

## **Практическая работа. Последовательные контейнеры.**

Тема: Последовательные контейнеры библиотеки STL.

*Цель: Сформировать практические навыки описания и использования последовательных контейнеров для обработки данных.*

### **Задание**

1. Описать шаблоны функций для выполнения следующих задач.
  1. переставить элементы одномерного массива в обратном порядке;
  2. выполнить циклический сдвиг одномерного массива на одну позицию влево;
  3. выполнить циклический сдвиг одномерного массива на одну позицию вправо;
  4. выполнить циклический сдвиг одномерного массива на заданное число позиций в заданном направлении;
  5. переставить местами два заданных элемента массива;
  6. нахождения минимума из двух значений;
  7. нахождения максимума из трёх значений;
  8. упорядочения трёх значений  $x, y, z$  в порядке  $x \leq y \leq z$ ;
2. Описать шаблоны классов (функциональный объект) для выполнения следующих задач.
  1. определение принадлежности значения заданному диапазону:  $x < a$ ;
  2. определение принадлежности значения заданному диапазону:  $x \geq a$ ;
  3. определение принадлежности значения заданному диапазону:  $x < a$  и  $x > b$ ;
  4. определение принадлежности значения заданному диапазону:  $x \leq a$  и  $x \geq b$ ;
3. Простейшие контейнеры - массивы. Применение алгоритмов STL для обработки объектов предопределённых типов, организованных в массивы в режиме консольного приложения.
  1. алгоритм `sort()`;
  2. алгоритм `reverse()`;
  3. алгоритм `find()`;
  4. алгоритм `sort()`;
  5. алгоритм `count_if()`;
4. Простейшие контейнеры - массивы. Применение алгоритмов STL для обработки объектов пользовательских типов, организованных в массивы.
  1. Отсортировать объекты класса комплексные числа (`Cmpl`) по модулю в порядке убывания с помощью алгоритма `sort()`.
  2. Переставить объекты класса комплексные числа (`Cmpl`) в обратном порядке с помощью алгоритма `reverse()`.

3. Отсортировать объекты класса комплексные числа (Cmpl) по вещественной части в порядке убывания с помощью алгоритма `sort()` и функционального объекта.
  4. Отсортировать объекты класса комплексные числа (Cmpl) по мнимой части в порядке возрастания с помощью алгоритма `sort()` и функционального объекта.
  5. Отыскать объект класса комплексные числа (Cmpl) по значению с помощью алгоритма `find()` и перегруженной операции `==`.
  6. Отыскать объект класса комплексные числа (Cmpl) по значению с помощью алгоритма `find()` и перегруженной операции `==`.
  7. Отыскать объект класса комплексные числа (Cmpl), удовлетворяющий условию, с помощью алгоритма `find_if` и функционального объекта.
5. Последовательный контейнер - `vector` . Применение алгоритмов STL для обработки объектов пользовательских типов, организованных в вектора.
1. Отсортировать объекты класса комплексные числа (Cmpl) по модулю в порядке убывания с помощью алгоритма `sort()`.
  2. Переставить объекты класса комплексные числа (Cmpl) в обратном порядке с помощью алгоритма `reverse()`.
  3. Отсортировать объекты класса комплексные числа (Cmpl) по вещественной части в порядке убывания с помощью алгоритма `sort()` и функционального объекта.
  4. Отсортировать объекты класса комплексные числа (Cmpl) по мнимой части в порядке возрастания с помощью алгоритма `sort()` и функционального объекта.
  5. Отыскать объект класса комплексные числа (Cmpl) по значению с помощью алгоритма `find()` и перегруженной операции.
  6. Отыскать объект класса комплексные числа (Cmpl) по значению с помощью алгоритма `find()` и перегруженной операции.
  7. Отыскать объект класса комплексные числа (Cmpl), удовлетворяющий условию, с помощью алгоритма `find_if` и функционального объекта.
6. Последовательный контейнер - `deque` . Применение алгоритмов STL для обработки объектов пользовательских типов, организованных в деки.
1. Отсортировать объекты класса комплексные числа (Cmpl) по модулю в порядке убывания с помощью алгоритма `sort()`.
  2. Вычислить сумму объектов класса комплексные числа (Cmpl) в деке с помощью алгоритма `accumulate()` и функционального объекта `plus<>()`.
  3. Отсортировать объекты класса комплексные числа (Cmpl) по вещественной части в порядке убывания с помощью алгоритма `sort()` и функционального объекта.

4. Отсортировать объекты класса комплексные числа (Cmpl) по мнимой части в порядке возрастания с помощью алгоритма sort() и функционального объекта.
  5. Отыскать объект класса комплексные числа (Cmpl) по значению с помощью алгоритма find() и перегруженной операции ==.
  6. Отыскать объект класса комплексные числа (Cmpl) по значению с помощью алгоритма find() и перегруженной операции ==.
  7. Отыскать объект класса комплексные числа (Cmpl), удовлетворяющий условию, с помощью алгоритма find\_if() и функционального объекта.
7. Последовательный контейнер - list . Применение алгоритмов STL для обработки объектов пользовательских типов, организованных в списки.
1. Отсортировать объекты класса комплексные числа (Cmpl) по модулю в порядке убывания с помощью алгоритма sort().
  2. Переставить объекты класса комплексные числа (Cmpl) в обратном порядке с помощью алгоритма reverse().
  3. Отсортировать объекты класса комплексные числа (Cmpl) по вещественной части в порядке убывания с помощью алгоритма sort() и функционального объекта.
  4. Отсортировать объекты класса комплексные числа (Cmpl) по мнимой части в порядке возрастания с помощью алгоритма sort() и функционального объекта.
  5. Отыскать объект класса комплексные числа (Cmpl) по значению с помощью алгоритма find() и перегруженной операции.
  6. Посчитать количество объектов класса комплексные числа (Cmpl) находящихся в заданном диапазоне с помощью алгоритма count\_if() и перегруженной операции +.
  7. Отыскать объект класса комплексные числа (Cmpl), удовлетворяющий условию, с помощью алгоритма find\_if() и функционального объекта.

## Теоретические сведения

### Шаблоны функций

Шаблоны позволяют вам давать обобщенные, в смысле произвольно используемых типов, определения классов и функций. Эти определения затем могут служить компилятору основой для классов или функций, создаваемых для конкретного типа данных. Как следствие, шаблоны являются эффективным способом реализации процедур, которые раньше обычно переписывались много раз для данных различного типа.

### Параметризованные типы

Шаблоны C++ часто называют параметризованными типами. Они позволяют вам компилировать новые классы или функции, задавая типы в качестве параметров.

### Шаблоны функций

Шаблон функции представляет собой обобщенное определение, из которого компилятор может автоматически генерировать код (создать представителя) функции.

### **Шаблон функции: синтаксис**

Синтаксис шаблона функции имеет следующий вид:

```
template <список_аргументов_шаблона>
возвр_тип имя_функции (параметры...)
{
// Тело функции
}
```

За ключевым словом `template` следуют один или несколько аргументов заключенных в угловые скобки и отделенных друг от друга запятыми! Каждый аргумент состоит из ключевого слова `class` и идентификатора, обозначающего тип. Затем следует определение функции. Оно похоже на обычное определение функции, за исключением того, что один или несколько параметров используют типы, специфицированные в списке аргументов шаблона.

### **Определение шаблонов функции. Пример**

```
template <class T>
void Swap( T t[], int indx1, int indx2 )
{
    T tmp = t[indx1];
    t[indx1] = t[indx2];
    t[indx2] = tmp;
}
```

### **Шаблон класса**

Шаблон классов представляет собой класс, в котором определены данные и методы, но фактический тип данных задаётся в качестве параметра при создании объектов класса. Шаблоны дают возможность использовать один и тот же код, который позволяет компилятору автоматизировать процесс реализации типа. Для шаблонных классов характерными являются следующие свойства:

- Шаблон позволяет передавать в класс один или несколько типов в виде параметров. Передаваемым типом может быть любой базовый или производный тип, включая тип `class`.
- Параметрами шаблона могут быть не только типы, но и константные выражения.
- Объявление шаблона должно быть только глобальным.
- Статические члены-данные специфичны для каждого реализованного шаблона.

## Шаблон класса: синтаксис

Синтаксис шаблона класса имеет следующий вид:

```
template <список_аргументов_шаблона>
class имя_класса
{
// Тело класса
};
```

За ключевым словом `template` следуют один или несколько аргументов, заключенных в угловые скобки и отделяемых друг от друга запятыми. Каждый аргумент является:

- либо именем типа, за которым следует идентификатор,
- либо ключевым словом `class`, за которым следует идентификатор, обозначающий **параметризирующий** тип.

Затем следует определение класса. Оно аналогично определению обычного класса, за исключением того, что использует список аргументов шаблона.

Например, шаблон класса с двумя параметрами может выглядеть так

```
template <class T, float R> //T-параметр типа,
//R-нетиповой параметр, константа типа float
class S
{
// Тело класса
};
```

В описании класса принято называть доступный раздел (`public`) интерфейсом, а закрытые разделы (`private`, `protected`) – реализацией.

```
class S
{
public:
//интерфейс
private:
//реализация
};
```

### Параметры типа и нетиповые параметры

Параметры шаблона, состоящие из ключевого слова `class` или `typename` и следующего за ним идентификатора, часто называют **параметрами типа**.

Другими словами, они информируют компилятор, что шаблон предполагает тип в качестве аргумента. Параметры шаблона, состоящие из имени типа и идентификатора, иногда называют **нетиповыми**. Они информируют компилятор, что в качестве аргумента предполагается константа типа.

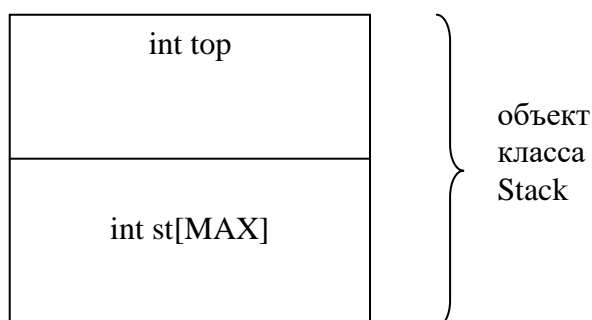
### Шаблоны классов. Пример

Шаблонный принцип можно расширить и на классы. В этом случае шаблоны обычно используются, когда класс является хранилищем данных (хороший тому пример - «Стандартная библиотека шаблонов (STL)»).

Стеки и связанные списки — тоже типичные примеры классов-хранилищ данных. Класс Stack приведённый ниже, например, может хранить только значения типа `int`. Вот немного сокращённая версия этого класса:

```
class Stack
{
public:
    //конструктор
    Stack();
    //int-аргумент
    void push(int var);
    //возвращает значение int
    int pop();
private:
    //целочисленный массив
    int st[MAX];
    //Индекс вершины стека
    int top;
};
```

Объект класса Stack в памяти представлен двумя полями



Если бы мы захотели хранить значения типа `long`, пришлось бы написать абсолютно новое определение класса:

```
class LongStack
{
public:
```

```

LongStack() ;//конструктор
void push(long var); // long -аргумент
long pop(); //возвращает значение long
private:
    long st[MAX]; //целочисленный массив
    int top; //Индекс вершины стека
};

```

Подобным же образом нам пришлось бы создавать классы для хранения данных каждого типа, если бы не шаблоны классов, которые и тут приходят к нам на помощь. С их помощью можно написать такую спецификацию класса, которая сохранит все, что попросят, без лишних вопросов. Создадим шаблон классов Stack для демонстрации возможностей шаблонов классов.

Спецификация и реализация шаблона классов при отдельной компиляции обязательно должны находиться в одном файле. Чтобы можно было компилировать шаблон, этот файл должен иметь расширение .cpp. Файл с шаблоном включают в программу с помощью директивы препроцессора #include.

Листинг 3. Программа TEMPLSTAK

```

//-----
#include "stdafx.h"
#include <windows.h>
#include <iostream>

using namespace std;

const int MAX = 100;
////////////////////////////////////
template <class Type>
class Stack
{
public:
    Stack() //конструктор
    { top = -1; }
    void push(Type var) //занести число в стек
    { st[++top] = var; }
    Type pop()//вынуть число из стека
    { return st[top--];}
private:
    Type st[MAX]; //стек: массив любого типа
    int top; //индекс вершины стека
};
////////////////////////////////////

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);

```

```

SetConsoleOutputCP(1251);
Stack<float> s1;//s1 - объект класса Stack<float>
s1.push(1111.1F);//занести 3 значения float
s1.push(2222.2F);
s1.push(3333.3F);
cout << "1: " << s1.pop() << endl;//вытолкнуть 3 значения float
cout << "2: " << s1.pop() << endl;
cout << "3: " << s1.pop() << endl;
Stack<long> s2;//s2 - объект класса Stack<long>
s2.push(123123123L);//занести 3 значения long
s2.push(234234234L);
s2.push(345345345L);
cout << "1: " << s2.pop() << endl;//вытолкнуть 3 значения long
cout << "2: " << s2.pop() << endl;
cout << "3: " << s2.pop() << endl;
system("pause");
return 0;
}
//-----

```

Здесь Stack является шаблоном классов. Идея шаблонов классов во многом сходна с идеей шаблонов функций. Ключевые слова `template` и `class` Stack говорят о том, что весь класс будет шаблоном.

## Библиотека STL

Классы C++ предоставляют прекрасный механизм для создания библиотеки структур данных. В прошлом производители компиляторов и разные сторонние разработчики ПО предлагали на рынке библиотеки классов-контейнеров для хранения и обработки данных. Теперь же в стандарт C++ входит собственная встроенная библиотека классов-контейнеров. Она называется Стандартной библиотекой шаблонов (в дальнейшем мы будем употреблять сокращение STL (<http://www.rsdn.ru/article/cpp/stl.xml>)) и разработана Александром Степановым и Менг Ли из фирмы Hewlett Packard. STL — это часть Стандартной библиотеки классов C++, которая может использоваться для хранения и обработки данных.

Контейнерные классы — это классы, предназначенные для хранения данных, организованных определенным образом. Примерами контейнеров могут служить массивы, линейные списки или стеки. Для каждого типа контейнера определены методы для работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере, поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Эта возможность реализована с помощью шаблонов классов, поэтому часть библиотеки C++, в которую входят контейнерные классы, а также алгоритмы и итераторы, о которых будет рассказано в следующих разделах,



называют стандартной библиотекой шаблонов (STL — Standard Template Library).

Использование контейнеров позволяет значительно повысить

- надежность программ,
- их переносимость,
- универсальность,
- а также уменьшить сроки их разработки.

Естественно, эти преимущества не даются даром: универсальность и безопасность использования контейнерных классов не могут не отражаться на быстродействии программы. Снижение быстродействия в зависимости от реализации компилятора может быть весьма значительным. Кроме того, для эффективного использования контейнеров требуется затратить усилия на вдумчивое освоение библиотеки.

В данной лекции описывается принцип построения STL и работа с ней. Тема большая и сложная, и мы не будем описывать все, что входит в библиотеку. Мы ограничимся здесь тем, что представим STL и приведем примеры наиболее часто используемых алгоритмов и контейнеров.

### ***Структура стандартной библиотеки шаблонов (STL)***

В библиотеке выделяют пять основных компонентов:

1. Контейнер (англ. container) — хранение набора объектов в памяти.
2. Итератор (англ. iterator) — обеспечение средств доступа к содержимому контейнера.
3. Алгоритм (англ. algorithm) — определение вычислительной процедуры.
4. Адаптер (англ. adaptor) — адаптация компонентов для обеспечения различного интерфейса.
5. Функциональный объект (англ. functor) — сокрытие функции в объекте для использования другими компонентами.

Разделение позволяет уменьшить количество компонентов. Например, вместо написания отдельной функции поиска элемента для каждого типа контейнера обеспечивается единственная версия, которая работает с каждым из них, пока соблюдаются основные требования.

Три наиболее важные компонента STL — это контейнеры, алгоритмы и итераторы.

**Контейнер** — это способ организации хранения данных. Вам, наверняка, уже встречались некоторые контейнеры, такие, как стек, связный список, очередь. Еще один контейнер — это массив, но он настолько тривиален и популярен, что встроен в C++ и большинство других языков программирования. Контейнеры бывают самые разнообразные, и в STL включены наиболее полезные из них. Контейнеры STL подключаются к

программе с помощью шаблонов классов, а значит, можно легко изменить тип хранимых в них данных.

Под **алгоритмами** в STL понимают процедуры, применяемые к контейнерам для обработки данных содержащихся в них различными способами. Например, есть алгоритмы сортировки, копирования, поиска и объединения. Алгоритмы представлены в STL в виде шаблонов функций. Однако они не являются методами классов-контейнеров. Наоборот, это совершенно независимые функции. На самом деле, одной из самых привлекательных черт STL является универсальность ее алгоритмов. Их можно использовать не только для объектов классов-контейнеров, но и для обычных массивов и даже для собственных контейнеров. (Контейнеры, тем не менее, содержат методы для выполнения некоторых специфических задач.)

**Итераторы** — это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать, как обычные указатели, и они будут ссылаться последовательно на все элементы контейнера. Итераторы — ключевая часть всего STL, поскольку они связывают алгоритмы с контейнерами. Их можно представить себе в виде кабеля, связывающего колонки вашей стереосистемы или компьютер с его периферией.

На рис. 1 показаны три компонента STL. В этом разделе мы обсудим их более детально.

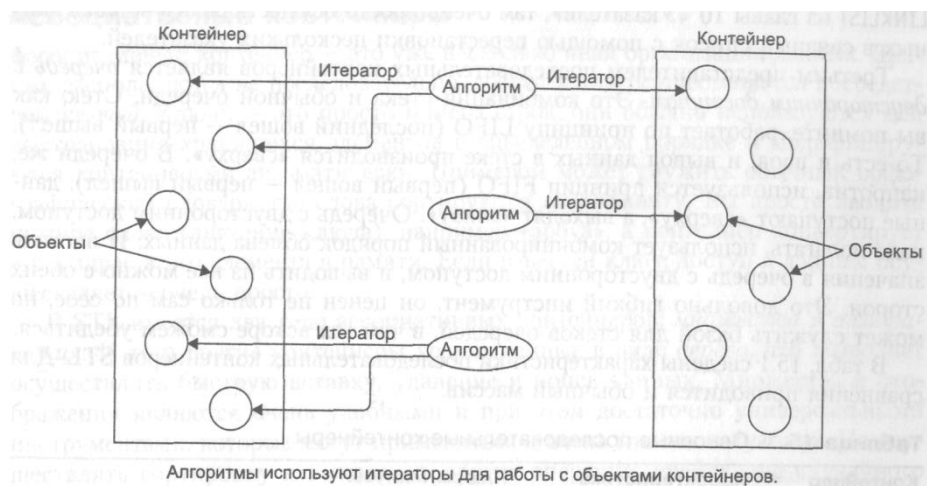


Рис. 1. Контейнеры, алгоритмы и итераторы.

Компоненты STL определяются в 13 заголовочных файлах:

algorithm	deque	functional	iterator	list	map
memory	numeric	queue	set	stack	utility
				vector	

## Примеры выполнения заданий.

Задание 1.1. Шаблон функции для перестановки элементов одномерного массива в обратном порядке.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <sstream>

using namespace std;

//Перестановка элементов массива в обратном порядке.
template <class t>
void VecSwap(t v[], int n)
{
    t temp;
    for (int i = 0; i < n / 2; i++)
    {
        temp = v[i];
        v[i] = v[n - i - 1];
        v[n - i - 1] = temp;
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    //Массив целых переменных.
    int v[5] = { 1, 2, 3, 4, 5 };
    VecSwap(v, 5);
    for (int i = 0; i < 5; i++) cout << v[i] << ", ";
    cout << endl;
    //Массив вещественных переменных.
    double vd[5] = { 1, 2, 3, 4, 5 };
    VecSwap(vd, 5);
    for (int i = 0; i < 5; i++) cout << vd[i] << ", ";
    cout << endl;
    return 0;
}
```

```
5, 4, 3, 2, 1,
5, 4, 3, 2, 1,
Для продолжения нажмите любую клавишу . . .
```

Задание 1.6. Шаблон функции для нахождения минимума из двух значений.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <sstream>

using namespace std;

//Шаблон функций. Поиск минимума двух значений.
template <class t>
t MinXY(t x, t y)
{
    if (x < y) return x;
    else return y;
}

int _tmain(int argc, _TCHAR* argv[])
{
    //Целые переменные.
    int x = 1;
    int y = 7;
    cout << MinXY(x, y) << endl; //Создание функции по шаблону.
    //Вещественные переменные.
    double v = 5, w = 3;
    cout << MinXY(v, w) << endl; //Создание функции по шаблону.
    return 0;
}
```

```
1
3
Для продолжения нажмите любую клавишу . . .
```

Задание 2.1. Шаблон классов для определения принадлежности значения заданному диапазону:  $x < a$ .

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <sstream>

using namespace std;

//Шаблон класса. Принадлежность значения //диапазону.
template <class t>
class Diapazon
{
public:
    bool operator()(t x, t a){ return x < a; }
};

int _tmain(int argc, _TCHAR* argv[])
{
    //Целые переменные.
    int x = 1;
    int y = 7;
    cout << Diapazon<int>()(x, y) << endl;
    //Вещественные переменные.
    Diapazon<double> d;
    double v = 5, w = 3;
    cout << d.operator()(v, w) << endl;
    return 0;
}
1
0
Для продолжения нажмите любую клавишу . . . _
```

Задание 3.1. Сортировка массива с помощью алгоритма sort().

```
#include "stdafx.h"
#include <iostream>
#include <algorithm> //Алгоритмы библиотеки STL.

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    int m[5] = { 2, 1, 5, 7, 3};
    //Сортировка массива.
    sort(m, m + 5);
    //Вывод массива на экран.
    for (int i = 0; i < 5; i++)
        cout << m[i] << ", ";
    cout << endl;
    //Вывод массива на экран.
    for each (int val in m)
        cout << val << ", ";
    cout << endl;
    return 0;
}
```

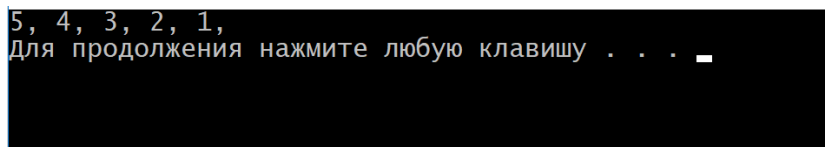
```
1, 2, 3, 5, 7,
1, 2, 3, 5, 7,
Для продолжения нажмите любую клавишу . . .
```

Задание 3.2. Перестановка массива с помощью алгоритма reverse().

```
#include "stdafx.h"
#include <iostream>
#include <algorithm> //Алгоритмы библиотеки STL.

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    int m[5] = { 1, 2, 3, 4, 5 };
    //Перестановка массива в обратном порядке.
    reverse(m, m + 5);
    for (int i = 0; i < 5; i++)
        cout << m[i] << ", ";
    cout << endl;
    return 0;
}
```



```
5, 4, 3, 2, 1,
Для продолжения нажмите любую клавишу . . . _
```

Задание 3.3. Поиск заданного элемента массива с помощью алгоритма find(). Алгоритмы связываются с контейнерами с помощью итераторов. В случае массива итератором является указатель на элемент массива с заданным индексом.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm> //Алгоритмы библиотеки STL.

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    int m[5] = { 2, 1, 5, 7, 3 };
    //Отыскиваем позицию заданного значения в массиве.
    int val = 5;
    int *p = find(m, m + 5, val);
    if (p != nullptr)
        cout << "Индекс искомого значения в массиве: " << p - m <<
endl;
    return 0;
}
```

```
Индекс искомого значения в массиве: 2
Для продолжения нажмите любую клавишу . . .
```



Задание 3.4. Поиск позиции элемента массива, удовлетворяющего условию с помощью алгоритма `find_if()`.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm> //Алгоритмы библиотеки STL.

using namespace std;
//Функциональный объект.
bool diapazon(int a)
{
    return (a <= 6) & (a >= 3);
}

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    int m[5] = { 2, 1, 5, 7, 3 };
    //Находим позицию элемента массива, удовлетворяющего условию.
    int *p = find_if(m, m + 5, diapazon);
    if (p != nullptr)
        cout << "Индекс искомого значения в массиве: " << p - m <<
endl;
    return 0;
}
```

```
Индекс искомого значения в массиве: 2
Для продолжения нажмите любую клавишу . . . _
```

Задание 3.5. Подсчёт числа значений массива, удовлетворяющих условию, с помощью алгоритма `count_if()`.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm> //Алгоритмы библиотеки STL.

using namespace std;
//Функциональный объект.
bool diapazon(int a)
{
    return (a <= 6) & (a >= 3);
}

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    int m[5] = { 2, 1, 5, 7, 3 };
    //Подсчитываем число значений массива, удовлетворяющих условию.
    int count = count_if(m, m + 5, diapazon);
    cout << count << " значения в заданном диапазоне " << endl;
    return 0;
}
```

```
2 значения в заданном диапазоне
Для продолжения нажмите любую клавишу . . .
```

Задание 4.1. Сортировка массива объектов класса Cmpl с помощью алгоритма sort() и перегруженной операции > по убыванию.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <sstream>
#include <functional>

using namespace std;

//Класс комплексные числа.
class Cmpl
{
    double re;
    double im;
public:
    string ToString()
    {
        string a;
        ostringstream os;
        os << re << " *i " << im;
        return os.str();
    }
    Cmpl(double r = 0, double i = 0): re(r), im(i)
    { }
    bool operator> (Cmpl b) const
    { return (re*re + im*im) > (b.re*b.re + b.im*b.im); }
};

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    Cmpl m[5] = { { 2, 2 }, { 1, 1 }, { 5, 5 }, { 7, 7 }, { 3, 3 },
};
    sort(m, m + 5, greater<Cmpl>());
    for each (Cmpl var in m) cout << var.ToString() << endl;
    return 0;
}
```

```
7 *i 7
5 *i 5
3 *i 3
2 *i 2
1 *i 1
Для продолжения нажмите любую клавишу . . .
```

Задание 4.2. Перестановка массива объектов класса Cmpl с помощью алгоритма reverse() в обратном порядке.

Задание 4.3. Сортировка массива объектов класса Cmpl с помощью алгоритма sort() и перегруженной операции > по убыванию вещественной части.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <sstream>
#include <functional>

using namespace std;
//Класс комплексные числа.
class Cmpl
{
    double re;
    double im;
public:
    string ToString()
    {
        string a;
        ostringstream os;
        os << re << " *i " << im;
        return os.str();
    }
    Cmpl(double r = 0, double i = 0) : re(r), im(i)
    {
    }
    bool greater_re(Cmpl b) const
    { return (re) > (b.re); }
};
//Функциональный объект.

class gr
{
public:
    bool operator()(Cmpl &a, Cmpl &b) const
    { return a.greater_re(b); }
};

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    Cmpl n[5] = { { 2, 3 }, { 1, 0 }, { 5, 2 }, { 7, 1 }, { 3, 4 },
};
    sort(n, n + 5, gr()); //Вызов функционального объекта.
```

```

    for each (Cmpl var in n) cout << var.ToString() << endl;
    return 0;
}

```

```

7 *i 1
5 *i 2
3 *i 4
2 *i 3
1 *i 0
Для продолжения нажмите любую клавишу . . .

```

Задание 4.4. Сортировка массива объектов класса Cmpl с помощью алгоритма sort() и перегруженной операции < по возрастанию мнимой части.

Задание 4.5. Поиск объекта в массиве объектов класса Cmpl с помощью алгоритма find().

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <sstream>
#include <functional>

using namespace std;
//Класс комплексные числа.
class Cmpl
{
    double re;
    double im;
public:
    string ToString()
    {
        string a;
        ostringstream os;
        os << re << " *i " << im;
        return os.str();
    }
    bool Cmpl::operator==(Cmpl b) const
    { return (re*re + im*im) == (b.re*b.re + b.im*b.im); }
    Cmpl(double r = 0, double i = 0) : re(r), im(i)
    { }
};

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    Cmpl m[5] = { { 2, 2 }, { 1, 1 }, { 5, 5 }, { 7, 7 }, { 3, 3 },
};
    Cmpl val = Cmpl(5, 5);
    Cmpl *p = find(m, m + 5, val);
    if (p != nullptr)
        cout << "Индекс искомого значения в массиве: " << p - m <<
endl;
    return 0;
}
```

```
Индекс искомого значения в массиве: 2
Для продолжения нажмите любую клавишу . . .
```

Задание 5.1. Сортировка вектора объектов класса Cmpl с помощью алгоритма sort() и перегруженной операции > по убыванию.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <sstream>
#include <functional>
#include <vector>

using namespace std;

//Класс комплексные числа.
class Cmpl
{
    double re;
    double im;
public:
    string ToString()
    {
        string a;
        ostringstream os;
        os << re << " *i " << im;
        return os.str();
    }
    Cmpl(double r = 0, double i = 0) : re(r), im(i)
    {
    }
    bool operator> (Cmpl b) const
    {
        return (re*re + im*im) > (b.re*b.re + b.im*b.im);
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    vector<Cmpl> m = { { 2, 2 }, { 1, 1 }, { 5, 5 }, { 7, 7 }, { 3, 3
}, };
    cout << "Исходный вектор======" << endl;
    for (int i = 0; i < m.size(); i++)
        cout << m[i].ToString() << endl;
    m.push_back(Cmpl(8,8));
    sort(m.begin(), m .end(), greater<Cmpl>()); //Функциональный
//объект.
```

```

cout << "После добавления и сортировки======" << endl;
for each (Cmpl var in m) cout << var.ToString() << endl;
vector<Cmpl>::iterator it;
m.pop_back();
cout << "После удаления======" << endl;
for (it = m.begin(); it < m.end(); it++)
    cout << it->ToString() << endl;

return 0;
}

```

```

Исходный вектор=====
2 *i 2
1 *i 1
5 *i 5
7 *i 7
3 *i 3
После добавления и сортировки=====
8 *i 8
7 *i 7
5 *i 5
3 *i 3
2 *i 2
1 *i 1
После удаления=====
8 *i 8
7 *i 7
5 *i 5
3 *i 3
2 *i 2
Для продолжения нажмите любую клавишу . . .

```



Задание 6.2. Вычислить сумму объектов класса комплексные числа (Cmpl) в деке с помощью алгоритма accumulate() и функционального объекта plus<>().

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <sstream>
#include <numeric>
#include <functional>
#include <deque>

using namespace std;

//Класс комплексные числа.
class Cmpl
{
    double re;
    double im;
public:
    string ToString()
    {
        string a;
        ostringstream os;
        os << re << " *i " << im;
        return os.str();
    }
    Cmpl(double r = 0, double i = 0) : re(r), im(i)
    {
    }
    Cmpl operator+ (Cmpl b) const
    {
        return Cmpl((re + b.re), (im + b.im));
    }
    bool operator> (Cmpl b) const
    {
        return (re*re + im*im) > (b.re*b.re + b.im*b.im);
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    deque<Cmpl> m = { { 2, 2 }, { 1, 1 }, { 5, 5 }, { 7, 7 }, { 3, 3
}, };
    cout << "Исходная дека======" << endl;
    for each (Cmpl var in m) cout << var.ToString() << endl;
    m.push_front(Cmpl(4, 4));
    m.push_back(Cmpl(9, 9));
```

```

sort(m.begin(), m.end(), greater<Cmpl>());
cout << "Дека после добавления и сортировки======" << endl;
for (int i = 0; i < m.size(); i++) cout << m[i].ToString() <<
endl;
m.pop_front();
m.pop_back();
cout << "Дека после удаления======" << endl;
deque<Cmpl>::iterator it;
for (it = m.begin(); it != m.end(); it++) cout << it->ToString()
<< endl;
Cmpl sum = m[0] + m[1];
cout << "m[0] + m[1] = " + sum.ToString() << endl;
sum = accumulate(m.begin(), m.end(), Cmpl(), plus<>());
cout << "Сумма элементов дека = " + sum.ToString() << endl;
return 0;
}

```

```

Исходная дека=====
2 *i 2
1 *i 1
5 *i 5
7 *i 7
3 *i 3
Дека после добавления и сортировки=====
9 *i 9
7 *i 7
5 *i 5
4 *i 4
3 *i 3
2 *i 2
1 *i 1
Дека после удаления=====
7 *i 7
5 *i 5
4 *i 4
3 *i 3
2 *i 2
m[0] + m[1] = 12 *i 12
Сумма элементов дека = 21 *i 21

```

Задание 7.6. Посчитать количество объектов класса комплексные числа (Cmpl) находящихся в списке в заданном диапазоне с помощью алгоритма count\_if() и функционального объекта.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <sstream>
#include <numeric>
#include <functional>
#include <list>

using namespace std;

//Класс комплексные числа.
class Cmpl
{
    double re;
    double im;
public:
    string ToString()
    {
        string a;
        ostringstream os;
        os << re << " *i " << im;
        return os.str();
    }
    Cmpl(double r = 0, double i = 0) : re(r), im(i)
    { }
    bool operator< (Cmpl b) const
    {
        return (re*re + im*im) < (b.re*b.re + b.im*b.im);
    }
};

//Функциональный объект. Принадлежность значения диапазону.
bool Diapazon (Cmpl x) { return x < Cmpl(3, 3); }

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    list<Cmpl> m = { { 2, 2 }, { 1, 1 }, { 5, 5 }, { 0.5, 0.1 }, { 3,
3 }, { 1, 1 } };
    list<Cmpl>::iterator it;
    cout << "Исходный список===== " << endl;
    for (it = m.begin(); it != m.end(); it++) cout <<
(*it).ToString() << endl;
    m.push_front(Cmpl(0, 0));
```

```

m.push_back(Cmpl(9, 9));
m.sort();
cout << "Список после добавления и сортировки=====" << endl;
for each (Cmpl var in m) cout << var.ToString() << endl;
int count = count_if(m.begin(), m.end(), Diapazon());
cout << "Значений в диапазоне < (3 i3) = " << count << endl;
return 0;
}

```

```

Исходный список=====
2 *i 2
1 *i 1
5 *i 5
0.5 *i 0.1
3 *i 3
1 *i 1
Список после добавления и сортировки=====
0 *i 0
0.5 *i 0.1
1 *i 1
1 *i 1
2 *i 2
3 *i 3
5 *i 5
9 *i 9
Значений в диапазоне < (3 i3) = 5

```

Задание 7.7. В списке отыскивается объект класса комплексные числа (Cmpl), удовлетворяющий условию, с помощью алгоритма find\_if и функционального объекта.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <string>
#include <sstream>
#include <numeric>
#include <functional>
#include <list>

using namespace std;

//Класс комплексные числа.
class Cmpl
{
    double re;
    double im;
public:
    string ToString()
    {
        string a;
        ostringstream os;
        os << re << " *i " << im;
        return os.str();
    }
    bool operator< (Cmpl b) const
    {
        return (re*re + im*im) < (b.re*b.re + b.im*b.im);
    }
    Cmpl(double r = 0, double i = 0) : re(r), im(i)
    {
    }
};

//Функциональный объект.Принадлежность значения диапазону.
class Diapazon
{
public:
    bool operator ()(Cmpl x){ return x < Cmpl(1, 1); }
};

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
```

```

list<Cmpl> m = { { 2, 2 }, { 1, 1 }, { 5, 5 }, { 0.5, 0.1 }, { 3,
3 }, { 1, 1 } };
for each (Cmpl var in m) cout << var.ToString() << endl;
//Элемент списка, удовлетворяющего условию.
list<Cmpl>::iterator iter;
iter = find_if(m.begin(), m.end(), Diapazon());
if (iter != m.end())
    cout << "Искомое значение в массиве: " << iter->ToString()
<< endl;
return 0;
}

```

```

2 *i 2
1 *i 1
5 *i 5
0.5 *i 0.1
3 *i 3
1 *i 1
Искомое значение в массиве: 0.5 *i 0.1
Для продолжения нажмите любую клавишу . . .

```

### Указания к выполнению

Тестовые наборы для тестирования задания помещайте в таблицу следующего вида:

Таб. 1. Тестовый набор.

Тестовый набор					
Номер теста	Исходные данные				Ожидаемый результат
1					
2					

### Содержание отчета

1. Задание.
2. Исходные тексты приложения.
3. Исполняемый файл приложения.
4. Тестовые наборы данных для тестирования типа данных.

**Контрольные вопросы**

1. Какие последовательные контейнеры STL вы знаете?
2. Назовите основные свойства контейнера vector?
3. Назовите основные свойства контейнера list?
4. Назовите основные свойства контейнера deque?
5. Что такое адаптеры контейнеров?
6. Какие адаптеры контейнеров STL вы знаете?
7. Назовите основные свойства адаптера контейнера stack?
8. Назовите основные свойства адаптера контейнера queue?
9. Назовите основные свойства адаптера контейнера priority\_queue?