритмы работы с ассоциативными контейнерами как с множествами?

8. Битовые множества

Битовые множества в STL – это специальный класс-контейнер, который предназначен для хранения битовых значений. Рассматриваются основные понятия, особенности и примеры использования битовых множеств.

Определение bitset

Битовое множество **bitset** – класс для представления и обработки длинной последовательности битов (битовый массив). Последовательность битов часто применяется для компактного хранения флагов для наборов элементов или условий. Для `bitset` определены операции произвольного доступа, изменения отдельных битов и всего массива. Итераторы в `bitset` отсутствуют.

Шаблон битового множества определён в заголовочном файле **<bitset>.** Биты нумеруются с `0`. Размер фиксируется, когда объявляется объект `bitset.` Бит задан, если его значение равно `1` и сброшен, если его значение равно `0`.

Примеры создания битовых множеств:

```
bitset <100> b1;                        // сто нулей
bitset <16> b2 (0xf0f);                 // 0000111100001111
bitset <16> b3 ("0000111100001111");
bitset <5> b4 ("00110011", 3);          // 10011
bitset <3> b5 ("00110101", 1, 3);       // 011
```

В последнем примере первый параметр – строка из `"0"` и `"1".` Второй параметр – позиция начала битового множества, третий – количество символов.

Рассмотрим две небольшие задачи.

1. Зададим 32-битное слово, определённое так:

`bitset <32> b1("0000000000001011");` – старшие биты заполняются нулями. Необходимо в векторе типа `int` разместить числа, соответствующие значению `"1"` в битовом множестве.

```
#include <bitset>
#include <vector>
int bitset_massiv1()
{
	vector<int> V;
	bitset <32> b1("0000000000001011");
	vector<int>::iterator i;
	// Итераторы в bitset отсутствуют
	for (int j = 0; j != b1.size(); ++j)
	{
		if (b1[j] != 0)
			V.push_back(j);
	}
	cout << "bitset: " << b1 << endl;
	cout << "vector: ";
	for (i = V.begin(); i != V.end(); ++i)
		cout << *i << " ";
	cout << endl << endl;
	return 0;
}
```

Результат:

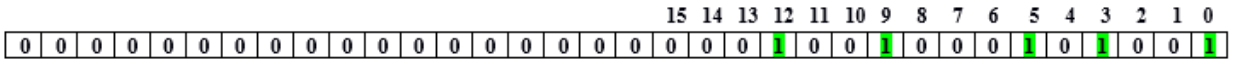
```
bitset: 00000000000000000000000000001011
vector: 0 1 3
```

2. Дано 32-битное слово:

```
bitset <32> b1("0001001000101001");
```

Необходимо в векторе типа `int` разместить числа, соответствующие значению `"1"` в битовом множестве и решить обратную задачу: числа в векторе перенести в битовое множество.

32-битное слово выглядит следующим образом:



Соответствующий ему вектор – `0, 3, 5, 9, 12.`

```
int bitset_massiv2()
{
	vector<int> V;
	bitset <32> b1("0001001000101001");
	bitset <32> b2;
	vector<int>::iterator i;
	for (int j = 0; j != b1.size(); ++j)
	{
		if (b1[j] != 0)
			V.push_back(j);
	}
	cout << "bitset: " << b1 << endl;
	cout << "vector: ";
	for (i = V.begin(); i != V.end(); ++i)
		cout << *i << " ";
	cout << endl;
	for (i = V.begin(); i != V.end(); ++i)
	{
			b2.set(*i);
	}
	cout << "bitset: " << b2 << endl << endl;
	return 0;
}
```

Результат:

```
bitset: 00000000000000000001001000101001
vector: 0 3 5 9 12
bitset: 00000000000000000001001000101001

Для продолжения нажмите любую клавишу . . .
```

Основные операции, выполняемые с битовыми множествами

Операция	Описание
`constructors()`	создает новое битовое множество
`operators()`	сравнивают и устанавливают битовые множества
`all()`	проверяет, что все биты установлены в `true`
`any()`	проверяет, чтобы хотя бы один бит установлен в `true`
`none()`	проверяет, что ни один из битов не был установлен в `true`
`count()`	возвращает количество битов, установленных в `true`
`flip()`	инвертирует биты, то есть меняет значение `true` на `false` (и наоборот) во всем массиве (или только в указанной позиции)
`reset()`	сбрасывает все биты (или только в указанной позиции) в значение `false`
`set()`	устанавливает все биты (или только в указанной позиции) в значение `true`

`test()`	возвращает значение данного бита
`to_string()`	строковое представление битового множества
`to_ullong()`	возвращает целочисленное представление битового множества
`size()`	возвращает количество битов, которые содержатся в массиве

Задания для самостоятельного выполнения

1. Дан вектор: `vector<int>,` в вектор внесены значения: `1, 5, 8, 17, 22, 30.` Числа из вектора перенести в битовое множество так, чтобы бит, соответствующий числу, получил значение `"1"`.
2. Пусть в магазине имеется 100 наиболее важных товаров, им соответствует множество битов. Номер бита соответствует номеру товара. Это своеобразная маска наличия товаров. Необходимо:
   - Создать структуру с тремя полями: номер товара, его название, булевская переменная, в которой будет отмечать наличие или отсутствие товара. Эта структура будет работать совместно с битовым множеством.
   - Написать операцию, заносящую `0` в соответствующий бит битового множества, когда товар закончился.
   - Написать операцию, которая при поступлении товара занесёт в соответствующий бит `1.`
   - Написать операцию внесения изменений в структуру при поступлении или отсутствии товара.
   - Используя структуру, выдать список названия всех товаров, которые отсутствуют в данный момент.

Контрольные вопросы

1. Что такое битовое множество?
2. Какие основные операции над битовыми множествами вы знаете?
3. Каким образом можно задать размер битового множества?
4. Каким образом можно использовать битовое множество при хранении и изменении информации о наборах данных?

9. Адаптеры контейнеров

В данном разделе изучаются специализированные контейнеры-адаптеры, которые применяются в различных задачах.

В библиотеке STL существует три адаптера контейнеров: стек (stack), очередь (queue) и очередь с приоритетами (priority_queue). Эти адаптеры не являются самостоятельными контейнерами, а реализованы на основе рассмотренных ранее последовательных контейнеров (вектора, двусторонней очереди и списка). Адаптеры используются довольно часто и обладают ограниченным функционалом по сравнению с базовыми контейнерами, на основе которых они созданы.

Адаптеры контейнеров не поддерживают итераторы.

Кроме адаптеров контейнеров в STL имеются адаптеры итераторов и адаптеры функций. Информацию об этих контейнерах можно посмотреть в приведённой литературе.

Стек

Стек – это динамическая структура данных – список, имеющая определённые правила поведения: все включения и исключения делаются на одном конце списка. Стек ещё называют списком типа LIFO (Last-In-First-Out – «последним включается – первым исключается»).

Для стека в качестве базовых контейнеров используются vector, list и deque, но функциональные возможности этих контейнеров будут ограничены.

Для стека допускаются два метода, изменяющие его размер:

- push(T) – добавить элемент в конец,
- pop() – удалить верхний элемента.

Кроме того, определены методы:

- empty() – проверить, пуст стек или нет,
- size() – найти количество элементов, т.е. определить размер стека,
- top() – получить доступ к последнему элементу, т.е. получить ссылку на элемент в вершине стека.

Для работы со стеком нужно подключить два заголовочных файла:
#include <stack> и **#include <vector>** (или **#include <list>**, или **#include <deque>**, в зависимости от того, какой базовый контейнер положен в основу стека).

Поскольку для стеков не применяются итераторы, чтобы просмотреть содержимое стека, нужно считать последний (на вершине стека) элемент с помощью метода top(), затем удалить этот верхний элемент, используя pop(), и проверить, не пуст ли стек – empty(). Всё это будем делать в цикле while.

```
#include <iostream>
#include <vector>
#include <stack>
void adapter_stack()
{
	stack<int, vector<int> > S;
	// или stack<int, list<int> > S;
	// или stack<int, deque<int> > S;
	S.push(5);
	S.push(10);
	S.push(15);
	S.push(20);
	while (!S.empty())
	{
		cout << "Количество элементов в стеке: " << S.size()
```

```
<< " Элемент на вершине: " << S.top() << endl;
            S.pop(); // Удаляем верхний элемент
      }
      cout << "Стек пуст" << endl;
}
```

Результат:

```
Количество элементов в стеке: 4 Элемент на вершине: 20
Количество элементов в стеке: 3 Элемент на вершине: 15
Количество элементов в стеке: 2 Элемент на вершине: 10
Количество элементов в стеке: 1 Элемент на вершине: 5
Стек пуст
```

Вместо вектора можно использовать список или двустороннюю очередь, результат будет тот же, для этого опишем стек таким образом:

```
stack<int, list<int> > S;
stack<int, deque<int> > S;
```

Рассмотрим задачу подсчёта баланса скобок, например, круглых скобок `()`. В этой задаче будем применять стек. В стеке нельзя получить доступ к произвольному элементу, не изменив его размер, поэтому будем использовать методы `empty()`, `top()` и `pop()`. Данные для задачи вводятся с клавиатуры в строку, затем передаются в стек и дальше работаем со стеком. Необходимо заметить, что, поскольку работа производится со стеком, *вводимые выражения обрабатываются с конца.*

```
1. #include <stack>
2. #include <string>
3. void adapter_stack1()
4. {
5. // Баланс круглых скобок
6. stack<char> br;
7. string str;
8. char c;
9. int a;   // Для круглых скобок ( )
10.      while (true)
11.      {
12.      a = 0;
13.      cout << "Введите выражение, содержащее скобки, или # для выхода:"
   << endl;
14.      cin >> str;
15.      auto is = str.begin();
16.      if (*is == '#') break;
17.      for (is; is != str.end(); ++is) br.push(*is);
18.      while (!br.empty())
19.           {
20.           c = br.top();
21.           //cout << c;
22.           switch (c)
23.               {
24.               case '(': a += 1; break;
25.               case ')': a -= 1; break;
26.               }
27.               if (a > 0) break;
28.               br.pop();
29.           }
30.           if (a == 0) cout << "Выражение корректно " << endl;
```

```
31.            else cout << "Нарушен баланс () " << endl;
32.      }
34.}
```

Результат:

```
Введите выражение, содержащее скобки, или # для выхода:
(a+b)*(d-r)
Выражение корректно
Введите выражение, содержащее скобки, или # для выхода:
((a/c-r)-n)-4
Выражение корректно
Введите выражение, содержащее скобки, или # для выхода:
)(c-d)
Нарушен баланс ()
Введите выражение, содержащее скобки, или # для выхода:
((d*f)
Нарушен баланс ()
Введите выражение, содержащее скобки, или # для выхода:
((s+k)-5*(r)
Нарушен баланс ()
Введите выражение, содержащее скобки, или # для выхода:
#
Для продолжения нажмите любую клавишу . . .
```

Комментарии

(1)	Для работы со стеком включаем в проект **#include <stack>**.
(6)	Определяем стек с типом данных **char: stack<char> br;**
(7)	**string str;** Вводимое выражение помещаем в **str.**
(8)	Каждый элемент стека заносим в переменную **c.**
(9)	Переменная **a** отвечает за подсчёт открывающих и закрывающих скобок ().
(10)	**while (true) {}** Создаём цикл для многократного выполнения программы (так называемый бесконечный цикл), выход из программы осуществляется при вводе символа **#.**
(12)	При каждом входе в цикл, при работе с новым выражением, переменная **a** обнуляется.
(14)	Считываем данные с клавиатуры в строку **str: cin >> str;**
(15)	Определяем переменную **is**, которая является итератором строкового контейнера: **auto is = str.begin();**
(16)	**if (*is == '#') break;** При вводе # – выход из бесконечного цикла.
(17)	**for (is; is != str.end(); is++) br.push(*is);** – пересылаем введенные данные в стек, применяя метод **push(*is).**
(18)	С помощью цикла **while (!br.empty())** будем считывать из стека данные, пока он не станет пустым.
(22)	В цикле применяем оператор выбора **switch (c),** в котором анализируются введённые данные. Выражение в операторе **switch** – это верхнее значение стека **c=br.top();**
(24)	Первоначально, **a = 0;** При анализе введённых данных, при вводе **‘(‘ a** увеличивается на **1.**
(25)	При вводе **‘)‘ a** уменьшается на **1.** При балансе скобок, **a = 0.**
(27)	**if (a > 0) break;** Эта ситуация возникает, когда в выражении сначала идёт закрывающая скобка, а затем – открывающая. Необходимо учитывать, что разбор идёт с конца выражения (стек).
(28)	Удаляем из стека верхний (проанализированный) элемент и переходим к разбору элемента, оказавшегося на вершине стека. Идём на цикл (18), проверяем, не пуст ли стек, и таким образом, анализируем все элементы в стеке.
(31)	Если **a == 0,** то "Выражение корректно".
(32)	В противном случае – "Нарушен баланс ()".

Очередь

Очередь – линейный список, в котором все включения производятся на одном конце списка, а все исключения делаются на другом его конце. Очередь иногда называют циклической памятью или списком типа FIFO (First-In-First-Out – «первым включается – первым исключается»).

Очередь не может быть построена на базе контейнра vector, поскольку у вектора отсутствует метод pop_front(). Для очереди базовыми контейнерами являются deque или list, по умолчанию применяется deque. Также как и стек, очередь ограничивает функционал базового контейнера.

Основные методы адаптера queue:

- front() – узнать значение первого элемента,
- back() – узнать значение последнего элемента,
- push(T) – добавить элемент в конец очереди,
- pop() – удалить первый элемент из очереди ,
- size() – найти количество элементов.

Чтобы использовать адаптер контейнеров queue необходимо подключить соответствующую библиотеку с помощью директивы **#include <queue>** (в случае использования базового контейнера list, подключаем также **#include <list>**).

Рассмотрим небольшую программу.

Создадим 3 очереди Q, Q1 и Q2, введём в Q числа 10, 15, 20. Затем Q1 и Q2 присвоим значения Q. На примере очереди Q посмотрим работу методов push(), empty(), size(), front() и pop(), в конце этой цепочки методов очередь становится пустой. Добавим в Q1 два новых элемента 30 и 3. Сравним очереди Q1 и Q2.

Для удобства сравнения выравним данные по последним элементам в очередях (см. табличку).

Q	Q1	Q2
	10	
	15	
10	20	10
15	30	15
20	3	20

Сравнение данных в очередях, также как и в стеках, происходит лексикографически. Сравнение начинаем с последних элементов 3 и 20, ‘3’ > ‘2’, следовательно, очередь Q1 > Q2. Если бы последние элементы были равны, сравнивали бы предпоследние и т.д.

```
#include <iostream>
#include <list>
#include <queue>
void adapter_queue()
{
	queue<int, list<int> > Q, Q1, Q2;
	Q.push(10);
	Q.push(15);
	Q.push(20);
	Q1 = Q;
	Q2 = Q;
	while (!Q.empty())
	{
		cout << "Количество элементов в очереди Q: " << Q.size() <<
		" Элемент в начале: " << Q.front() << endl;
		Q.pop(); // Удаляем первый элемент
	}
	cout << "Очередь Q пуста" << endl;
	Q1.push(3);
	Q1.push(30);
```

```
        if (Q1 < Q2) cout << "Очередь Q1 < Q2" << endl;
        else cout << "Очередь Q1 >= Q2" << endl;
        while (!Q1.empty())
        {
              cout << "Количество элементов в очереди Q1: " << Q1.size() <<
              " Элемент в конце: " << Q1.front() << endl;
              Q1.pop(); // Удаляем первый элемент
        }
        cout << "Очередь Q1 пуста" << endl;
}
```

Результат:

```
Количество элементов в очереди Q: 3 Элемент в начале: 10
Количество элементов в очереди Q: 2 Элемент в начале: 15
Количество элементов в очереди Q: 1 Элемент в начале: 20
Очередь Q пуста
Очередь Q1 >= Q2
Количество элементов в очереди Q1: 5 Элемент в конце: 10
Количество элементов в очереди Q1: 4 Элемент в конце: 15
Количество элементов в очереди Q1: 3 Элемент в конце: 20
Количество элементов в очереди Q1: 2 Элемент в конце: 30
Количество элементов в очереди Q1: 1 Элемент в конце: 3
Очередь Q1 пуста
```

Для задач нахождения максимальных и минимальных значений в очередях можно воспользоваться передачей элементов очереди в массив. Рассмотрим, как это можно сделать.

```
#include <queue>
#include <list>
void queue_massiv()
{
      queue<int, list<int> > Q;
      int x, n, max;
      cout << "Введите ряд чисел, за которыми следует 0:" << endl;
      while (cin >> x, x != 0)
            Q.push(x);
      n = Q.size();
      int* a = new int[n];
      for (int i = 0; i < n; ++i)
      {
            a[i] = Q.front();
            cout << a[i] << "  ";
            Q.pop(); // Удаляем первый элемент
      }
      max = a[0];
      for (int i = 0; i < n; ++i)
      {
            if (max < a[i])
                  max = a[i];
      }
      cout << endl << "Максимальный элемент в очереди: " << max << endl;
      delete[] a;
}
```

Результат:

```
Введите ряд чисел, за которыми следует 0:
12
23
51
9
0
12  23  51  9
Максимальный элемент в очереди: 51
Для продолжения нажмите любую клавишу . . .
```

Очередь с приоритетами

Очередь с приоритетами `priority_queue` является контейнером-адаптером, в котором первым в очереди всегда оказывается элемент с наибольшим значением, который можно удалить. Также как `stack` и `queue`, `priority_queue` ограничивает функционал базового контейнера. Для работы с очередью с приоритетами в качестве базовых контейнеров используются `vector` и `deque`, по умолчанию применяется vector. Поэтому подключаются директивы **`#include <queue>`**, а также **`#include <vector>`** (по умолчанию) или **`#include <deque>`**.

Основные методы `priority_queue`:

- `push(T)` – добавляет элемент в очередь на основе приоритета элемента из оператора operator<,
- `pop()` – удаляет самый большой элемент,
- `size()` – возвращает количество элементов.
- `empty()` – проверяет, является ли priority_queue пустой очередью,
- `top()` – осуществляет доступ к наибольшему элементу из оставшихся (или в соответствии с предикатом).

Рассмотрим программу `adapter_pr_queue()`. Сначала напишем шаблонную функцию печати `print_pr_queue(T& prq)`. В самой программе `adapter_pr_queue()` ввод данных осуществляется через вектор с помощью метода `push_back()`, затем проинициализируем этими данными очередь с приоритетами. Это позволит увидеть начальную последовательность ввода данных (на примере вектора) и затем последовательность в очереди с приоритетами (от большего к меньшему). Далее, будем использовать функциональный объект STL `greater()`, который отсортирует очередь от меньшего к большему.

Элемент добавляется в очередь в соответствии с его приоритетом. Очередь с приоритетами можно проинициализировать и с помощью массива.

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
template<typename T> void print_pr_queue(T& prq)
{
	// Шаблонная функция для вывода
	while (!prq.empty())
	{
		cout << prq.top() << " ";
		prq.pop();
	}
	cout << endl;
}
```

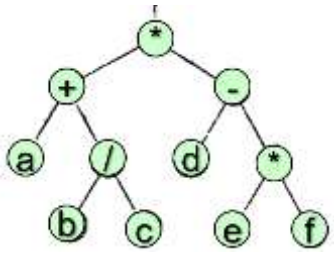
```
void adapter_pr_queue()
{
	vector <int> v;
	vector <int>::iterator iv;
	v.push_back(5);
	v.push_back(20);
	v.push_back(10);
	v.push_back(7);
	cout << "Значения в векторе: " << endl;
	for (iv = v.begin(); iv != v.end(); ++iv)
		cout << *iv << " ";
	cout << endl;
	// Определим очередь с приоритетами PQ1, инициализируем её
	// данными вектора
	priority_queue <int> PQ1(v.begin(), v.begin() + 4);
	cout << "Значения в очереди с приоритетами: " << endl;
	print_pr_queue(PQ1);
	// Определим очередь с приоритетами PQ2, также инициализируем
	// её данными вектора, используем greater<int>
	priority_queue<int, vector<int>, greater<int> > PQ2
		(v.begin(), v.begin() + 4);
	cout << "Значения в очереди с приоритетами (с greater): " << endl;
	print_pr_queue(PQ2);
	// Определим очередь с приоритетами PQ3 аналогично PQ2,
	// введём дополнительно 2 элемента
	priority_queue<int, vector<int>, greater<int> > PQ3
	(v.begin(), v.begin() + 4);
	PQ3.push(15); // Добавили ещё два элемента
	PQ3.push(1);
	cout << "Значения в очереди с приоритетами (с greater) "
		<< "после добавления двух элементов: " << endl;
	cout << "Всего элементов: "  << PQ3.size() << endl;
	print_pr_queue(PQ3);
	// Инициализация очереди с помощью массива
	int a[10] = { 2,5,9,0,3,8,1,10,0,12 };
	cout << "Значения в очереди с приоритетами: " << endl;
	priority_queue <int> PQ4(a, a + 10);
	print_pr_queue(PQ4);
}
```

Результат:

```
Значения в векторе:
5 20 10 7
Значения в очереди с приоритетами:
20 10 7 5
Значения в очереди с приоритетами (с greater):
5 7 10 20
Значения в очереди с приоритетами (с greater) после добавления двух элементов:
Всего элементов: 6
1 5 7 10 15 20
Значения в очереди с приоритетами:
12 10 9 8 5 3 2 1 0 0
Для продолжения нажмите любую клавишу . . .
```

Задания для самостоятельного выполнения

1. Найти максимальный элемент в стеке.
2. Найти минимальный элемент в очереди.
3. Написать программу ввода серии чисел и вывода их в обратном порядке.
4. Пусть имеется арифметическое выражение, постоим по нему бинарное дерево и обойдём его снизу-вверх (обратный обход). В результате получим постфиксную форму. Предположим, что у вас уже построена постфиксная форма и в неё введены числа, считываем её в строку.
   Будем использовать стек. Читаем операнды слева направо, при появлении знака операции, над предыдущими двумя операндами производится действие, соответствующее знаку операции. Результат операции заносится в вершину стека. Таким образом обрабатываем всю строку.
   Задача: вычислить значение арифметического выражения.
   Например: арифметическое выражение `(a+b/c)*(d-e*f).`
   Бинарное дерево:



   Обратный обход: `a b c / + d e f * - *`
   Вместо букв подставьте числа, например: `5 6 3 / + 4 7 2 * - *.`

Контрольные вопросы

1. Что такое адаптеры контейнеров?
2. Какие виды адаптеров контейнеров вы знаете?
3. Какие базовые контейнеры могут быть использованя для стека, очереди и очереди с приоритетами?
4. Назовите особенности адаптера «стек» и его основные операции.
5. Назовите особенности адаптера «очередь» и его основные операции.
6. Назовите особенности адаптера «очередь с приоритетами» и его основные операции.
7. Поскольку для стеков, очередей и очередей с приоритетами не применяются итераторы, каким образом можно посмотреть их содержимое?
8. Что означает линейныt списки типа `LIFO` и `FIFO`?
9. Каким образом, не применяя операцию `push,` можно внести данные в адаптеры?
10. Какие различия вы можете назвать между тремя изученными адаптерами контейнеров (стек, очередь и очередь с приоритетами)?

Список литературы и Интернет-источников

1. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт — 2-е изд., испр. СПб.: Невский Диалект, 2001. — 352 с.: ил.
2. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. — М.: МЦМНО, 2001. — 960 с.: ил.
3. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. — 3-е изд., испр. М.: Вильямс, 2019. — 1328 с.: ил.
4. Страуструп, Б. Программирование. Принципы и практика использования C++ / Б. Страуструп — 2-е изд., испр. М.: Вильямс, 2016. — 1329 с.: ил.
5. Седжвик, Р Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск / Р. Седжвик. — К.: Издательство «ДиаСофт», — 688 с.: ил.
6. Уайс, М. А. Организация структур данных и решение задач на C++ / М.А. Уайс — М.: ЭКОМ Паблишерз, 2008. — 896 с.: ил.
7. Аммерааль, Л. STL для программистов на C++ / Л. Аммерааль. — М.: ДМК Пресс, 2011. — 242 с.: ил.
8. Мейерс, С. Эффективное использование STL / С. Мейерс. — СПб.: Питер, 2003. — 224 с.: ил.
9. Мюссер, Д. C++ и STL: справочное руководство / Д. Мюссер, Ж. Дердж, А. Сейни. — 2-е издание. — М.: Вильямс, 2010. — 432 с.: ил.
10. Степенов, А. Руководство по стандартной библиотеке шаблонов (STL) / А. Степенов, М. Ли. — М.: Технический университет, 1999, — 43 с.: ил.
11. Головиц, Я. C++17 STL. Стандартная библиотека шаблонов / Я. Головиц. — Санкт-Петербург: Питер, 2018. — 432 с.: ил.
12. Москвин, П.В. Азбука STL / П.В. Москвин. — М.: Горячая линия — Телеком, 2012. — 262 с.: ил.
13. Остерн, М. Обобщенное программирование и STL / М. Остерн. — Санкт-Петербург: Невский диалект, 2004. — 544 с.: ил.
14. Коллинз, У. Дж. Структуры данных и стандартная библиотека шаблонов / У. Коллинз. — М.: Бином-Пресс, 2004. — 624 с.: ил.
15. Контейнеры стандартной библиотеки C++ | Microsoft Docs
16. Библиотека стандартных шаблонов (STL) (cppstudio.com)
17. Абрамян, М.Э. Электронный задачник по программированию. — Южный федеральный университет, 1998–2021. Programming Taskbook (ptaskbook.com).

Приложение А Некоторые заголовочные файлы STL

Контейнерные классы

Заголовочный файл	Описание
<bitset>	Битовое множество
<deque>	Двусторонняя очередь
<list>	Двусвязный список
<map>	Словарь, словарь с дубликатами
<queue>	Очередь, очередь с приоритетами
<set>	Множество, множество с дубликатами
<stack>	Стек
<vector>	Одномерный массив

Заголовочные файлы для STL

Заголовочный файл	Описание
<algorithm>	Алгоритмы
<functional>	Функциональные объекты
<iterator>	Итераторы
<numeric>	Числовые операции
<utility>	Операторы и пары

Методы включения и удаления элементов в контейнерах

Метод	Описание
insert(p, x)	Добавление **x** перед элементом, на который указывает **p.**
push_back(x)	Добавление **x** в конец.
push_front(x)	Добавление нового нулевого элемента (только для списков и очередей).
clear()	Удаление всех элементов.
erase(p)	Удаление элемента в позиции **p.**
pop_back()	Удаление последнего элемента.
pop_front()	Удаление нулевого элемента (только для списков и очередей с двумя концами).

Приложение Б Алгоритмы библиотеки STL

В таблице указаны алгоритмы, использованные в пособии.

Алгоритм	Описание
Алгоритмы определены в заголовочном файле <algorithm>	
Немодифицирующие алгоритмы	
count	Возвращает количество элементов в последовательности.
count_if	Возвращает количество элементов в последовательности, значения которых соответствуют заданному условию.
find	Возвращает позицию первого вхождения указанного элемента.
find_if	Возвращает позицию первого элемента в последовательности, где элемент удовлетворяет заданному условию.
Модифицирующие алгоритмы	
copy	Копирует значения из исходной последовательности в последовательность назначения.
replace	Заменяет элементы в последовательности, которые соответствуют заданному значению, новым значением.
replace_if	Аналогично replace(), но при выполнении заданного условия.
remove	Удаляет элементы с указанными значениями из заданной последовательности, не нарушая порядок оставшихся элементов.
reverse	Меняет порядок следования элементов на обратный.
swap	Меняет местами значения двух элементов, присваивая содержимое первого элемента второму элементу, а содержимое второго – первому.
Сортировка	
nth_element	Разделяет последовательность элементов, так, что элементы, меньшие, чем тот, на который указывает итератор `nth`, оказываются слева от него, а все большие – справа.
merge	Объединяет элементы из двух отсортированных исходных последовательностей в одну отсортированную последовательность.
partial_sort	Частичная сортировка – упорядочивает заданное число элементов, остальные остаются неотсортированными. Сортировка производится в порядке возрастания или согласно критерию упорядочивания, заданному предикатом.
sort	Сортирует элементы в указанной последовательности в возрастающем порядке или согласно критерию упорядочивания, заданному предикатом.
stable_sort	Сортирует элементы в последовательности по возрастанию или согласно критерию упорядочивания и сохраняет относительный порядок одинаковых по значению элементов.
Работа с множествами	
includes	Проверяет, входит ли одно множество элементов во второе множество.
set_difference	Строит отсортированную последовательность из элементов, имеющихся в одной последовательности, но отсутствующих во второй.
set_intersection	Производит пересечение двух множеств в одно упорядоченное множество; критерий порядка сортировки может быть указан предикатом.

set_union	Объединяет все элементы, входящие хотя бы одно из двух множеств, в одно множество; критерий упорядочивания может быть указан предикатом.
Численный алгоритм (заголовочный файл <numeric>)	
accumulate	Выполняет различную обработку данных, возможно с использованием функциональных объектов.

Приложение В Порядок выполнения и защиты практических работ

Перед проведением практических работ студентам даётся теоретический материал и методические указания по выполнению работ.

Для получения зачёта по каждой работе студент представляет преподавателю реализацию предложенных в работе заданий в форме проекта и результата выполнения программы, а также демонстрирует работоспособность своей программы. Во время собеседования студент обязан проявить знания по цели работы, теоретическому материалу, методам выполнения каждого этапа работы, ответить на вопросы преподавателя.

Дополнительные материалы, содержащие теоретические сведения и практические задания, даются студентам в виде электронных файлов.

Учебное издание

Кульман Татьяна Николаевна

Использование стандартной библиотеки шаблонов STL в программировании структур и алгоритмов обработки данных

УЧЕБНОЕ ПОСОБИЕ

Редактор Ю. С. Цепилова
Технический редактор Ю. С. Цепилова
Компьютерная верстка Ю. С. Цепилова
Корректор Ю. С. Цепилова

Подписано в печать. Формат 60×90/8. Усл. печ. л. 9,87.
Тираж 35 экз. Заказ № 18.

ГБОУ ВО МО «Университет «Дубна»
141980, г. Дубна Московской обл., ул. Университетская, 19