ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ (МГТУ ГА)»

Кафедра вычислительных машин, комплексов, систем и сетей

Л.А. Надейкина

ПРОГРАММИРОВАНИЕ

ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Утверждено Редакционно-издательским советом МГТУ ГА в качестве учебного пособия

Москва
2019

УДК 004.415.2(075.8)
ББК 6Ф7.3
Н-17

Печатается по решению редакционно-издательского совета
Московского государственного технического университета ГА

Рецензенты:
Черкасова Н.И. (МГТУ ГА) – канд. физ.-мат. наук, доцент;
Акиншин Р.Н. (РАН) – д-р техн. наук, доцент

Надейкина Л.А.
Н-17 Программирование. Обобщенное программирование: учебное пособие. / Л.А. Надейкина. — Воронеж ООО «МИР», 2019. — 80 с.

ISBN 978-5-6042751-4-6

Данное учебное пособие издается в соответствии с рабочей программой дисциплины «Программирование» для студентов II курса направления подготовки 09.03.01 бакалавр очного обучения.

В Учебном пособии рассматривается на базе языка С++ одна из основных парадигм современного программирования: обобщенное программирование. При объектно-ориентированном программировании основное внимание уделяется аспекту данных, а при обобщенном — алгоритмам.

Цель обобщенного программирования — создание кода, который не зависит от типов данных. Шаблоны — это средства С++, предназначенные для создания обобщенных программ.

Библиотека STL содержит шаблоны контейнерных классов, по разному сконструированных и предназначенных для хранения данных разных типов. STL предоставляет обобщенное представление алгоритмов, не зависящее как от типов данных, но и от типов контейнеров, которые хранят эти данные. Реализация такого подхода осуществляется с помощью итераторов. Шаблоны обеспечивают обобщенное представление типа данных, хранимых в контейнере. Итераторы является обобщенным представлением процесса перемещения по элементам контейнера.

Рассмотрены шаблоны функций, определяющих общий алгоритм семейства функций, и шаблоны классов, позволяющие определить общие свойства классов. Рассмотрена библиотека STL, которая содержит набор шаблонов, представляющих контейнеры, итераторы, объекты функций и алгоритмы.

Рассмотрено и одобрено на заседании кафедры 18.02.2019 г. и методического совета 24.01.2019 г.

ББК 6Ф7.3
Св. тем. план 2019 г.
поз. 24

НАДЕЙКИНА Людмила Анатольевна
ПРОГРАММИРОВАНИЕ. ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ
Учебное пособие

В авторской редакции

Подписано в печать 12.03.2019 г.
Формат 60x80/16 Печ. л. 3 Усл. печ. л. 2,79
Заказ 418/090444 Тираж 30экз.
Московский государственный технический университет ГА
125993 Москва, Кронштадтский бульвар, д.20
Отпечатано ООО «МИР»
394033, г. Воронеж, Ленинский пр-т 119А, лит. Я, оф. 215

Раздел 1
Основы обобщенного программирования

1.1. Шаблоны функций

Обобщенное программирование — это еще одна парадигма программирования, поддерживаемая языком C++. Оно имеет общую с ООП цель — упростить повторное использование кодов программ и методов абстрагирования общих понятий. Однако в то время как в ООП основное внимание уделяется данным, в обобщенном программировании упор делается на алгоритмы. И у ОП другая область применения. ООП — это инструмент для разработки больших проектов, тогда как обобщенное программирование предоставляет инструменты для выполнения задач общего характера, таких как сортировка данных или объединение списков. Обобщённое программирование — парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание.

Шаблоны функций, так же, как и шаблоны классов, поддерживают в C++ парадигму обобщенного программирования, то есть программирования с использованием типов в качестве параметров. Механизм шаблонов в C++ допускает применение абстрактного типа в качестве параметра при определении функции или класса. После того как шаблон функции или класса определен, он может использоваться для определения конкретных функций или классов.

Шаблон – синтаксическая конструкция языка, позволяющая создавать параметризованные классы и функции с отложенным определением одного или нескольких типов, использованных в тексте определения класса или функции [1].

Рассмотрим вначале шаблоны функций.

Цель определения шаблона семейства функций – автоматизация определения одноименных функций, обрабатывающих разные типы данных**.** Шаблон определяется один, но в нем определение шаблонной функции параметризуется. Параметризовать в шаблонной функции можно тип возвращаемого результата, типы локальных объектов и типы любых параметров.

Формат определения шаблона семейства функций:

template <список параметров шаблона>
определение_шаблонной_функции

Префикс ***template <…>*** указывает, что объявлен шаблон. Для параметризации используется список формальных параметров шаблона, который следует после служебного слова ***template***, заключенный в угловые скобки < >. Среди параметров шаблона могут быть типизирующие, нетипизирующие и параметры-шаблоны. Каждый типизирующий параметр обозначается служебным словом ***class*** или ***typename***, за которым следует идентификатор параметра. Пример определения шаблона семейства

функций, вычисляющих абсолютные значения числовых величин разных типов:

template <class R>
R abs (R x) { return x>0 ? x : -x;}

Шаблон семейства функций состоит из заголовка шаблона:

template <список_параметров_шаблона>

и параметризованного определения функции, в котором тип возвращаемого значения, типы локальных объектов тела функции и типа параметров обозначаются именами типизирующих параметров шаблона, введенных в заголовке. Параметризованное определение функции, входящее в шаблон, будем называть ***шаблонной функцией***.

Пример определения шаблона семейства функций для обмена значений двух передаваемых им аргументов:

template <typename T>
void swap (T&x, T&y)
{ T z = x; x = y; y = z;}

Здесь ***T*** определяет и тип ссылок – параметров функции и тип локальной переменной ***z***.

Шаблон семейства функций служит для определения конкретных функций, называемых ***специализациями шаблонной функции***.

Специализации создаются на основе шаблона по тем вызовам, которые компилятор обнаружит в программе.

Если далее в программе обнаруживаются вызовы функций с данными именами, компилятор идентифицирует типы параметров. Далее он формирует соответствующие определения функций и организует вызовы этих функций в той последовательности, в какой они следуют в программе. Пусть есть следующие вызовы:

abs(-45.8);
long a=4, b=5;
swap(a, b);
double c=3.8, d=6.8;
swap(c, d);

компилятор сначала сформирует определение функции:

double abs (double x) { return x>0 ? x : -x;}

и организует выполнение функции и в точку вызова вернется значение ***45.8.*** Далее компилятор сформирует следующие определения (и последующие вызовы) функций:

void swap (long &x, long &y)
{ long z = x; x = y; y = z; }
void swap (double &x, double &y)
{ double z = x; x = y; y = z; }

Компилятор автоматически генерирует правильный код, соответствующий переданному типу.

Формирование на основе шаблона и имеющегося вызова функции конкретного кода функции (специализации) называется ***инстанцированием (****конкретизацией****, актуализацией****) шаблона.***

Еще один шаблон семейства функций поиска в массиве максимального элемента [2]. Шаблонная функция возвращает ссылку на элемент массива с максимальным значением:

```
#include<iostream>
using namespace std;
//определение шаблона
template < typename type>
type & max ( int n , type A[ ] )
{ int imax =0;
  for(int i=1  ; i< n ; i++)
  if ( A[imax] < A[i] ) imax = i;
return A[imax];
}
 int main(){
int x[4] = {10,20,30,15};
//аргумент целочисленный массив
cout<<"max(4,x)= "<< max(4,x)<<endl;
max(4,x)=0;    // ссылка "возвращает" не значение, сам элемент
for(int i=0;i<4; i++)
cout<<"x["<<i<<"]= "<< x[i]<< '\t';

double y[3] = {10.1, 20.2, 30.3};
//аргумент - массив типа double
cout<<endl<< "max ( 3, y)= "<< max (3, y)<<endl;
max (3, y)=0;
for(int i=0;i<3; i++)
cout<<"y["<<i<<"]= "<< y[i]<< '\t';
return 0;
}
```

Результат выполнения программы:

```
max(4,x)=30
x[0]=10       x[1]=20       x[2]=0       x[3]=25

max(3,y)=30.3

y[0]=10.1       y[1]=20.2       y[2]=0
```

В программе использовались два разных обращения к функции ***max().*** В первом случае параметр – целочисленный массив и возвращаемое значение – ссылка на ***int.*** Во втором случае аргумент – имя массива типа ***double,*** и возвращаемое значение **–** ссылка на ***double.***

Механизм шаблонов позволяет автоматизировать подготовку определений перегруженных функций. Нет необходимости готовить заранее

все варианты перегруженных функций. Компилятор, автоматически анализируя вызовы шаблонных функций, формирует необходимые определения именно для таких типов аргументов, которые использовались в вызовах. Процесс логического (автоматического) определения типов аргументов шаблона функций по типам аргументов в конкретном обращении к шаблонной функции называют ***выведением (deduction) аргументов шаблона функций***. Перечислим основные свойства параметров шаблона**.**

1. Список параметров шаблона не может быть пустым.

2. Кроме типизирующих параметров, у шаблона функций могут быть нетипизирующие параметры.

3. Нетипизирующие параметры явно специфицируются в заголовке шаблона как обычные параметры функций и могут иметь тип как базовый, так и производный. На типы нетипизирующих параметров наложены ограничения. Они не могут быть вещественными, не могут быть классами, не могут иметь тип ***void***. Они могут быть:

- целочисленными;
- указателями на объект или функцию;
- ссылками на объект или функцию;
- указателями на поля данных и методы классов.

4. Аргумент, соответствующий нетипизирующему параметру шаблона, должен быть выражением соответствующего параметру типа.

5. Каждый типизирующий параметр должен начинаться со слова ***class*** или ***typename****:*

template < typename type1, typename type2, typename type3>

6. Аргумент, соответствующий типизирующему параметру шаблона, может быть именем любого типа, как базового, так и производного. Но для этого типа текст шаблонной функции должен иметь смысл.

7. Имена параметров должны быть уникальными в определении шаблона. Имя параметра шаблона скрывает другие использования того же идентификатора в области, глобальной по отношению к шаблону. Если внутри шаблонной функции необходим доступ к внешним объектам с тем же именем, нужно применять операцию указания области видимости.

Для всех описанных выше примеров, типы аргументов шаблона логически выводятся по типам аргументов в вызове функции. Для этой схемы все параметры шаблона должны быть обязательно использованы в спецификациях параметров шаблонной функции.

Этим требованиям не соответствует, например, ***шаблон функции суммирования элементов массива****:*

```
template <typename T, typename RT>
RT sum(T* begin, T* end) {
T *p;
RT s = RT(0);
for (p = begin; p != end; ++p)
s += *p;
return s;}
```

В данном примере остался не использованным в спецификации параметров шаблонной функции параметр шаблона с именем ***RT***. Его применений в качестве типа возвращаемого функцией значения и для определения объекта ***s*** в теле функции недостаточно, чтобы компилятор мог по обращению к функции узнать тип аргумента, соответствующего параметру ***RT***. При попытке вызова шаблонной функции, например, так:

```
int mas[] = {1, 2, 3, 4, 5};
cout << sum(mas, mas + 5) << endl;
```

будет выдано сообщение об ошибке:

```
error C2783: 'RT sum(T *,T *)' : could not deduce template argument for
'RT'
```

В связи с возможностью подобных затруднений Стандарт определяет общий формат обращения к шаблонным функциям с явным указанием фактических параметров шаблона[3]:

```
Имя_шаблонной_функции<список_фактических_параметров_шаблона>
 (список_фактических_параметров_шаблонной_функции);
```

Рассмотрим программу с примерами разных обращений к шаблонным функциям:

```
#include <iostream>
using namespace std;
// Шаблон функции, вычисляющей абсолютное значение
// числовых величин разных типов
template <typename type>
type abs(type x) {
return x > 0 ? x : -x;
}
 // Шаблон функции суммирования элементов массива
template <typename T, typename RT>
RT sum(T* begin,T* end) {
T *p;
RT s = RT(0);
for (p = begin; p != end; ++p)
s += *p;
return s;
}
 // Примеры обращений к шаблонной функции
int main() {
setlocale(LC_CTYPE, "Russian");
int a = 12;
 // Аргумент шаблона задан явно:
cout << abs<int>(a) << endl;
double z = -66.3;
 // Пустой список аргументов:
```

```
cout << abs<>(z) << endl;
 // Без списка аргументов
cout << abs(z) << endl;
 short s[5] = {20000,32000,20000,25000,15000};
// Необходимо задавать явно
// все параметры шаблона
cout << sum<short, int>(s, s+5) << endl;
return 0;
}
```

Результат выполнения программы:

```
12
66.3
66.3
112000
```

Если параметр ***RT*** в определении шаблона функции ***sum*** для возвращаемого значения указать первым в списке параметров шаблона, то при вызове функции можно не задавать явно значение для второго параметра шаблона (***T***) – его компилятор определит автоматически при выведении типа в процессе инстанцирования:

```
template <typename RT, typename T>
RT sum(T * begin, T * end) {
T *p;
RT s = RT(0);
for (p = begin; p != end; ++p)
s += *p;
return s;
}
 ...
short s[5] = {20000, 32000, 20000,25000, 15000};
// Достаточно задать явно только первый
// параметр шаблона:
cout << sum<int>(s, s+5) << endl;
…
```

Чтобы применить шаблон функции *к типу данных, определённому пользователем* (структуре, объекту класса), необходимо ***перегрузить операции для этого типа***, используемые в шаблонной функции.

Прототип шаблона

Как и при работе с обычными функциями, для шаблонов функций существуют определения и описания. В качестве ***описания*** шаблона функций используется ***прототип шаблона***:

```
template <список_формальных_параметров_шаблона>
    Прототип_шаблонной_функции
```

Прототип_шаблонной_функции – это её заголовок (возможно, без имён параметров), вслед за которым помещён разделитель "точка с запятой". В списке параметров прототипа шаблона имена параметров не обязаны совпадать с именами тех же параметров в определении шаблона.

Параметры шаблона, не являющиеся типами. Применение нетипизирующих параметров шаблона.

Параметры, не являющиеся типами, можно использовать в шаблонах функций. Шаблонная функция может иметь любое количество нетипизирующих параметров. Нетипизирующим может быть и возвращаемое функцией значение. Рассмотрим несколько примеров.

1. Рассмотрим программу, в которой шаблон определяет семейство функций, каждая из которых вычисляет количество нулевых элементов одномерного массива параметризованного типа.

```
#include <iostream>
using namespace std;
// Прототип шаблона
template <typename D>
long count0(int, D *);

int main() {
int A[] = { 0, 1, 0, 0, 6, 0, 4, 10};
int n = sizeof(A)/sizeof(A[0]);
cout << "count0(n, A) = " << count0(n, A) << endl;
double X[] = { 10.0, 0.0, 3.3, 0.0, 2.1};
n = sizeof(X)/sizeof(X[0]);
cout << "count0(n, X) = "<< count0(n, X) << endl;
return 0;
}
// Шаблон функций для определения количества
// нулевых элементов в массиве
template <typename T>
long count0(int size, T* mas) {
long k = 0;
for (int i = 0; i < size; ++i)
if (mas[i] == T(0)) ++k;
return k;
}
```

Результат выполнения программы:

```
count0(n,A) = 4
count0(n,X) = 2
```

2. Рассмотрим программу с шаблоном функции, обнуляющей любой двумерный числовой массив:

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

```
template<int n, int m, typename T>
void InitMas(T mas[n][m]) { // Передача массива
for (int i = 0; i < n; ++i)
for (int j = 0; j < m; ++j)
mas[i][j] = T(0);
}
int main() {
const int n = 3, m = 2;
int g[n][m];
 //Вызов ф-ции с явным заданием параметров шаблона
InitMas<n,m,int>(g);     //допустимо InitMas<n,m>(g);
 for (int i = 0; i < n; ++i) {
for (int j = 0; j < m; ++j)
cout << g[i][j] << '\t';
cout << endl;
}
return 0;
}
```

Результат выполнения программы**:**

```
0 0
0 0
0 0
```

3. Если в шаблонной функции использовать ***ссылку на массив***:

```
 template<int n, int m, typename T>
void InitMas(T (& mas)[n][m]) { // Ссылка на массив
for (int i = 0; i < n; ++i)
for (int j = 0; j < m; ++j)
mas[i][j] = T(0);
}
```

то вызвать функцию можно так:

```
…int g[n][m];
// Вызов без явного задания параметров шаблона
InitMas(g);
```

1.2.Явная специализация шаблонной функции

При использовании шаблонов функций возможна перегрузка. Могут быть шаблоны с одинаковыми именами шаблонных функций, но разными спецификациями параметров. Наряду с этим с помощью шаблона может создаваться функция с таким же именем, что и явно определённая функция. В обоих случаях "распознавание" конкретного вызова выполняется по сигнатуре, то есть по типам, порядку и количеству аргументов и по тому пространству имён, к которому относится имя функции.

В некоторых случаях для конкретного варианта типов аргументов может потребоваться особое поведение шаблонной функции. Особое в данном случае означает отличное от того поведения, которое предусмотрено

алгоритмом тела шаблонной функции. При этом программист может написать отдельное определение шаблона функции, которое называют ***явной специализацией шаблонной функции***.

Новый способ определения специализации содержит конструкцию **template<>**. Типы данных, для которого предназначена специализация, указываются внутри угловых скобок после имени функции. Должны быть специализированы все параметры шаблона, ***частичная специализация шаблона функций запрещена***. В этом случае используется перегрузка шаблона другим шаблоном или обычной функцией.

Рассмотрим программу с явной специализацией шаблонной функции.

```
// Использование шаблона функций
#include <iostream>
#include <cstring>
using namespace std;
template <typename T>
T maximum(T v1, T v2, T v3) {
T max = v1;
if (v2 > max) max = v2;
if (v3 > max) max = v3;
return max;
}
// Эта функция замещает обобщенную версию функции
// maximum для нахождения максимальной строки среди строк
template<> // явная специализация шаблонной функции
char* maximum<char*>(char *v1, char* v2, char* v3){
char * max = v1; // выполнение лексикографического сравнения строк
if (strcmp(v2,max) > 0) max = v2;
if (strcmp(v3,max) > 0) max = v3;
return max;
}
int main() {
int i1 = 3, i2 = 1, i3 = 10;
cout << "max = " << maximum(i1,i2,i3) << "\n";// Версия int
double d1 = -3.2, d2 = -5.1, d3 = -2.4;
cout << "max = " << maximum(d1, d2,d3) << "\n";// Версия double
 char c1 = 'C', c2 = 'Z', c3 = 'a';
cout << "max = " << maximum(c1,c2,c3) << "\n";  // Версия char
 char *s1 = "Victorov", *s2 = "Antonov", *s3 = "Victor";
// Версия char*
cout << "max = " << maximum(s1,s2,s3) << "\n";
// maximum <char*>(s1,s2,s3);
return 0;
}
```

Результат выполнения программы:

```
max = 10
max = -2.4
```

max = a
max = Victorov

Здесь ***maximum <char*>(s1,s2,s3);*** - вызов специализированной функции, Префикс ***template<>*** не содержит параметров, это означает, что в этой специализации шаблона параметры отсутствуют. Однако в шаблоне может быть несколько параметров, и те или иные специализации могут содержать их разное сочетание.

Определим шаблон функций для вычисления суммы значений элементов "сегмента" массива. Параметры шаблона функций: ***T*** – тип элементов массива и тип возвращаемого шаблонной функцией результата, ***N*** – индекс начала сегмента. Параметры шаблонной функции: ***T mas [ ]*** – массив, ***int len*** - количество элементов в сегменте.

```
#include <iostream>
using namespace std;
template <int N, typename T>
T sum(T mas [], int len=1) {
T s = T(0);
for(int i=N; i<N+len;i++)
s+=mas[i];
return s;
}
// явная специализация шаблонной функции
template < >
char sum <0,char>(char mas[], int len) {
int sr=0;
for(int i=0; i< len;i++)
sr+=mas[i];
return (char)(sr/len);
}
int main () {
int A[] = {11,1,1,0,0,6,0,4,10};
int n = sizeof(A)/ sizeof(A[0]);
cout<<"sum<3,int>(A,n-3) = "<< sum<3,int>(A,n-3)<<endl;
cout<<"sum<3>(A,n-3) = "<< sum<3>(A,n-3)<<endl;
double B[] = {10.0, 0.0, 3.3, 0.0, 2,1};
cout<<"sum<2,double>(B) = "<< sum<2,double>(B)<<endl;
cout<<" sum<0>(B) = "<< sum<0>(B)<<endl;
//работа со специализированной версией
char h []={'1', '2', '3', '4', '5', '6', '7', '8'};
cout<<"sum<0,char>(h,8) = "<< sum<0,char>(h,8)<<endl;
return 0;
}
```

Результат выполнения программы:

```
sum<3,int>(A,n-3) = 20
sum<3>(A,n-3) = 20
```

sum<2,double>(B) = 3.3
sum<0>(B) = 10
sum<0,char>(h,8) = 4

Каждый из элементов массива ***h*** представляет символ цифры от ***'1'*** до ***'8'***. Усреднив их коды, функция возвращает символ ***'4'***.

1.3. Шаблоны классов

Шаблоны классов**,** которые иногда называют родовыми или параметризированными типами**,** позволяют создавать семейство родственных классов.

Определение шаблонного (обобщенного, родового) класса имеет вид:

template <список_параметров_шаблона>
спецификация_шаблонного_класса

Здесь угловые скобки являются неотъемлемым элементом определения. Список параметров шаблона должен быть заключен именно в угловые скобки.

Как и параметры шаблона функции, параметры шаблона класса и соответствующие им аргументы, могут быть трех видов: типизирующие (задающие тип в спецификации класса), нетипизирующие и параметры-шаблоны. Каждый типизирующий параметр шаблона вводится с помощью служебных слов ***class*** или ***typename***.

Шаблон семейства классов определяет способ построения отдельных классов, определяет ***"архитектуру"*** конкретных классов**.** В определении класса, входящего в шаблон, особую роль играет ***имя класса***. Оно является не именем отдельного класса, а ***параметризованным именем семейства классов***.

Шаблон семейства классов служит для автоматического формирования определений конкретных классов - специализаций шаблона. Специализации создаются по тем обращениям к шаблонному классу при создании конкретных классов, которые транслятор встречает в тексте программы. Инстанцирование (актуализация) шаблона классов – это генерация определения класса.

Рассмотрим, например, точку на плоскости. Для ее представления ранее мы разработали класс ***point***, в котором положение точки задавалось двумя координатами ***x*** и ***y*** — полями типа ***double***.

Представим теперь, что в другом приложении требуется задавать точки для целочисленной системы координат, то есть использовать поля типа ***int***.

Можно вообразить себе системы, в которых координаты точки имеют тип ***short*** или ***unsigned char***.

Чтобы не определять каждый раз конкретные классы для схожих задач ***Бьерна Страуструпа добавил в C++ поддержку шаблонов классов [1].***

Определение шаблонного (обобщенного, родового) класса имеет вид:

template <параметры_шаблона> class имя_класса { /*... */ };

Например, определение шаблонного класса ***point*** будет выглядеть следующим образом:

```
template<class T>
class  point {
public:
point(T _x = T(0), T _y = T(1)) : x(_x), y(_y) {}
void show() {
cout<< " (" << x << "," <<y<< ")" << endl;}
point<T>&  operator+ (point<T> & p)
{ return point<T> (x+p.x, y+p.y);}
private:
T x, y;
};
```

Здесь префикс ***template <class T>*** указывает, что объявлен шаблон классов, в котором ***T*** — некоторый абстрактный тип (параметр шаблона), служебное слово ***class*** в данном контексте задает вовсе не класс, а означает лишь то, что ***T*** — это параметр шаблона. После объявления, имя ***T*** используется внутри шаблона точно так же, как имена других типов. Повторим, что язык позволяет вместо ключевого слова ***class*** перед параметром шаблона использовать другое ключевое слово — ***typename***.

В шаблонный класс входят: два закрытых поля данных ***x, y*** (их тип определяет параметр шаблона), конструктор и два метода, иллюстрирующих разное применение параметра шаблона классов и имени шаблонного класса. Метод ***show()*** не имеет параметров и ничего не возвращает, параметр шаблона в нем не используется. Метод ***operator+()*** выполняет перегрузку операции сложения, тип параметра и возвращаемого значения – параметризированное имя шаблонного класса ***point<T>.***

Как и в случае обычных классов, компилятор автоматически включает в класс ***point<T> деструктор, конструктор копирования, и операцию – функцию присваивания*** с такими прототипами***:***

```
~ point<T>:
point (const point<T>&)
point<T>& operator = (const point<T>&)
```

В данном примере все методы определены внутри класса и реализуются как встроенные. Их синтаксис мало отличается от определения в обычном классе. Отличие – параметризация имени шаблонного класса при обозначении типа. Отметим еще раз, имя шаблонного класса (в данном примере) – это ***point***, а классу, вводимому приведенным шаблоном, соответствует тип ***point<T>.***

Так имя шаблонного класса ***point*** без списка параметров входит в определение имени ***конструктора и деструктора***. Но если в методах нужно использовать имя шаблонного класса для обозначения типа, следует использовать конструкцию ***point<T>.***

Использование шаблона классов

При включении шаблона класса в программу никакие классы на самом деле не генерируются до тех пор, пока не будет создан экземпляр шаблонного класса, в котором вместо абстрактного типа ***T*** указывается некоторый конкретный тип. Такая подстановка называется ***инстанцированием*** (***актуализацией)*** шаблона классов, которая приводит к созданию некоторой спецификации шаблона.

Когда шаблон классов введен, появляется возможность определять конкретные объекты конкретных классов, каждый из которых параметрически порожден из шаблона. Формат определения объектов:

имя_шаблонного_класса (аргументы шаблона)
имя_объекта (аргументы_ конструктора);

Как и для обычного класса, экземпляр создается либо объявлением объекта, например,

point<int> anyPoint(13, -5);

либо объявлением указателя на актуализированный шаблонный тип с присваиванием ему адреса, возвращаемого операцией ***new***, например,

point<double>* pOtherPoint = new point<double>(9.99. 3.33);

Встретив подобные объявления, компилятор генерирует код соответствующего класса.

Пример применения шаблонного типа ***point<T>***:

```
int main() {
point<int>one, two(10, 20);   //два объекта одного класса
cout<<"one:\t";
one. show();                  //one. point<int>::show();
cout<<"two:\t";
two. show();                  //two. point<int>::show();
one=two;                      //one. point<int>::operator=(two);
one = one+ two;               //one =one. point<int>::operator+(two);
 cout<<"one:\t";
one. show();
//создается динамический объект другого класса
point<double>*ptr=new point<double>(9.9, 3.3);
ptr-> show();                 //ptr-> point<double>::show()
return 0;
}
```

Результат выполнения программы:

```
one:      (0, 1)
two:      (10,20)
one:      (20, 40)
(9.9,3.3)
```

Внешнее определение методов и дружественных функций шаблонных классов.

Методы шаблона автоматически становятся шаблонными функциями. Спецификация шаблонного класса может не включать полных определений его методов. В этом случае определение метода выносится за пределы шаблона класса, и синтаксис его определения отличается. Особым образом надо определять и дружественные функции шаблонных классов. Формат внешнего определения заголовка метода:

template <описание_параметров_шаблона>
возвр_тип имя_класса <параметры_шаблона>::
имя_функции (список_параметров функции)

Покажем это на примере метода ***show ( )*** шаблона ***point:***

```
template<class T>
class point { …
void show();
…};

// Внешнее определение метода show()
template <class T>    //Префикс шаблона
void                  //тип возвращаемого значения
point<T>::            //Пространство имен
show()                // имя метода
{cout << " (" << x << ", " <<y << ")" <<endl;}
```

Обратите внимание на появление того же префикса ***template <class T>,*** который предварял объявление шаблона класса, а также на более сложную запись операции квалификации области видимости для имени ***show():*** если раньше мы писали ***point::,*** то теперь пишем ***point<T>::,*** добавляя к имени класса список параметров шаблона, заключенный в угловые скобки (в данном случае один параметр ***T***)

Рассмотрим это на примере. Введем ***шаблон классов последовательность значений***, тип которых определяется параметром шаблона. Шаблонный класс включает поля данных:

- длину последовательности (реальную) ***int size***;
- массив фиксированного размера ***data[maxsize];***

и методы:

- конструктор с параметрами, позволяющий формировать объект – последовательность по массиву-параметру;
- конструктор умолчания для создания пустой последовательности;
- операцию - функцию перегрузки операции += конкатенации последовательностей.

Шаблонный класс имеет дружественные функции: операция функция перегрузки операции вывода << и функцию, возвращающую первый элемент последовательности.

```
#include <iostream>
using namespace std;
const int max=200;
```

```
template <typename T>
class line  {     //последовательность
T data [max];
int size;
public:
line():size(0){ }   //конструктор умолчания
line(T ar [], int k);   //прототип конструктора с параметрами
//Конкатенация последовательностей
line<T>&operator+=(line<T>);
//прототип шаблона дружественной функции перегрузки вывода
template<class M>
friend ostream & operator << ( ostream& ,  line <M>& );
//прототип шаблона дружественной функции, возвращающую первый
элемент
template<class F>
friend F& getfirst ( line <F> );
};
//Конкатенация последовательностей
template <typename T>
line<T>& line<T>:: operator+=(line<T>ar){
if(size+ar.size>max) { cout<<"error";exit(1);}
for(int i=0;i<ar.size; i++)
data[i+size]=ar.data[i];
size+=ar.size;
return *this;
}
//конструктор с параметрами
template <typename T>
line<T>::line (T ar[], int k){
if(k<0||k>max){ cout<<"Error"; exit(1);}
for(int i=0;i<k; i++)
data[i]=ar[i];
size=k;
}
// шаблон дружественной функции перегрузки вывода
template<class M>
 ostream & operator << ( ostream&out ,  line <M>&ar )
{out<<"size="<<ar.size<<"line element: "<<endl;
for(int i=0;i<ar.size; i++)
out<<ar.data[i]<<((i+1)%11 ? ";" : ";\n");
out<<endl;
return out;
}
template <class F>
F& getfirst ( line <F> f)
{return f.data[0];}
```

```
int main()
{int ar[]={1,2,3,4,5,6,7,8,9,0},
len =sizeof(ar)/ sizeof(ar[0]);
line<int>l1(ar,len);
cout<<l1;
line <int>l2(ar,len)
l1+=l2;
cout<<l1;
line<char>str1("asgftyy",7), str2("1234567890",10);
str1+=str2;
cout<<str1;
cout << getfirs(str1);
return 0;
}
```

Результат выполнения программы:

```
size=10
line element:
1;2;3;4;5;6;7;8;9;0;
size=20
line element:
1;2;3;4;5;6;7;8;9;0;1;
2;3;4;5;6;7;8;9;0;
size=17
line element:
a;s;g;f;t;y;y;1;2;3;4;
5;6;7;8;9;0;
a
```

1.4. Параметры шаблонов

Параметрами шаблонов могут быть абстрактные типы или переменные встроенных типов, например, таких как ***int.***

Первый вид параметров мы уже рассмотрели – это типизирующие параметры. При инстанцировании на их место подставляются аргументы либо встроенных типов, либо типов, определенных программистом.

Второй вид – это нетипизирующие параметры, используются, когда предусматривается возможность ***"настройки"*** шаблона ***некоторой константой***. Например, можно создать шаблон массивов, содержащих ***n*** элементов типа ***T***:

```
template <class T, int n>
class Array { / * . . . * / };
```

Инстанцировав шаблон типом ***point***

```
Array<point, 20> ap;
```

создадим массив ***ap*** из ***20*** элементов типа ***point***.

Приведем менее тривиальный пример использования параметров второго вида[2]. Определим две функции:

```
void  f1() { cout << "I am f1 () . " << endl; }
void  f2() { cout << "I am f2 ( ) . " << endl; }
template <void (*pf)()>
struct  A
{ void show() { pf();} };
int main() {
A<&f1> aa;
aa. show();   // вывод: I am f1().
A<&f2> ab;
ab. show();// вывод: I am f2().
return 0;
}
```

Здесь параметр шаблона имеет тип указателя на функцию. При инстанцировании класса в качестве аргумента подставляется адрес (константа) соответствующей функции.

В отличие от шаблонов функций, для параметров шаблона класса можно указывать умалчиваемые значения.

В качестве примера приведем шаблон классов для представления массивов фиксированного размера, определяемого ***нетипизирующим*** параметром. Другой ***типизирующий*** параметр будет задавать тип элементов массива. Для обоих параметров шаблона зададим умалчиваемые значения. В шаблонном классе определим конструктор умолчания, и операцию-функцию индексирования.

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

template <class T=char, int size=64>
class arr  {
T data [size];
int length;
public:
arr():length(size){ }   //конструктор умолчания
T& operator [ ] (int i) //операция-функция индексирования
 { if( i<0 | | i>= size){ cout<<"Index error"; exit(1);}
return data[i];
}
};  //конец шаблона классов

int main () {
arr<double,5> rf;
arr<int>ri;            // элементы массива типа int,  и умолчание для size
arr< > rc;             // умолчание для обоих параметров
```

```
for (int i=0; i<5; i++)
{  rf[i]=i;   ri[i]=i*i;  rc [i] = 'A'+ i; }
for (int i=0; i<5; i++)
cout<<rf[i]<< "  " << ri[i]<< " "  <<rc[i]<< '\t';
return 0;
}
```

Результат выполнения программы:

0 0 A 1 1 B 2 4 C 3 9 D 4 16 E

Параметры шаблона в примере имеют умалчиваемые значения. Для параметра ***T*** –это тип ***char***. А для параметра ***int size*** - умалчиваемое значение задается константой ***64***. Аргументом нетипизирующего параметра может быть только константное выражение. В том случае, когда использовались оба умалчиваемые значения, при создании объектов после имени шаблонного класса необходимо писать пустые угловые скобки <>.

1.5. Специализации шаблонов классов

Шаблон классов определяет метасредства языка и описывает, какие действия должен выполнить компилятор после препроцессорной обработки текста программы. Рассмотрим схематично этот процесс для программы с шаблоном классов ***arr< >***

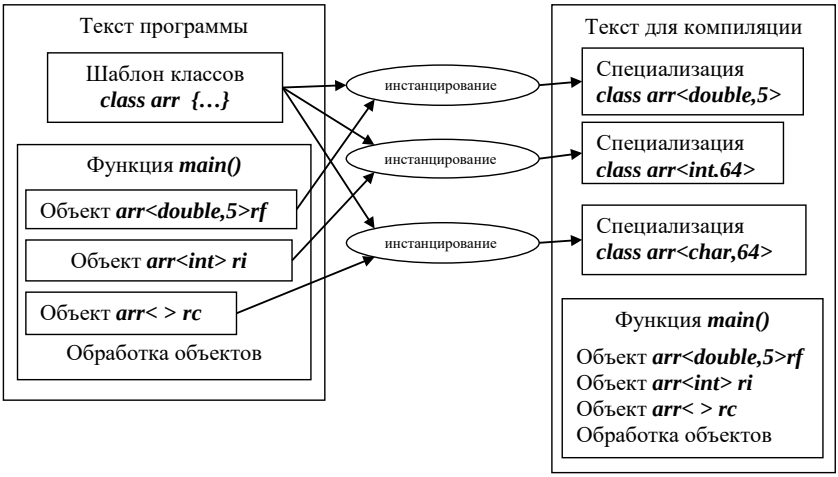

Рисунок 1. Схема автоматической генерации специализаций шаблона классов

В исходном тексте один шаблон, а количество определений классов, то есть количество сгенерированных специализаций, равно числу инстанцирований шаблона для разных типов данных.

Зная определение шаблона и набор аргументов при обращении к шаблону, компилятор формирует спецификацию конкретного класса и создает определения его методов. Сформированные классы называются сгенерированными специализациями. При такой автоматизированной генерации каждый вариант класса получает одинаковую по набору полей данных спецификацию и одинаковые алгоритмы методов.

В ряде случаев это неудобно, так как для конкретного набора аргументов шаблона могут существовать более эффективные алгоритмы методов и более подходящие наборы полей данных. Можно предусмотреть ***пользовательскую специализацию шаблона класса***. Ее называют просто ***специализацией шаблона***, а о сгенерированных специализациях вообще не говорят. Пользовательская специализация шаблона может быть ***явной (полной) и частичной.***

Явная (полная) специализация шаблона классов

Представляет собой конкретную реализацию класса для фиксированных комбинаций аргументов. Явная специализация располагается после определения шаблона класса и имеет следующую общую форму:

template <>
спецификация_параметризованного_класса

Спецификация параметризованного класса имеет вид:

class имя_шаблонного_класса <список_аргументов_шаблона>

{ /* …*/ };

В теле класса специализации каждое появление параметра шаблона должно быть заменено конкретными аргументами шаблона, указанными в заголовке. Имена конструкторов и деструкторов не параметризуются.

Например, невозможно дать обобщенный алгоритм, проверяющий отношение < (меньше) для двух аргументов типа ***T***, который одновременно подходил бы и для числовых типов, и для традиционных Си-строк, завершающихся нулевым байтом. Следует воспользоваться специализацией для типа ***char****:

```
// общий шаблон
template <class T> class Sample {
bool Less(T) const;
/*...*/
};
// специализация для char*
template <> class Sample<char*> {
bool Less(char*) const;
/*...*/
};
```

В качестве примера введем специализацию рассмотренного выше шаблона ***line<>,*** когда последовательность представляет собой строку символов в стиле Си, оканчивающуюся байтовым нулем.

Может быть введена следующая специализация шаблона классов ***line<>:***

```
template < >
class line<char>  {
char data [max];
int size;
public:
line(char*str=0){   // конструктор с параметрами
size=(int)strlen(str);
if(size>max){cerr<<"error"; exit(1);}
if(str!=0)
strcpy(data,str);
}
//Конкатенация последовательностей, прототип
line<char>&operator+=(line<char>);
//прототип  дружественной функции перегрузки вывода
friend ostream & operator << ( ostream& ,  line <char>& );
};
//Конкатенация последовательностей
line<char>& line<char>:: operator+=(line<char>ar){
if(size+ar.size>max) { cout<<"error";exit(1);}
strcat (data, ar.data);
size+=ar.size;
return *this;
}
// дружественная функция перегрузки вывода (это не шаблон!)
 ostream & operator << ( ostream&out ,  line <char>&ar )
{out<<"My string: size="<<ar.size<<"\nline: "<<endl;
out<<ar.data<<endl<<endl;
return out;
}
int main(){
line<char> l1("Цифры:");
cout<<l1;
line<char> l2("0123456789");
l1+=l2;
cout<<l1;
int ar[]={1,2,3,4,5,6,7,8,9,0},
len =sizeof(ar)/ sizeof(ar[0]);
line<int>l3(ar,len);
cout<<l3;
line <int>l2(ar,len)
l1+=l2;
return 0;
}
```

Результат выполнения программы:

My string: size= 6
line: Цифры:
My string: size= 16
line: Цифры:0123456789
size=10
Line elements:
1;2;3;4;5;6;7;8;9;0;

Частичная пользовательская специализация

Если шаблонный класс имеет несколько параметров, то возможна *частичная специализация.*

Пример частичной специализации:

// общий шаблон

template <class T1, class T2> class Pair { / * . . . * / };

/ / специализация, где для T2 установлен тип int

template <class T1> class Pair <T1, int> { /*..*/ };

В любом случае общий шаблон должен быть объявлен прежде любой специализации!

Иногда есть смысл специализировать не весь класс, а какой-либо из его методов. Например,

template<classT>

// обобщенный метод

bool Sample<T>::Less(T ob) const { / * . . . * / };

/ / специализированный метод

bool Sample<char*>: :Less(char* ob) const { / * . . . * / };

Достоинства и недостатки шаблонов

Шаблоны представляют собой мощное и эффективное средство обращения с различными типами данных, которое можно назвать *параметрическим полиморфизмом*, обеспечивают безопасное использование типов, в отличие от *макросов препроцессора*, и являются вкупе с шаблонами функций средством реализации идей обобщенного программирования и метапрограммирования.

Однако следует иметь в виду, что эти средства предназначены для грамотного использования и требуют знания многих тонкостей. Программа, использующая шаблоны, содержит код для каждого порожденного типа, что может увеличить размер *исполняемого файла*. Кроме того, с одними типами данных шаблоны могут работать не так эффективно, как с другими. В этом случае имеет смысл использовать специализацию.

Стандартная библиотека C++ предоставляет большой набор шаблонов для различных способов организации хранения и обработки данных.

1.6 Разработка пользовательского шаблона классов списков.

Прежде чем заняться разработкой пользовательских шаблонов классов, перечислим правила описания шаблонов:

- Локальные классы не могут содержать шаблоны в качестве своих элементов.
- Шаблоны методов не могут быть виртуальными.
- Шаблоны классов могут содержать статические элементы, дружественные функции и классы.
- Шаблоны могут быть производными, как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов.

Шаблон классов позволяет задать *шаблонный класс*, параметризованный типом данных. Передача классу различных типов данных в качестве параметра создает семейство родственных классов. Наиболее широкое применение шаблоны находят при создании *контейнерных классов*. Контейнерным называется класс, который предназначен для хранения каким-либо образом организованных данных и работы с ними. Преимущество использования шаблонов состоит в том, что как только *алгоритм* работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода.

В качестве примера шаблона рассмотрим полное описание параметризованного класса двусвязного списка ***List.*** Список состоит из узлов, связанных между собой с помощью указателей. Каждый узел хранит типизированное данное. Шаблонный класс включает вспомогательный локальный класс для представления одного узла списка.

Класс ***List*** предназначен для хранения данных. Чтобы хранить в нем данные любого типа, требуется описать этот класс как шаблон и передать тип в качестве параметра.

```
template <class Data> // Data- параметр шаблона
class List{
class Node{  //класс определяет узел списка
public:
Data d;     //данные
Node *next, *prev; //указатели на последующий и предыдущий узелы
Node (Data dat = 0){ // конструктор
d = dat; next = 0; prev = 0;
}
};
```

/* Поскольку класс ***Node*** описан внутри класса, представляющего список, поля для простоты доступа из внешнего класса сделаны доступными (***public***). Это позволяет обойтись без функций доступа и изменения полей.*/

```
Node *pbeg, *pend; // указатели на начало и конец списка
public:
```

```
List(){ pbeg = 0; pend = 0;} //конструктор
~List( ); // Деструктор
void add(Data d);   // Добавление узла в конец списка
Node * find(Data i); // Поиск узла по ключу
// Вставка узла d после узла с ключом key:
Node * insert(Data key, Data d);
bool remove(Data key); // Удаление узла
void print();                  // Печать списка в прямом направлении
void print_back();         // Печать списка в обратном направлении
};
```

```
//-------------------------------------
```

/* Деструктор списка освобождает память при уничтожении списка*/

```
template<class Data>
List <Data>::~List(){
if (pbeg !=0){
Node *pv = pbeg;
while (pv){
pv = pv->next; delete pbeg;
pbeg = pv;}
//---------------------------------------
```

/*Метод печати списка в прямом направлении поэлементно просматривают список, переходя по соответствующим ссылкам */

```
template<class Data>
void List <Data>::print(){
Node *pv = pbeg;
cout<< endl<< "list: ";
while (pv){
cout<< pv->d << ' ';
pv = pv->next;}
cout<< endl;
}
```

```
//-------------------------------------------
```

/*Метод печати списка в обратном направлении*/

```
template<class Data>
void List <Data>::print_back(){
Node *pv = pend;
cout<< endl << " list back: ";
while (pv){
cout<< pv->d << ' ';
pv = pv->prev;}
cout<< endl;
}
//-------------------------------------
```

/*метод ***add*** выделяет память под новый объект типа ***Node*** и присоединяет его к списку, обновляя указатели на его начало и конец*/

```
template<class Data>
void List <Data>::add(Data d){
Node *pv = new Node(d);
if (pbeg == 0)pbeg = pend = pv;
else{
pv->prev = pend;
pend->next = pv;
pend = pv;}
}
//-----------------------------------------------------
```

/* метод ***find*** выполняет поиск узла с заданным ключом и возвращает указатель на него в случае успешного поиска и 0 в случае отсутствия такого узла в списке*/

```
template<class Data>
Node * List <Data>::find (Data d){
Node *pv = pbeg;
while (pv){
if(pv->d == d)break;
pv = pv->next;
}
return pv;
}
//-------------------------------------------------------------
```

/*метод ***insert*** вставляет в список узел после узла с ключом ***key*** и возвращает указатель на вставленный узел. Если такого узла в списке нет, вставка не выполняется и возвращается значение 0*/

```
template<class Data>
Node * List <Data>::insert(Data key, Data d){
if(Node *pkey = find(key)){ // Поиск узла с ключом key
// Выделение памяти под новый узел и его инициализация:
Node *pv = new Node(d);
pv->next = pkey->next; // Установление связи нового узла с последующим
pv->prev = pкey; // Установление связи нового узла с предыдущим
pkey->next =pv; // Установление связи предыдущего узла с новым
// Установление связи последующего узла с новым
if ( pкey != pend)(pv->next)->prev = pv;
else pend = pv; // Обновление указателя на конец списка,
// если узел вставляется в конец:
return pv;}
return 0;
}
```

```
// ------------------------------------------
/*Метод remove удаляет узел с заданным ключом из списка и возвращает
значение true в случае успешного удаления и false, если узел с таким ключом
в списке не найден */
template<class Data>
bool List <Data>::remove(Data key){
if(Node *pkey = find(key)){
if (pkey == pbeg){            //Удаление из начала списка
pbeg = pbeg->next; pbeg->prev = 0;}
else if (pkey == pend){ // Удаление из конца списка
pend = pend->prev; pend->next = 0;}
else {                        //Удаление из середины списка
(pkey->prev)->next = pkey->next;
(pkey->next)->prev = pkey->prev;}
delete pkey;
return true;
}
return false;
}
```

Если требуется использовать шаблон ***List*** для хранения данных не встроенного, а определенного пользователем типа, в описание этого типа необходимо добавить перегрузку операции вывода в потоки, сравнения на равенство, а если для его полей используется динамическое выделение памяти, то и операцию присваивания.

При определении синтаксиса шаблона было сказано, что в него, кроме типов и шаблонов, могут передаваться *переменные.* Они могут быть целого или перечисляемого типа, а также указателями или ссылками на объект или функцию. В теле шаблона они могут применяться в любом месте, где допустимо использовать константное выражение. В качестве примера создадим шаблон класса, содержащего блок памяти определенной длины и типа.

```
template <class Type, int kol>
class Block{
public:
Block(){p = new Type [kol];}//конструктор
~Block(){delete [] p;} //деструктор
operator Type *();//операция  преобразования объекта к данному p
protected:
Type * p;//указатель на первый элемент блоки
};
template <class Type, int kol>
Block <Type, kol>:: operator Type *(){
return p;
};
```

После создания и отладки шаблоны классов удобно помещать в заголовочные файлы.

Примеры создания объектов по шаблонам:

```
List <int> List_int;  // список целых чисел
List <double> List_double; // список вещественных чисел
List <monster> List_monster; // список объектов класса monster
Block <char, 128> buf;       // блок символов
Block <monstr, 100> stado;   // блок объектов класса monster
```

При использовании параметров шаблона по умолчанию список аргументов может оказаться пустым, при этом угловые скобки опускать нельзя:

```
template<class T = char> class String;
String<>* p;
```

Для каждого инстанцированного класса компилятор создает имя, отличающееся и от имени шаблона, и от имен других инстанцированных классов. Тем самым каждый инстанцированный класс определяет отдельный тип.

В классе-шаблоне разрешено объявлять статические методы и поля, однако следует учесть, что каждый инстанцированный класс обладает собственной копией статических элементов.

После создания объектов с ними можно работать так же, как с объектами обычных классов, например:

```
for (int i = 1; i<10; i++) List_double.add(i*0.08);
List_double.print();

for (int i = 1; i<10; i++) List_monster.add(i);
List_monster.print();
strcpy (buf, "Очень важное сообщение");
cout << buf << endl;
```

Для упрощения использования шаблонов классов можно применить переименование типов с помощью ***typedef:***

```
typedef List <double> Ld;
Ld List_double;
```

1.7. Шаблонный класс векторов

Разработаем шаблонный класс ***Vect*** для представления динамических одномерных массивов (векторов). Класс должен обеспечивать хранение данных любого типа ***T***, для которого предусмотрены конструктор по умолчанию, конструктор копирования и операция присваивания.

Класс должен содержать:

- конструктор по умолчанию, создающий вектор нулевого размера;
- конструктор, создающий вектор заданного размера;
- операцию индексирования, возвращающую ссылку на соответствующий элемент вектора;

- метод, добавляющий элемент в произвольную позицию вектора;
- метод, добавляющий элемент в конец вектора;
- метод, удаляющий элемент из конца вектора.

При необходимости можно добавить в класс другие методы. А также предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций. В ***main()*** продемонстрируем использование этого класса.

Одним из принципиальных вопросов при разработке контейнерного класса является вопрос реализации хранения элементов контейнера, или, другими словами, выбор подходящей структуры данных (массив, список, бинарное дерево и т. п.).

В данном случае мы остановим наш выбор на динамическом массиве, размещаемом в памяти посредством операции ***new,*** поскольку это наиболее простое решение. После размещения адрес первого элемента вектора будет запоминаться в поле ***T* first***, а адрес элемента, следующего за последним, — в поле ***T* last.*** Используем в классе метод ***size(),*** возвращающий количество элементов в векторе.

При решении этой задачи мы постараемся сделать класс ***Vect*** как можно более похожим (с точки зрения его интерфейса и, частично, реализации) на контейнерный класс ***std::vector*** из библиотеки ***STL.*** Поэтому в состав интерфейса будут включены методы:

void insert(T* _P, const T& _x) — вставка элемента в позицию _P;
void push_back(const T& _x) — вставка элемента в конец вектора;
void pop_back() — удаление элемента из конца вектора;
T* begin() — получение указателя на первый элемент;
T* end() — получение указателя на элемент, следующий за последним;
size_t size() — получение размера вектора (тип ***std::size_t*** является синонимом типа ***unsigned int***).

Эти методы имитируют интерфейс класса ***std::vector***. Уточним, забегая вперед, что вместо типа ***iterator***, имеющегося в ***std::vector***, здесь используется указатель ***T****, но концептуально это одно и то же. Целью такого подражания является психологическая подготовка к более легкому восприятию контейнерных классов ***STL.***

К двум конструкторам, требующимся по заданию, добавим конструктор копирования, чтобы обеспечить возможность передачи объектов класса в качестве аргументов для любой внешней функции.

По некоторым параметрам наш класс ***Vect***, однако, будет превосходить класс ***std::vector.*** Для целей отладки и обучения мы хотим визуализировать работу таких методов класса, как конструктор копирования и деструктор, путем вывода на терминал соответствующих сообщений.

Чтобы эти сообщения были более информативны, мы снабдим каждый объект класса так называемым отладочным именем (поле markName), которое позволит распознавать источник сообщения.

Трудно переоценить, какую пользу приносят эти сообщения начинающим программистам, помогая прочувствовать скрытые механизмы

функционирования класса. Для более опытных читателей эти средства могут оказаться полезными при поиске неочевидных ошибок в программе.

Также предусмотрим генерацию и обработку исключений для возможных ошибочных ситуаций. Наиболее очевидными являются: *ошибка индексирования,* когда клиент пытается получить доступ к несуществующему элементу вектора, и *ошибка удаления несуществующего элемента* из конца вектора — когда вектор пуст. В принципе, возможна еще одна нештатная ситуация, связанная с нехваткой памяти, когда операция new возвращает нулевой указатель. Но мы подробно ее рассмотрели, как решать эту проблему, в разделе «Исключения в конструкторах» [4], и сейчас, ради краткости изложения, будем полагать, что обладаем неограниченной памятью.

Для обработки исключительных ситуаций создадим иерархию классов во главе с базовым классом ***VectError.*** Производный класс ***VectRangeErr*** будет предназначен для обработки ошибок индексирования, а производный класс ***VectPopErr*** - для обработки ошибки удаления несуществующего элемента.

Ниже приводится код предлагаемого решения задачи, после которого даются некоторые пояснения.

```
//VectError.h
#ifndef _VECT_ERROR_
#define _VECT_ERROR_
#include <iostream>
#define DEBUG
class VectError {
public:
VectError() {}
virtual void ErrMsg() const {
std::cerr << "Error with Vect object.\n";
}
void Continue() const {
#ifdef DEBUG
std::cerr << "Debug: program is being continued.\n";
#else
 throw;
#endif
}};
class VectRangeErr : public VectError {
public:
VectRangeErr ( int _min, int _max, int_actual) :
min(_min), max(_max), actual(_actual) { }
void ErrMsg() const {
std::cerr << "Error of Index: ";
std::cerr << "possible range: " << min << " - " <<
max << ", ";
```

```
std::cerr << "actual Index: " << actual << std::endl;
Continue();
}
private:
int min. max;
int actual;
};
class VectPopErr : public VectError {
public:
void ErrMsg() const {
std::cerr « "Error of pop\n";
Continue();
}};
//Vect.h
//=================Vector==========================
#ifndef _VECT__
#define _VECT_
#include <iostream>
#include <string>
#include "VectError.h"
// Template class Vect
template<class T> class Vect {
public:
explicit Vect()//явный, неконвертируемый конструктор по умолчанию
: first(0), last(0) { }
// явный конструктор, создающий вектор заданного размера;
explicit Vect (size_t  _n,  const T& _v = T())
{ //std::size_t –синонимом  unsigned //int
Allocate(_n);
for (size_t i = 0 ; i < _n; ++i)
* ( fisrst + i) = _v;
}
Vect(const Vect&); // конструктор копирования
Vect& operator =(const Vect &); // операция присваивания
~Vect() { //деструктор
#ifdef DEBUG
cout << "Destructor of " << markName << endl:
#endif
Destroy();
first = 0; last = 0;
}
// --------------методы ––––––––––––
//установить отладочное имя
void mark(std::string& name) { markName = name: }
std::string mark() const { return markName; }// получить отладочное имя
```

```
size_t siz() const;    // получить размера вектора
T* begin() const { return first; }  // получить указатель на 1-й элемент
//получить указатель на следующий за последним элементом
T* end() const { return last; }
T& operator [] (size_t i);            // операция индексирования
void insert (T* _P, const T& _ x );      // вставка элемента в позицию _P
void push_back(const T& _ x ); // вставка элемента в конец вектора
void pop_back();// удаление элемента из конца вектора
void show() const; // вывод в cout содержимого вектора

protected:
void Allocate(size_t _n) {
fisrt = new T [_n * sizeof(T)];
last = fisrt +_n;
}
void Destroy () {
for (T* p = first; p != last; ++p) p->~T();
delete [] first;  }

T* first; // указатель на 1-й элемент
T* last: // указатель на элемент, следующий за последним
std::string markName;
};

// конструктор копирования
template<class T>
Vect<T>::Vect(const Vect<T>& other) {
size_t n = other.size();
Allocate(n);
for (size_t i = 0 ; i < n; ++i)
*(first + i) = *(other.first + i);
markName = string("Copy of ") + other.markName;
#ifdef DEBUG
cout << "Copy constructor: "<< markName<< endl;
#endif
// Операция присваивания
template<class T>
Vect<T>& Vect<T>::operator =(const Vect& other)
if (this == &other) return *this;
Destroy();
size_t n = other.size();
Allocate(n);
for (size_t i = 0 ; i < n; ++i)
*(first + i) = *(other.first + i);
return *this;
}
// Получение размера вектора
```

```
template<class T>
size_t Vect<T>::size() const {
if (first > last)
throw VectError();
return (0 == first ? 0 : last - first);
}
// Операция доступа по индексу
template<class T>
T& Vect<T>::operator[](size_t i) {
if(i < 0 || i > (size() - 1) )
throw VectRangeErr (0, last - first- 1, i);
return (*(first + i ) ) ;
}
// Вставка элемента со значением _x в позицию _P
template<class T>
void Vect<T>::insert(T* _P, const T& _x) {
size_t n = size() + 1;                    // новый размер
T* new_first = new T [n * sizeof(T)];
T* new_last = new_first + n;
size_t offset = _P - first;
for (size_t i = 0; i < offset; ++i)
*(new_first + i) = *(first +i );
*(new_first + offset) = _x;
for (i = offset; i < n; ++i)
*(new_first + i + 1) = *(first + i ) ;
Destroy();
first = new_first;
last = new _last;
}
// Вставка элемента в конец вектора
template<class T>
void Vect<T>::push_back(const T& _x){
if (!size()) {
Allocated(1);
*first = _x;
}
else insert(end(), _x); }
// Удаление элемента из конца вектора
template<class T>
void Vect<T>::pop_back() {
if(last == first)
throw VectPopErr();
T* p = end() - 1;
p->~T();
last- -;
}
```

```
// Вывод в cout содержимого вектора
template<class T>
void Vect<T>::show() const {
cout << "\n===== Contents of " << markName <<"=====" << endl;

size_t n = size();
for (size_t i = 0 ; i < n; ++i)
cout << * ( first+ i) << " ";
cout << endl;
}
#endif /* _VECT_ */
//Main.cpp
#include "Vect.h"
#include <iostream>
#include <string>
using namespace std;
template<class T> void SomeFunction(Vect<T> v) {
std::cout << "Reversive output for " << v.mark() <<
size_t n = v.size();
for (int i = n - 1; i >= 0: - -i)
std::cout << v[i] << " ";
std::cout << endl;
}
int main() {
try {
string initStr[5] = {"first", "second", "third" "fourth", " fifth" } ;
Vect<int> v1(10); v1.mark(string("v1"));
size_t n = v1.size();
for (int i = 0; i < n; ++i)
v1[i] = i+1;
v1.show();
SomeFunction(v1);
try
{
Vect<string> v2(5);
v2.mark(string("v2"));
size_t n = v2.size();
for (int i = 0: i < n; ++i)
v2[i] = initStr [ i ];
v2.show();
v2.insert(v2.begin()+3 , "After_third");
v2.show();
cout << v2[6] << endl;
v2.push_back("Add_1");
v2.push_back("Add_2");
```

```
v2.push_back("Add_3");
v2.show();
v2.pop_back();
v2.pop_back();
v2.show();
}
catch (VectError& vre) { vre.ErrMsg(); }
try {
Vect<int> v3;
v3.mark(string("v3"));
v3.push_back(41);
v3.push_back(42);
v3.push_back(43);
v3.show();
Vect<int> v4;
v4.mark(string("v4"));
v4 = v3;
v4.show();
v3.pop_back(); v3.pop_back();
v3.pop_back(); v3.pop_back();
v3.show();
}
catch (VectError& vre) { vre.ErrMsg(); }
}
catch (...) { cerr << "Epilogue: error of Main().\n"; }
return 0;
}
```

Модуль ***VectError.h*** содержит иерархию классов исключений. В базовом классе ***VectError*** определены два метода. Виртуальный метод ***ErrMsg(),*** обеспечивающий вывод по умолчанию сообщения об ошибке (в производных классах этот метод замещается конкретными методами)

Метод ***Continue(),*** управляющий стратегией продолжения работы программы после обнаружения ошибки. Стратегия зависит от конфигурации программы. Конфигурация программы управляется наличием или отсутствием единственной директивы в начале файла: ***#define DEBUG.***

• Если лексема ***DEBUG*** определена, то программа компилируется в отладочной конфигурации, если не определена — то в выпускной конфигурации. Для метода ***Continue()*** это означает, что:

- в отладочной конфигурации выводится сообщение «Debug: program is being continued», после чего работа программы продолжается;

- в выпускной конфигурации повторно возбуждается (оператор ***throw***) то исключение, которое было первопричиной цепочки событий, завершившихся вызовом *данного метода.*

• В производных классах ***VectRangeErr и VectPopErr*** переопределяется метод ***ErrMsg()*** с учетом вывода информации о конкретной ошибке. После

вывода вызывается метод ***Continue ()*** базового класса. Обратите внимание, что класс ***VectRangeErr*** содержит поля ***min, max и actual***, в которых запоминается и затем выводится с помощью метода ***ErrMsg()*** информация, существенная для диагноза ошибки.

Модуль ***Vect.h*** содержит определение шаблонного класса ***Vect.*** Отметим наиболее интересные моменты. Для управления выделением и освобождением ресурсов класс снабжен методами ***Allocate()*** и ***Destroy().*** Напомним, что операция ***new T [n],*** где ***n*** — количество элементов, которое мы хотим поместить в массив, не только выделяет память, но инициализирует элементы посредством выполнения ***T()*** — конструктора по умолчанию для типа T. Поэтому в методе ***Destroy ()*** мы сначала в цикле вызываем деструкторы ***~T()*** для всех элементов массива, а уже потом освобождаем память операцией ***delete***.

Обратите внимание на то, что в заголовке оператора ***for*** условие продолжения циклических вычислений записано в виде ***p != last***, а не ***p < last***. Здесь мы следуем традиции ***STL***, порожденной тем обстоятельством, что для произвольной структуры данных (например, для списка), соседние элементы не обязаны занимать подряд идущие ячейки памяти, а могут быть разбросаны в памяти как угодно. Поэтому адрес любого элемента в контейнере может оказаться больше адреса ***last.*** Условие же ***p != last*** позволяет корректно организовать цикл для любой произвольной структуры данных. Деструктор и конструктор копирования содержат вывод некоторых сообщений, дополненных печатью содержимого ***markName***. Весь этот вывод осуществляется только в отладочной конфигурации.

Операция присваивания реализована стандартным способом.

В операции доступа по индексу ***operator[] ()*** , если индекс оказывается вне диапазона, вызывается исключение типа ***VectRangeErr***. Вызов производится через анонимный экземпляр класса с передачей конструктору трех аргументов: минимальной границы, максимальной границы и текущего значения для индекса.

В методе получения размера вектора ***size ()*** предусмотрена проверка на правильность между ***first*** и ***last.*** Оператор, обнаружив несоответствие, генерирует исключение типа ***VectError.***

Наиболее сложным в реализации оказался метод ***insert ()*** — вставка элемента со значением ***_x*** в произвольную позицию ***_P***. Алгоритм вставки работает следующим образом. Создается новый вектор размером на единицу больше существующего вектора; адреса размещения нового вектора сохраняются в локальных переменных ***new_first, new_last***. Затем в новый вектор копируется первая часть существующего вектора, начиная с первого элемента и заканчивая элементом, предшествующим элементу с адресом ***_P.*** Поскольку доступ к элементам осуществляется через смещение относительно указателя ***first,*** перед копированием требуется вычислить ***offset*** — смещение, соответствующее адресу ***_P***. Затем элементу с адресом ***new_first + offset*** присваивается значение ***_x.*** После этого оставшаяся часть существующего вектора копируется в новый вектор, начиная с позиции ***new_first + offset + 1***.

Вот и все. Осталось освободить ресурсы, занятые существующим вектором ***(Destroy()),*** и назначить полям ***first и last*** новые значения.

Метод ***push_back()*** — вставка элемента со значением ***_x*** в конец вектора — получился очень простым. Если вектор пуст, то выделяются ресурсы для хранения одного элемента, и этому элементу присваивается значение ***_x.*** В противном случае вызывается ***insert()*** для вставки элемента в позицию, определенную с помощью ***end().***

В методе ***pop_ back()*** — удаление элемента из конца вектора — проверяется, непуст ли вектор. В пустом векторе ***first = last = 0***. Если это так, то удалять нечего, и, значит, клиент ошибся, затребовав такую операцию, а следовательно, надо ему об этом сообщить — генерируется исключение типа ***VectPopErr.*** В противном случае определяется указатель на последний элемент вектора, вызывается для него деструктор ***~ T(),*** и уменьшается значение ***last*** на единицу.

Модуль ***Main.cpp*** содержит определение глобальной шаблонной функции ***SomeFunction()*** и определение функции ***main().*** Функция ***SomeFunction()*** обеспечивает реверсивный вывод содержимого вектора в поток ***cout.*** Стратегическое же ее назначение — проверить передачу объектов конкретного класса ***Vect*** в качестве аргументов функции.

Функция ***main()*** предназначена для тестирования класса ***Vect.*** Внешний блок ***try*** содержит в себе два внутренних блока ***try***. И это не случайно!

Первый внутренний блок ***try*** предназначен для тестирования операций с объектом ***v2***, среди которых есть операция, вызывающая ошибку индексирования.

Второй блок ***try*** работает с объектом ***v3***, причем одна из операций вызывает ошибку удаления несуществующего элемента.

Обратите внимание на то, что обработчики ***catch*** в обоих случаях имеют своим параметром объект базового класса ***VectError***, но благодаря полиморфизму они будут перехватывать и исключения производных классов ***VectRangeErr*** и ***VectPopErr***, так что в результате будет вызываться конкретный метод ***ErrMsg()*** для данного типа ошибки!

После вывода сообщения об ошибке метод ***ErrMsg()*** вызывает метод ***Continue()***, а работа последнего зависит от конфигурации программы. В отладочной конфигурации ***Continue()*** сообщает о продолжении работы программы и возвращает управление клиенту. В выпускной конфигурации ***Continue()*** возбуждает исключение повторно. Так вот, для того чтобы его поймать, и служит внешний блок ***try*** со своим обработчиком ***catch(…)*** в основной программе.

Раздел 2.

Стандартная библиотека шаблонов STL (Standard Template Library)

2.1. Механизмы, используемые при построении STL.

Стандарт языка C++ включает обширную библиотеку функций и классов, создающих эффективную среду для разработке программ. Особое

место в стандартной библиотеке занимает ее подмножество - ***Standard Template Library – стандартная библиотека шаблонов (STL).***

Назначение STL обеспечивать программиста типовыми структурами данных и наиболее эффективными алгоритмами, настроенными на обработку информации, представленной этими данными[5]. В отличие от традиционных библиотек в STL типовые структуры данных и алгоритмы для их обработки представлены в виде шаблонов. Тем самым программист имеет возможность настраивать структуры данных и алгоритмы STL на обработку самых разных типов данных. Возможность настройки существенно расширяет универсальность и гибкость STL.

Шаблоны были введены в язык программирования C++ как средство выражения параметризованных типов. Примером параметризованного типа является, например, список, для которого вы не хотите реализовывать отдельные версии для каждого типа хранимых в нем элементов. Вместо этого вы хотите предусмотреть единственную реализацию списка (шаблон), которая использует "заполнитель" типа элементов (параметр шаблона), при помощи которого компилятор может генерировать различные классы списка (экземпляры шаблонов).

Сегодня C++ шаблоны используются совершенно неожиданным для их создателей образом. Шаблонное программирование включает в себя такие методики как, вычисления во время компиляции, шаблонное метапрограммирование, порождающее программирование и этот список можно продолжать.

Метапрограммирование — вид программирования, связанный с созданием программ, которые порождают другие программы как результат своей работы. Метапрограммирование реализовано в той или иной мере в очень разных языках; если не рассматривать экзотические и близкие к ним языки, то самым известным примером метапрограммирования является C++ с его системой шаблонов.

Метапрограммирование шаблонов (template metaprogramming – ***TMP***) – это процесс написания основанных на шаблонах программ на ***C++,*** исполняемых во время компиляции (генерация специализаций). Шаблонная метапрограмма – это программа, написанная на C++, которая исполняется *внутри компилятора C++*. Когда ***TMP***-программа завершает исполнение, ее результат – фрагменты кода на C++, конкретизированные из шаблонов, – компилируется как обычно.

C++ не предназначался для метапрограммирования шаблонов, но с тех пор, как технология ***TMP*** была открыта в начале 90-х годов, она оказалась настолько полезной, что, вероятно, и в сам язык, и в стандартную библиотеку будут включены расширения, облегчающие работу с ***TMP***. Да, ***TMP*** было именно открыто, а не придумано. Средства, лежащие в основе ***TMP***, появились в C++ вместе с шаблонами [6].

Технология ***TMP*** дает два преимущества. Во-первых, она позволяет делать такие вещи, которые иными способами сделать было бы трудно либо вообще невозможно. Во-вторых, поскольку шаблонные метапрограммы исполняются во время компиляции C++, они могут переместить часть

работы со стадии исполнения на стадию компиляции. В частности, некоторые ошибки, которые обычно всплывают во время исполнения, можно было бы обнаружить при компиляции. Другое преимущество – это то, что программы C++, написанные с использованием ***TMP***, можно сделать эффективными почти во всех смыслах: компактность исполняемого кода, быстродействия, потребления памяти. Но коль скоро часть работы переносится на стадию компиляции, то, очевидно, компиляция займет больше времени. Для компиляции программ, в которых применяется технология ***TMP***, может потребоваться больше времени, чем для компиляции аналогичных программ, написанных без применения ***TMP***. В данном пособии не будет проведен детальный разбор, методологии ***TMP***. Остановимся на первоначальном предназначении шаблонов.

В целом, ***STL*** состоит из двух основных частей: ***классов контейнеров***, пригодных для хранения элементами разных типов и ***набора обобщенных алгоритмов*** для выполнения различных операций над контейнерами, над их элементами.

Алгоритмы ***STL*** называют обобщенными потому, что они не зависят ни от вида контейнера, в котором находятся элементы, ни от типа элементов контейнера. Это достигается за счет еще одного механизма – ***итератора***. Итератор – это связывающее звено алгоритма с контейнером. Назначение итератора – обеспечить алгоритму универсальный доступ к элементам контейнера, не зависящий от его вида. Концепция итератора более трудная для понимания.

Таким образом, ядро библиотеки STL образуют три элемента: контейнеры, алгоритмы и итераторы.

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

Использование контейнеров позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки.

Обобщенные алгоритмы реализуют большое количество процедур, применимых к контейнерам — например, поиск, сортировку, слияние и т. п. Однако они не являются методами контейнерных классов. Наоборот, алгоритмы представлены в ***STL*** в форме глобальных шаблонных функций.

Благодаря этому достигается их универсальность: эти функции можно применять не только к объектам различных контейнерных классов, но также, например, и к массивам.

Независимость функций от типов контейнеров достигается за счет косвенной связи функции с контейнером: в функцию передается не сам контейнер, а пара адресов ***first, last***, задающая диапазон обрабатываемых элементов. Реализация указанного механизма взаимодействия базируется на использовании так называемых ***итераторов***.

Контейнеры и итераторы.

Итераторы — это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать, как обычные указатели, для последовательного продвижения по контейнеру, а также разыменовывать для получения или изменения значения элемента. Для них должны быть определены операции сравнения, операция присваивания.

Итераторы предназначены для предоставления единого метода последовательного доступа (перебора) элементов ***контейнера,*** не зависящего от вида контейнера и типа элементов в нем. Итератор может пробегать как все элементы, так и некоторое их подмножество

В результате использования ***итераторов*** алгоритм может «не замечать», с каким контейнером он работает. Тем самым появляется возможность унификации алгоритмов.

Итератор обеспечивает:

- *единый механизм перебора элементов контейнера, не зависящий ни от его вида, ни от его реализации, ни от типа элементов;*
- *множественный доступ к контейнеру, позволяющий работать с контейнером сразу нескольким клиентам, при этом каждый из клиентов будет пользоваться своим итератором.*

При последовательном переборе элементов контейнера ***итератор*** не обязательно перемещается только по смежным (соседним) элементам.

Алгоритм последовательного перебора элементов для каждого вида итератора и для различных контейнеров может быть определен по-своему.

Но для алгоритма клиента, который пользуется итератором, обращаясь к элементам контейнера, совокупность перебираемых элементов представляется в виде последовательности, имеющей ***начало и конец***.

Итератор – это абстракция, обобщающая понятие указателя. Подобно тому, как значением указателя является адрес элемента массива, так значением итератора служит позиция элемента в контейнере.

Внешне одинаковые операции, получаемые при операциях с итераторами разных контейнеров, могут быть по-разному получены внутри контейнеров.

Так перемещение итератора к следующему элементу контейнера (операция ++) по-разному реализуется в векторе и в списке.

Следовательно, для каждого класса контейнеров должен быть определен свой класс (или несколько классов) итераторов (локальный или

дружественный), предоставляющий средства для создания итератора для обхода элементов контейнера.

Контейнер содержит элементы последовательности, начало, и конец последовательности, которые представляются значениями итераторов.

Эти значения доступны с помощью методов любого контейнерного класса:

begin()-возвращает значение итератора (позицию в контейнере), установленного на начало последовательности;

end()-возвращает значение итератора, установленного за последним элементом последовательности. Итераторам будет посвящен отдельный параграф.

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

Последовательные контейнеры обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности. К ним относятся векторы (***vector***), двусторонние очереди ***(deque***) и списки (***list***), а также так называемые адаптеры, то есть варианты, контейнеров — стеки (***stack***), очереди (***queue***) и очереди с приоритетами (***priority_queue***).

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

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров: словари (***map***), словари с дубликатами (***multlmap***), множества (***set***), множества с дубликатами (***multiset***) и битовые множества (***bitset***). Программист может создавать собственные контейнерные классы на основе имеющихся контейнеров в стандартной библиотеке.

Контейнерные классы обеспечивают стандартизованный интерфейс при их использовании. Смысл одноименных операций для различных контейнеров одинаков, основные операции применимы ко всем типам контейнеров. Стандарт определяет только интерфейс контейнеров, поэтому разные реализации могут сильно отличаться по эффективности.

2.2. Итераторы

Чтобы понять, механизм итераторов, рассмотрим, реализацию шаблонную функцию поиска значения в обобщенном массиве, хранящем объекты типа ***T*** (***T*** – параметр шаблона функции):

```
template <class T>
T* find(T* ar, int n, const T& value) {
```

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

При продвижении по массиву адрес следующего элемента вычисляется с использованием арифметики указателей, то есть он отличается от адреса предыдущего элемента на некоторое фиксированное число байтов, требуемое для хранения одного элемента.

Попытаемся теперь расширить сферу применения нашей функции — хорошо бы, чтобы она решала задачу поиска заданного значения среди объектов, хранящихся в виде линейного списка.

Однако, к сожалению, ничего не выйдет: адрес следующего элемента в списке нельзя вычислить, пользуясь арифметикой указателей.

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

Авторы STL решили эту проблему, введя понятие ***итератора*** как более абстрактной сущности, чем указатель, но обладающей похожим поведением.

Для всех контейнерных классов STL определен тип ***iterator***, однако, реализация его в разных классах разная.

Например, в классе ***vector***, где объекты размещаются один за другим, как в массиве, тип итератора определяется посредством:

```
typedef  T* iterator;
```

А вот в классе ***list*** тип итератора реализован как встроенный класс ***iterator***, поддерживающий основные операции с итераторами.

К основным операциям, выполняемым с любыми итераторами, относятся:

• *Разыменование итератора: если* ***p*** *— итератор, то* ****p*** *— значение объекта, на который он "ссылается" (позицию которого он сохраняет).*

• *Присваивание одного итератора другому.*

• *Сравнение итераторов на равенство и неравенство* ***(== и !=).***

•*Перемещение его по всем элементам контейнера с помощью префиксного* ***(++p)*** *или постфиксного* ***(p++)*** *инкремента.*

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

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

Как было сказано выше, для всех контейнерных классов определены унифицированные методы ***begin()*** *и* ***end()****, возвращающие адреса* ***first и last*** *соответственно.*

Организация циклов просмотра элементов контейнеров тоже имеет некоторую специфику.

Так, если ***i*** - имя итератора, то вместо привычной формы

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

используется следующая:

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

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

Начиная со стандарта C++11, ключевое слово ***auto*** может использоваться вместо типа переменной при инициализации для выполнения вывода типа. Рассмотрим пример использования ***auto*** для выведения типа итератора:

//определение контейнера с помощью списка инициализации элементов

```
std::list<double> numbers {1,2,3,4,5,6};
for(auto i= numbers.begin(); i!= numbers.end();++i)
cout<<*i;
```

Это весьма удобно использовать, так как полное имя типа итератора:

```
std::list<double>:: iterator;
```

Стандарт C++11 ввел новый вид цикла ***for*** , основанный на итераторах и удобный для перебора контейнеров. Называется он ***range-based for statement.*** В примере ***auto*** выводит тип элемента массива:

```
int array [4]={1,2,4,5};
for(auto& item : array)
item*=item;
for(int  item : array)
cout<<item;
```

Итераторы подчиняются принципу чистой абстракции, то есть любой объект, который ведет себя как итератор, является итератором.

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

Входные итераторы (InputIterator) используются алгоритмами ***STL*** для чтения значений из контейнера, аналогично тому, как программа может вводить данные из потока ***cin***.

Выходные итераторы (OutputIterator) используются алгоритмами для записи значений в контейнер, аналогично тому, как программа может выводить данные в поток ***cout***.

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

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

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

В то время как значения прямых, двунаправленных и итераторов произвольного доступа могут быть сохранены, значения входных и выходных итераторов сохраняться не могут (аналогично тому, как не может быть гарантирован ввод тех же самых значений при вторичном обращении к потоку cin). Следствием является то, что любые алгоритмы, базирующиеся на входных или выходных итераторах, должны быть однопроходными.

Важную роль в ***STL*** играют итераторы потоков, которые делятся на итераторы потоков ввода и вывода. Практически итератор потока вывода используется для отображения данных на экране. Суть применения потоковых итераторов в том, что они превращают любой поток в итератор, используемый точно так же, как и прочие итераторы: перемещаясь по цепочке данных, считывает значения объектов и присваивает им другие значения. (Действуют по системе шаблона – подставляют ("определяют") переменные в определенную конструкцию).

Итератор потока ввода — это удобный программный интерфейс, обеспечивающий доступ к любому потоку, из которого требуется считать данные. Конструктор итератора имеет единственный параметр — поток ввода. А поскольку итератор потока ввода представляет собой шаблон, то ему передается тип вводимых данных. Вообще-то должны передаваться четыре параметра, но последние три имеют значения по умолчанию, и вряд ли стоит их изменять прежде, чем досконально изучить ***STL***. Каждый раз, когда нужно ввести очередной элемент информации, используйте оператор ++ точно так же, как с основными итераторами. Считанные данные можно узнать, если применить разыменовывание ***(*)***.

Итератор потока вывода весьма схож с итератором потока ввода, но у его конструктора имеется дополнительный параметр, которым указывают строку-разделитель, добавляемую в поток после каждого выведенного элемента. Ниже приведен пример программы, читающей из стандартного потока ***cin*** числа, вводимые пользователем и дублирующие их на экране, завершая сообщение строкой ***“ — new data“.*** Работа программы заканчивается, как только пользователь введет число ***33:***

```
#include <iostream>
 using namespace std;
 int main() {
```

```
 istream_iterator<int> is (cin);
ostream_iterator<int> os  (cout, " — new data");
int input;
while  ((input = *is) != 33) {
*os++ = input;
 is++ ; }
 return 0;}
```

Потоковые итераторы имеют одно существенное ограничение — в них нельзя возвратиться к предыдущему элементу. Единственный способ сделать это - заново создать итератор потока

Вернемся к функции ***find()***, которую мы безуспешно пытались обобщить для любых типов контейнеров.

В ***STL*** аналогичный алгоритм имеет следующий прототип:

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

2.3. Контейнеры

Общие свойства контейнеров

В табл. 1 приведены имена типов, определенные с помощью ***typedef*** в большинстве контейнерных классов.

Таблица 1

Унифицированные типы, определенные в STL

тип	определение
value_type	Тип элемента контейнера
size_type (unsigned int)	Тип индексов, счетчиков элементов и т. д.
iterator	Тип итератор
reverse_iterator	Тип обратного итератора
const_iterator	Константный итератор (значения элементов изменять запрещено)
const_reverse_iterator	Константный обратный итератор
reference	Ссылка на элемент
const_reference	Константная ссылка на элемент (значение элемента изменять запрещено)
key_type	Тип ключа (для ассоциативных контейнеров)
key_compare	Тип критерия сравнения (для ассоциативных контейнеров)

Итератор используется для просмотра контейнера в прямом или обратном направлении. От итератора требуется уметь ссылаться на элемент контейнера и реализовывать операцию перехода к его следующему элементу.

Константные итераторы используются тогда, когда значения соответствующих элементов контейнера не изменяются.

В табл. 2 представлены некоторые общие для всех контейнеров операции и методы.

Таблица 2.

Операции и методы, общие для всех контейнеров

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

При помощи итераторов можно просматривать контейнеры, не заботясь о фактических типах данных, используемых для доступа к элементам. Для этого в каждом контейнере определено несколько методов, перечисленных выше в таблицах.

Другие поля и методы контейнеров рассмотрим по мере необходимости.

STL определяется в заголовочных файлах: ***algorithm, deque, functional, iterator, list, map, memory, numeric, queue, set, stack, utility, vector*** и др.

Последовательные контейнеры

К основным последовательным контейнерам относятся ***вектор (vector), список (list)*** и ***двусторонняя очередь (deque).*** Чтобы использовать в программе эти контейнеры, нужно включить в программу соответствующие заголовочные файлы:

#includcde <vector>
#include<list>
#include<deque>

Пример простейшего создания контейнеров, тип объектов, сохраняемых в контейнере, задается с помощью аргумента шаблона:

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

Векторы (vector), двусторонние очереди (deque) и списки (list) поддерживают разные наборы операций, среди которых есть совпадающие операции. Они могут быть реализованы с разной эффективностью.

Например, *вектор* — это структура, эффективно реализующая произвольный доступ к элементам, добавление в конец и удаление из конца. *Двусторонняя очередь* эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов. *Список* эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам. Остановимся подробнее на каждом из них.

1. Векторы (vector)

Класс векторов STL — это класс шаблона контейнеров последовательностей для хранения элементов заданного типа в линейном порядке и быстрого произвольного доступа к любому элементу. Он является наиболее подходящим типом контейнера для последовательности, когда на первом месте стоит производительность произвольного доступа.

Контейнер ***вектор*** является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации ***[]*** или метода ***at().***

Метод ***at()*** аналогичен операции индексации, но в отличие от нее, проверяет выход индекса за границу вектора. Если такое нарушение обнаруживается, то метод генерирует исключение ***out_of_range***.

Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

Синтаксис:

template <class T, class Allocator = Allocator<T>>
class vector {
/*Параметры:
T - тип данных элементов, сохраняемых в векторе.

Allocator-тип, представляющий сохраненный объект распределителя, содержащий сведения о распределении и отмене распределения памяти для вектора. */

//методы класса, члены и

//некоторые конструкторы:

```
explicit  vector();
explicit vector( size_type Count);
vector(  size_type Count,   const T& Val= T(0));
vector(  const vector& Right);
template <class InputIterator>
vector( InputIterator First,   InputIterator Last);};
```

Параметры конструкторов:

Count - количество элементов в создаваемом векторе.

Val - значение элементов в создаваемом векторе.

Right - Вектор, для которого создаваемый вектор станет копией.

First- положение первого элемента в диапазоне копируемых элементов.

Last - положение первого элемента за пределами диапазона копируемых элементов.

Примеры программы создания и вывода векторов:

```
#include <vector>
#include <iostream>
int main()
{   using namespace std;
 vector <int>::iterator  v1_Iter,  v2_Iter,  v3_Iter,  v4_Iter,  v5_Iter,  v6_Iter;
//создание векторов, используя различные конструкторы:
 // Создание пустого вектора
  vector <int> v;
 //Создание вектора из трех элементов, со значениями по умолчанию (0).
 vector <int> v1(3);
// Создание вектора из 5 элементов, со значениями  2
 vector <int> v2(5, 2);
// Создание вектора v4копии  вектора     v2
 vector <int> v4(v2);
// Создание временного вектора для   демонстрации копирования диапазона
vector <int> v5(5);
for (auto i : v5) {
   v5[i] = i;  }
```

```
// Создание  v6 копированием диапазона элементов  вектора  v5[ first,  last)
   vector <int> v6(v5.begin() + 1, v5.begin() + 3);

 //вывод содержания векторов:
cout << "v1 =";
 for (auto& v : v1){
 cout << " " << v; }
 cout << endl;

 … //пропустили вывод элементов  векторов v2- v5

cout << "v6 =";
for (auto& v : v6){
cout << " " << v;}
 cout << endl;

// Создание вектора v7  перемещением элементов  вектора v2
 vector <int> v7(move(v2));
vector <int>::iterator v7_Iter;

cout << "v7 =";
for (auto& v : v7){
cout << " " << v; }
cout << endl;

vector<int> v8{ { 1, 2, 3, 4 } };  //создание вектора по списку инициализации
for (auto& v : v8){
cout << " " << v ; }
cout << endl;
}
```

В шаблоне **vector** определены операция ***присваивания*** и функция ***копирования (assign*)**:

```
vector<T>& operator=(const vector<T>& x);// операция присваивания

void assign(size_type n, const T& value);//копирование n значений value в
                                          существующий вектор

template <class InputIter>

void assign( InputIter  first , InputIter last);//копирование диапазона в
                                                 //существующий вектор
```

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

Пример:

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

Метод capacity() определяет **размер оперативной памяти,** занимаемой вектором:

size_type capacity() const;

Память под вектор выделяется динамически, но не под один элемент в каждый момент времени (это было бы расточительным расходованием ресурсов), а сразу под группу элементов, например, 256 или 1024.

Существует также функция выделения памяти ***reserve(),*** которая позволяет задать, сколько памяти требуется для хранения вектора:

void reserve(size_type n);

Пример применения функции:

vector <int> v;

v.reserve(1000); / / Выделение памяти под 1000 элементов

Для изменения размеров вектора служит функция ***resize:***

void resize(size__type sz, T c = T());

Эта функция увеличивает или уменьшает размер вектора в зависимости от того, больше задаваемое значение ***sz*** чем значение **size(),** или меньше.

Второй параметр задает значение, которое присваивается всем новым элементам вектора. Они помещаются в конец вектора. Если новый размер меньше, чем значение ***size()***, из конца вектора удаляется ***size() - sz*** элементов.

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

void push__back(const T& value); -добавляет элемент в конец вектора,

void pop_back();- удаляет элемент из конца вектора

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

Функция ***insert*** служит для вставки элемента в вектор. Первая форма функции вставляет элемент ***value*** в позицию, заданную первым параметром (итератором), и возвращает итератор, ссылающийся на вставленный элемент. Вторая форма функции вставляет в вектор n одинаковых элементов. Третья форма функции позволяет вставить несколько элементов, которые могут быть заданы любым диапазоном элементов подходящего типа. Функция ***erase***() служит для удаления одного элемента вектора (первая форма функции) или диапазона, заданного с помощью итераторов (вторая форма). Примеры:

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

vector <int> v2;
for (int i = 1; i<6; i++) v2.push_back(i);
// Содержимое v: 1 2 3 4 5
v2.erase(v2.begin()); / / Содержимое v2: 2 3 4 5
v2.erase(v2.begin() ,v2.begin() + 2); / / Содержимое v2: 4 5

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

Функция ***swap()*** служит для обмена элементов двух векторов одного типа, но необязательно одного размера:
vector <int> v1, v2;
vl.swap(v2); / / Эквивалентно v2.swap(v1)

Для векторов определены **операции сравнения** ==, !=,<,<=.

2. Двусторонние очереди (deque)

Двусторонняя очередь — это последовательный контейнер, который, наряду с вектором, поддерживает произвольный доступ к элементам и обеспечивает вставку и удаление из обоих концов очереди. Те же операции с элементами внутри очереди занимают время, пропорциональное количеству перемещаемых элементов. Распределение памяти выполняется автоматически. Двусторонняя очередь организованна сложнее, чем вектор, с использованием блочной структуры, но доступ также как в векторе осуществляется посредством указателей. Доступ к элементам очереди осуществляется за постоянное время, хотя оно и несколько больше, чем для вектора.

Для создания двусторонней очереди можно воспользоваться следующими конструкторами (приведена упрощенная запись), аналогичными конструкторам вектора:
explicit deque(); // 1
explicit deque (size_type count);

deque(size__type count, const T& value = T(0)); / / 2
template <class InputIter> / / 3
deque(Inputlter first , InputIter last);
deque(const deque& other); //4

deque(std::initializer_list<T> init); //5 ,*начиная С++11*

Конструктор 1 является конструктором по умолчанию. Конструктор 2 создает очередь длиной ***count*** и заполняет ее одинаковыми элементами — копиями ***value.*** Конструктор 3 создает очередь путем копирования указанного с помощью итераторов диапазона элементов. Тип

итераторов должен быть «для чтения». Конструктор 4 является конструктором копирования.

Пример:

```
#include <deque>
#include <string>
 int main()
{  // C++11 initialize list syntax:
   std::deque<std::string> words1 {"the", "frogurt", "is", "also", "cursed"}; //5
    // words2 == words1
   std::deque<std::string> words2 (words1.begin (), words1.end ()); //3
    // words3 == words1
   std::deque<std::string> words3 (words1); //4
    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
   std::deque<std::string> words4 (5, "Mo"); //2
   std::deque<monstr> ml (10); //1
    return 0;

}
```

В шаблоне ***deque*** определены операция ***присваивания,*** функция ***копирования, итераторы, операции сравнения,*** операции и функции ***доступа к элементам*** *и* ***изменения объектов,*** аналогичные соответствующим операциям и функциям вектора.

Кроме перечисленных, определены функции ***добавления и выборки из начала*** очереди:

void push_front(const T& value);

void pop_front();

При выборке элемент удаляется из очереди. Для очереди не определены функции ***capacity*** и ***reserve***, но есть функции ***resize*** и ***size.***

3. Списки (list)

Список не предоставляет произвольного доступа к своим элементам, зато вставка и удаление работают эффективно для любой позиции элемента в списке, выполняются за постоянное время. Класс ***list*** реализован в ***STL*** в виде двусвязного списка. Каждый узел списка содержит ссылку на последующий и предыдущий элемент списка. Поэтому операции инкремента и декремента для итераторов списка выполняются за постоянное время, а передвижение на n узлов требует времени, пропорционального n. После выполнения операций вставки и удаления значения всех итераторов и ссылок остаются действительными.

Для создания списков можно воспользоваться следующими конструкторами (приведена упрощенная запись), аналогичными конструкторам вектора:

explicit list();
explicit list(size_type count, const T& value = T());

```
template< class InputIt >
list( InputIt first, InputIt last);

list( const list& other );

list( std::initializer_list<T> init );
```

```
#include <list>
#include <string>
int main()
{ // инициализация списка списком инициализации C++11 initializer list syntax:
   std::list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};
    // words2 == words1
   std::list<std::string> words2 (words1.begin(), words1.end());
    // words3 == words1
   std::list<std::string> words3 (words1);
    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
   std::list<std::string> words4 (5, "Mo");
   return 0;
}
```

Список поддерживает ***операцию присваивания, функцию копирования, операции сравнения и итераторы,*** аналогичные векторам и очередям. Доступ к элементам для списков ограничивается следующими методами:

```
reference  front();
const_reference  front() const;
reference  back();
const_reference  back() const;
```

Для занесения элемента в начало или конец списка определены методы, аналогичные соответствующим методам очереди:

```
void push_front(const T& value);
void pop_front();
void push_back(const T& value);
void pop_back();
```

Кроме того, действуют все остальные методы для изменения объектов ***list***, аналогичные векторам и очередям.

Для списка не определена функция ***capacity,*** поскольку память под элементы отводится по мере необходимости. Можно изменить размер списка, удалив или добавив элементы в конец списка аналогично двусторонней очереди:

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

Кроме перечисленных, для списков определено несколько специфических методов.

Сцепка списков (splice) служит для перемещения элементов из одного списка в другой без перераспределения памяти, только за счет изменения указателей:

void splice(iterator position, list<T>& x);
void splice(iterator position, list<T>& x, iterator i) ;
void splice(iterator position, list<T>& x, iterator first , iterator last);

Оба списка должны содержать элементы одного типа. Первая форма функции вставляет в вызывающий список перед элементом, позиция которого указана первым параметром, все элементы списка, указанного вторым параметром, например:

list <int> L1, L2;…
// Формирование списков
L1.splice(L1.begin() + 4, L2);

Второй список остается пустым. Нельзя вставить список в самого себя. Вторая форма функции переносит элемент, позицию которого определяет третий параметр, из списка ***x*** в вызывающий список. Допускается переносить элемент в пределах одного списка.

Третья форма функции аналогичным образом переносит из списка в список несколько элементов. Их диапазон задается третьим и четвертым параметрами функции.

Для ***удаления элемента*** по его значению применяется функция ***remove:***

void remove(const T& value);

Если элементов со значением value в списке несколько, все они будут удалены. Можно удалить из списка элементы, удовлетворяющие некоторому условию. Для этого используется функция remove_if:

template <class Predicate> void remove_if (Predicate pred);

Параметром является функция-предикат, задающая условие, накладываемое на элемент списка.

Для ***упорядочивания*** элементов списка используется метод sort:

void sort():
template <class Compare> void sort(Compare comp)

В первом случае список сортируется по возрастанию элементов (в соответствии с определением операции < для элементов), во втором — в соответствии с функциональным объектом ***Compare***.

Функциональный объект имеет значение ***true***, если два передаваемых ему значения должны при сортировке остаться в прежнем порядке, и ***false*** — в противном случае. Порядок следования элементов, имеющих одинаковые значения, сохраняется. Время сортировки пропорционально $\mathbf{N*\log_2 N}$, где N — количество элементов в списке.

Метод ***unique*** оставляет в списке только первый элемент из каждой серии, идущих подряд одинаковых элементов. Первая форма метода имеет следующий формат:

void unique():

template <class BinaryPrecdicate>
void unique(BinaryPredicate binary_pred);

Вторая форма метода ***unique*** использует в качестве параметра бинарный предикат, что позволяет задать собственный критерий удаления элементов списка. Предикат имеет значение ***true,*** если критерий соблюден, и ***false*** — в противном случае. Аргументы предиката имеют тип элементов списка.

Для ***слияния списков*** служит метод ***merge***

void merge(list<T>& x);
template <class Compare> void merge(list<T>& x, Compare comp);

Оба списка должны быть упорядочены (в первом случае в соответствии с определением операции < для элементов, во втором — в соответствии с функциональным объектом ***Compare***). Результат — упорядоченный список. Если элементы в вызывающем списке и в списке-параметре совпадают, первыми будут располагаться элементы из вызывающего списка.

Метод ***reverse*** служит для ***изменения порядка*** следования элементов списка на обратный: ***void reverse();***

Шаблонная функция *print()* для вывода содержимого контейнера

В процессе работы над программами, использующими контейнеры, часто приходится выводить на экран их текущее содержимое. Приведем шаблон функции, решающей эту задачу для любого типа ***(T)*** контейнера.
//файл ***iterPrint.h***

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

Обратите внимание на служебное слово ***typename***, с которого начинается объявление ***итератора p***. Дело в том, что библиотека STL «знает», что ***T::iterator*** — это некоторый тип, а компилятор C++ таким знанием не обладает. Поэтому без ***typename нормальные компиляторы*** фиксируют ошибку.

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

```
#include <iostream>
#include <deque>
#include  "iterPrint.h"//шаблон функции для вывода контейнера
using namespace std;

int main() {
deque<int> dec; print(dec); // Container is empty
dec.push_back(4); print(dec); //4
dec.push_front(3); print(dec); / / 3 4
dec.push_back(5); print(dec); / / 3 45
dec.push_front(2); print(dec); / / 2 3 4 5
dec.push_back(6); print(dec); / / 2 3 4 5 6
dec.push_front(1); print(dec); / / 1 2 3 4 5 6
return 0;
}
```

К сожалению, инициализация *двусторонней очереди* не поддерживается диапазоном элементов *контейнера другого типа,* то есть определение: ***deque<int>dec(v1.beg(), v1.end())*** не верно.

Адаптеры контейнеров

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

4. Стек

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

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

где параметр ***Container*** задает класс-прототип. По умолчанию для стека прототипом является класс ***deque***. Смысл такой реализации заключается в том, что специализированный класс просто переопределяет интерфейс класса-прототипа, ограничивая его только теми методами, которые нужны новому классу.

В табл. 3 показано, как сформирован интерфейс класса ***stack*** из методов класса-прототипа.

Таблица 3

Интерфейс класса stack

Методы класса stack	***Методы класса-прототипа***
push ()	***push_back()***
pop()	***pop_back()***
top()	***back()***

empty()	***empty()***
size()	***size()***

В соответствии со своим назначением стек не только не позволяет выполнить произвольный доступ к своим элементам, но даже не дает возможности пошагового перемещения, в связи, с чем ***концепция итераторов в стеке не поддерживается.***

Напоминаем, что метод ***pop()*** *не возвращает* удаленное значение. Чтобы считать значение на вершине стека, используется метод ***top().***

Пример работы со стеком — программа вводит из файла числа и выводит их на экран в обратном порядке:

```
int main() {
ifstream in ("inpnum.txt");
stack<int> s;
int x;
while (in >> x) s.push(x);
while (!s.empty()) {
cout<< s.top() << ' ';
s.pop();
}
return 0;
}
```

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

5. Очередь

Шаблонный класс ***queue*** (заголовочный файл ***<queue>***) является адаптером, который может быть реализован на основе двусторонней очереди (реализация по умолчанию) или списка.

Класс ***vector*** в качестве класса-прототипа не подходит, поскольку в нем нет выборки из начала контейнера.

Очередь использует для проталкивания данных один конец, а для выталкивания — другой.

В соответствии с этим ее интерфейс образуют методы, представленные в табл. 4.

Таблица 4.

Интерфейс класса queue

Методы класса queue	**Методы класса-прототипа**
push()	***push_back()***
pop()	***pop_front()***
front()	***front ()***

back()	***back()***
empty()	***empty ()***
size()	***size()***

6.Очередь с приоритетами

Шаблонный класс ***priority_queue*** (заголовочный файл ***<queue>)*** поддерживает такие же операции, как и класс ***queue,*** но реализация класса возможна либо на основе вектора (реализация по умолчанию), либо на основе списка.

Очередь с приоритетами отличается от обычной очереди тем, что для извлечения выбирается максимальный элемент из элементов, хранимых в контейнере. Поэтому после каждого изменения состояния очереди максимальный элемент из оставшихся элементов сдвигается в начало контейнера.

Если очередь с приоритетами организуется для объектов класса, определенного программистом, то в этом классе должна быть определена операция <.

Пример работы с очередью с приоритетами:

```
int  main() {
priority_queue <int> P;
P.push(17); P.push(5); P.push(400);
P.push(2500); P.push(1);
while (!P.empty()) {
cout<< P.front() << ' ';
P.pop();
} return 0;
}
```

Результат выполнения программы:

2500 400 17 5 1

Ассоциативные контейнеры

В ассоциативных контейнерах элементы не выстроены в линейную последовательность. Они организованы в более сложные структуры, что дает большой выигрыш в скорости поиска. Как правило, построены они на основе сбалансированных деревьев поиска (стандартом регламентируется только интерфейс контейнеров, а не их реализация).

Поиск производится с помощью *ключей,* обычно представляющих собой одно числовое или строковое значение.

Существует пять типов ассоциативных контейнеров: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset). Словари часто называют также *ассоциативными массивами* или *отображениями.* В последних версиях стандарта появились и новые версии этих контейнеров, рассматривать их можно самостоятельно в [3].

Словари (map)

Словарь построен на основе пар значений, первое из которых представляет собой ключ для идентификации элемента, а второе — собственно элемент. Можно сказать, что ключ ассоциирован с элементом, откуда и произошло название этих контейнеров. Например, в англо-русском словаре ключом является английское слово, а элементом — русское.

Обычный массив тоже можно рассматривать как словарь, ключом в котором служит номер элемента. В словарях, описанных в ***STL,*** в качестве ключа может использоваться значение произвольного типа.

Ассоциативные контейнеры описаны в заголовочных файлах ***<map>*** и ***<set>.*** Для хранения пары «ключ—элемент» используется шаблон ***pair,*** описанный в заголовочном файле ***<utility>:***

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

Шаблон ***pair*** имеет два параметра, представляющих собой типы элементов пары. Первый элемент имеет имя ***first***, второй — **second.** Определено два конструктора: один должен получать два значения для инициализации элементов, второй (конструктор копирования) — ссылку на другую пару. Конструктора по умолчанию у пары нет, то есть при создании объекта ему требуется присвоить значение явным образом.

Для пары определены проверка на равенство и операция **сравнения** на меньше (все остальные операции отношения генерируются в ***STL*** автоматически на основе этих двух операций). ***Пара p1 меньше пары p2, если p1.first < p2.first или p1.first == p2.first & & p1.second < p2.second.***

Для ***присваивания значения*** паре можно использовать функцию ***make_pair:***

```
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y);
```

Пример формирования пар:

```
#include <iostream>
#include <utility>
using namespace std:
int main(){
pair<int, double> p1(10, 12.3), p2(p1);
p2 = make_pair(20, 12.3);// Эквивалентно p2 = pair <int. double >(20. 12.3)
cout << "p1: " << p1 . first<< " " << p1.second<< endl;
cout << "p2: " << p2.first << " " <<p2.second <<endl;
p2.first -= 10;
```

```
i f (p1 == p2) cout << "p1 == p2\n";
p1.second -= 1;
if (p2 > p1) cout << "p2 > p1\n";
}
```

Результат работы программы:

```
p1: 10 12.3
p2: 20 12.3
p1 == p2
p2 > pi
```

Заголовочный файл **<*utility*>** при использовании **<*map*>** или <***set***> подключается автоматически.

В словаре (map), в отличие от словаря с дубликатами (multimap), все ключи должны быть уникальны. Элементы в словаре хранятся в отсортированном виде, поэтому для ключей должно быть определено отношение «меньше». Шаблон словаря содержит три параметра: тип ключа, тип элемента и тип функционального объекта, определяющего отношение «меньше»

```
template <class Key, class T, class Compare = less<Key> >
class map{
public:
typedef  pair <const Key, T>  value_type;
explicit map(const Compare& comp = Compare());//1
template <class InputIter>
map (InputIter first, InputIter last, const Compare& comp = Compare());//2
map(const map <Key, T, Compare>& x);//3
…
};
```

Как видно из приведенного описания (оно дано с сокращениями), тип элементов словаря ***value_type*** определяется как пара элементов типа ***Key и T***. ***Первый конструктор*** создает пустой словарь, используя указанный функциональный объект. **Второй конструктор** создает словарь и записывает в него элементы, определяемые диапазоном указанных итераторов ***[first, last).*** ***Третий конструктор*** является ***конструктором копирования***.

Как и для всех контейнеров, для словаря определены **деструктор, операция присваивания** и **операции отношения.** Для **доступа** к элементам по ключу определена операция ***[ ]*** :

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

В качестве примера словаря рассмотрим телефонную книгу, ключом в которой служит фамилия, а элементом — номер телефона:

```
#include <fstream>
#include <iostream>
#include <string>
#include <map>
```

```
using namespace std;
typedef  map <string, long,  less <string> > map_sl;// 1

int main(){
map_sl  ml;
ifstream in("phonebook");
string str;
long num;
while (in >> num, !in.eof()){ // Чтение номера
in.get(); // Пропуск пробела
getline (in, str); // Чтение фамилии
m1[str] = num; // Занесение в словарь
cout<< str <<" " << num << endl;
}
m1["Petya P."] = 2134622; // Дополнение словаря
map_sl :: iterator i;
cout << "m1:" << endl; // Вывод словаря
for (i =m1.begin(): i !=m1.end(); i++)
cout << (*i).first << " " <<(*i).second <<endl;
i = m1.begin(); i++; // Вывод второго элемента

cout << "Второй элемент: ";
cout << (*i).first<< " " << (*i).second << endl;

cout << "Vasia: " << m1["Vasia"] << endl; // Вывод элемента по ключу
return 0;
}
```

Сведения о каждом человеке расположены в файле phonebook на одной строке: сначала идет номер телефона, затем через пробел фамилия:

```
1001002  Petya K.
3563398  Ivanova N.M.
1180316  Vovochka
2334476  Vasia
```

Ниже приведен результат работы программы (обратите внимание, что словарь выводится в упорядоченном виде):

```
Petya K.  1001002
Ivanova N.M.  3563398
Vovochka  1180316
Vasia  2334476
m1:
Ivanova N.M.  3563398
Petya K.  1001002
Petya P.  2134622
Vasia  2334476
Vovochka  1180316
```

Второй элемент: Petya K. 1001002
Vasia: 2334476

Для поиска элементов в словаре определен целый ряд функций:

iterator find (const key_type& x);
const_iterator find (const key_type& x) const;
iterator lower_bound (const key_type& x);
const_iterator lower_bound (const key__type& x) const;
iterator upper_bound (const key_type& x);
const_iterator upper_bound (const key_type &x) const;
size_type count (const key_type& x) const;

Функция ***find*** возвращает итератор на найденный элемент в случае успешного поиска, или ***end ()*** в противном случае. Функция ***upper_bound*** возвращает итератор на первый элемент, ключ которого не меньше ***x***, или ***end(),*** если такого нет (если элемент с ключом ***x*** есть в словаре, будет возвращен итератор на него). Функция ***lower_bound*** возвращает итератор на первый элемент, ключ которого больше ***x***, или ***end(),*** если такого нет.

Функция ***count*** возвращает количество элементов, ключ которых равен ***x*** (таких элементов может быть 0 или 1).

Для **вставки и удаления** элементов определены функции:

pair<iterator, bool> insert (const value_type & x);
iterator insert (iterator position, const value_type& x);
template <class InputIter>
void insert (InputIter first. InputIter last);
void erase (iterator position);
size_type erase (const key_type & x);
void erase (iterator first, iterator last);
void clear ();

Словари с дубликатами (multimap)

Как уже упоминалось, словари с дубликатами ***(multimap)*** допускают хранение элементов с одинаковыми ключами. Поэтому для них не определена операция доступа по индексу ***[]***, а добавление с помощью функции ***insert*** выполняется успешно в любом случае. Функция возвращает итератор на вставленный элемент. Элементы с одинаковыми ключами хранятся в словаре в порядке их занесения.

Множество (set)

Во множестве хранятся объекты, упорядоченные по некоторому ключу, являющемуся атрибутом самого объекта. Например, множество может хранить объекты класса ***Man***, упорядоченные в алфавитном порядке по значению ключевого поля ***name.*** Если во множестве хранятся значения одного из встроенных типов, например, ***int,*** то ключом является сам элемент.

Множество — это ассоциативный контейнер, содержащий только значения ключей, то есть тип value_type соответствует типу key_type. Значения ключей должны быть уникальны. Шаблон множества имеет два параметра: тип ключа и тип функционального объекта, определяющего отношение «меньше»:

```
template <class Key, class Compare = less<Key> >
class set {
public:
typedef  Key  key_type;
typedef  Key value_type;
explicit set (const Compare& comp = Compare());
template <class InputIter>
set(InputIter first, InputIter last,
const Compare& comp = Compare());
set(const set<Key, Compare>& x);
pair<iterator, bool> insert(const value_type & x);
iterator insert(iterator position, const value_type& x);
template <class InputIter>
void insert(InputIter first, InputIter last);
void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first , iterator last);

void swap(set<Key,Compare>&);
void clear();
iterator find(const key_type& x) const;
size_type count(const key_type& x) const;
iterator lower__bound(const key_type& x) const;
iterator upper_bound(const key_type& x) const;
pair<iterator, iterator> equal_range(const key_type& x) const;
```

Из описания, приведенного с сокращениями, видно, что интерфейс множества аналогичен интерфейсу словаря. Ниже приведен простой пример, в котором создаются множества целых чисел:

```
#include <iostreani>
#include <set>
using namespace std;
typedef set<int, less<int> > set_i ;
set_i::iterator i ;

int main(){
int a[4] = {4, 2, 1, 2};
set_i  s1; // Создается пустое множество
set_i  s2(a, a + 4); // Множество создается копированием массива
set_i s3(s2); // Работает конструктор копирования
s2.insert(10); // Вставка элементов
```

```
s2.insert(6);
for ( i = s2.begin(); i != s2.end(); i++) // Вывод
cout << *i <<" ";
cout << endl;
// Переменная для хранения результата equal_range;
pair < set_i ::iterator,  set_i::iterator > p;
p = s2.equal_range(2);
cout<< *(p.first) << " " << *(p.second) << endl;
p = s2.equal_range(5);
cout << *(p.first) << " " << *(p.second)<< endl;
return 0;
}
```

Результат работы программы:

```
1 2 4 6 10
2 4
6 6
```

Как и для словаря, элементы в множестве хранятся отсортированными. Повторяющиеся элементы во множество не заносятся.

Множества с дубликатами (multiset)

Во множествах с дубликатами ключи могут повторяться, поэтому операция вставки элемента всегда выполняется успешно, и функция ***insert*** возвращает итератор на вставленный элемент. Элементы с одинаковыми ключами хранятся во множестве в порядке их занесения. Функция ***find*** возвращает итератор на первый найденный элемент или ***end(),*** если ни одного элемента с заданным ключом не найдено.

При работе с одинаковыми ключами в ***multiset*** часто пользуются функциями ***count, lower__bound, upper__bound и equal_range,*** имеющими тот же смысл, что и для словарей с дубликатами.

2.4. Обобщенные алгоритмы STL

Алгоритмы STL предназначены для работы с контейнерами и другими последовательностями. Каждый алгоритм реализован в виде шаблона или набора шаблонов функции, поэтому может работать с различными видами последовательностей и данными разнообразных типов. Для настройки алгоритма на конкретные требования пользователя применяются функциональные объекты. Использование стандартных алгоритмов, как и других средств стандартной библиотеки, избавляет программиста от написания, отладки и документирования циклов обработки последовательностей, что уменьшает количество ошибок в программе, снижает время ее разработки и делает ее более читаемой и компактной.

Объявления стандартных алгоритмов находятся в заголовочном файле ***<algorithm>,*** стандартных функциональных объектов — в файле ***<functional>.***

Все алгоритмы STL можно разделить на четыре категории:

- не модифицирующие операции с последовательностями;
- модифицирующие операции с последовательностями;
- алгоритмы, связанные с сортировкой;
- алгоритмы работы с множествами и словарями.

В качестве параметров алгоритму передаются итераторы, определяющие начало и конец обрабатываемой последовательности. Вид итераторов определяет типы контейнеров, для которых может использоваться данный алгоритм. Например, алгоритм сортировки (***sort)*** требует для своей работы итераторы произвольного доступа, поэтому он не будет работать с контейнером ***list***. Алгоритмы не выполняют проверку выхода за пределы последовательности.

При описании параметров шаблонов алгоритмов используются следующие сокращения:

In — итератор для чтения;
Out — итератор для записи;
For — прямой итератор;
Bi — двунаправленный итератор;
Ran — итератор произвольного доступа;
Pred — унарный предикат (условие);
BinPred — бинарный предикат;
Comp — функция сравнения;
Op — унарная операция;
BinOp — бинарная операция

Не модифицирующие операции с последовательностями.

Алгоритмы этой категории просматривают последовательность, не изменяя ее. Они используются для получения информации о последовательности или для определения положения элемента, представлены в табл. 5.

Таблица 5

Не модифицирующие операции с последовательностями

Алгоритм	Выполняемая функция
adjacent_find	Нахождение пары соседних значений
count	Подсчет количества вхождений значения в последовательность
count_jf	Подсчет количества выполнений условия в последовательности
equal	Попарное равенство элементов двух последовательностей
find	Нахождение первого вхождения значения в последовательность
find_end	Нахождение последнего вхождения одной последовательности в другую

find_first_of	Нахождение первого значения из одной последовательности в другой
find_if	Нахождение первого соответствия условию в последовательности
for_each	Вызов функции для каждого элемента последовательности
mismatch	Нахождение первого несовпадающего элемента в двух последовательностях
search	Нахождение первого вхождения одной последовательности в другую
search_n	Нахождение n-го вхождения одной последовательности в другую

Рассмотрим некоторые их этих алгоритмов подробнее на примерах. Алгоритм **count** подсчитывает количество вхождений в контейнер (или его часть) значения, заданного его третьим аргументом. Алгоритм **find** выполняет поиск заданного значения и возвращает итератор на самое первое вхождение этого значения. Если значение не найдено, то возвращается итератор, соответствующий возврату метода **end().** В следующей программе показано использование этих алгоритмов.

```
#include <algorithm>
#include  "iterPrint.h"//шаблон функции для вывода контейнера
using namespace std;

int main() {
int arr[ ]= {1, 2, 3, 4, 5, 2, 6, 2, 7};
int n = sizeof(arr) / sizeof(int);
vector<int> v1(arr, arr + n);
int value = 2 ; // искомая величина
int how_much = count(v1.begin(), v1.end(), value);
cout<< how_much << endl; // вывод: 3
list<int> loc_list;// список позиций искомой величины
vector<int>: :iterator location = v1.begin();
while (1) {
location = find(location,  v1.end(),  value);
if (location == v1.end()) break;
loc_list.push_back(location – v1.begin());
location++;
}
print(loc_list); // вывод: 1 5  7
return 0;
}
```

В приведенной программе создается вектор ***v1***, наполняясь при инициализации значениями из массива ***arr***. Затем с помощью алгоритма ***count*** подсчитывается количество вхождений в вектор значения ***value***, равного двум. В цикле ***while*** выясняется, на каких позициях в векторе

размещена эта величина. Обратите внимание на то, что первый аргумент алгоритма ***find*** (переменная ***location***) первоначально — перед входом в цикл — принимает значение итератора, указывающего на нулевой элемент контейнера. Затем ***location*** получает значение итератора, указывающего на найденный элемент. Если поиск завершился успешно, то, во-первых, вычисляется позиция найденного элемента как разность значений ***location*** и адреса нулевого элемента. Полученное значение заносится в список ***loc_list*** .

Во-вторых, итератор ***location*** сдвигается операцией инкремента па следующую позицию в контейнере, чтобы обеспечить (на следующей итерации цикла) продолжение поиска в оставшейся части контейнера.

Если поиск завершился неудачей, то ***break*** приведет к выходу из цикла.

Алгоритмы count_if и find_if

Алгоритмы ***count_if и find_if*** отличаются от алгоритмов ***count и find*** тем, что в качестве третьего аргумента они требуют некоторый предикат.

Предикат — это функция, возвращающая значение типа ***bool***. Например, если в предыдущей программе добавить определение глобальной функции:

bool isMyValue (int x) { return ((x> 2) && (x< 5)); }

и заменить инструкцию с вызова ***count на вызов count_if*** :

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

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

Аналогичная замена инструкции вызова ***find*** на ***find_if*** :

location = find_if (location, v1.end(), isMyValue); позволила наполнение списка ***loc_list*** двумя значениями: 2 и 3(номера позиций вектора ***v1***, на которых находятся числа, удовлетворяющие предикату ***isMyValue***).

Алгоритм for_each

Вызов функции для каждого элемента последовательности (***[fist, last)***). Нужно определить, соответствующую функцию с одним аргументом типа ***T (T*** — тип данных, содержащихся в контейнере). Функция не имеет права модифицировать данные в контейнере, но может их использовать в своей работе. Имя этой функции передается в качестве третьего аргумента алгоритма.

Например, в следующей программе ***for_each*** используется для перевода всех значений массива из дюймов в сантиметры и вывода их на экран.

```
void InchToCm (double inch) {
cout<< (inch * =2.54) << ' ';}
int main(){
double inches [ ] = {0.5, 1.0, 1.5, 2.0, 2.5};
for_each (inches, inches + 5, InchToCm);
return 0;
}
```

Алгоритм search

Некоторые алгоритмы оперируют одновременно двумя контейнерами. Таков и алгоритм ***search,*** который находит первое вхождение в первую последовательность *[first1, last1)* второй последовательности *[first2,last2).*

Например:

```
int main() {
int arr[] = {11, 77, 33, 11, 22, 33, 11, 22, 55};
int pattern [] = { 11, 22, 33 };
int* ptr = search(arr, arr + 9, pattern, pattern + 3);
if (ptr == arr + 9)
cout<< "Pattern not found" << endl;
else
cout<< "Found at position " << (ptr - arr) <<endl;
list<int> lst (arr, arr + 9);
1ist<int>:: iterator ifound;
ifound = search (lst.begin(), lst.end(), pattern, pattern + 3);
if (ifound == lst.end())
cout<< "Pattern not found" << endl;
else
cout<< "Found." <<endl;
return 0;
}
```

Результат выполнения программы:

```
Found at position 3
Found.
```

Отметим, что список не поддерживает произвольного доступа к своим элементами соответственно не допускает операций *«+»* и *«-»* с итераторами. Поэтому мы можем только зафиксировать факт вхождения последовательности ***pattern*** в контейнер ***lst***.

Модифицирующие операции с последовательностями

Алгоритмы этой категории тем или иным образом изменяют последовательность, с которой они работают. Они используются для копирования, удаления, замены и изменения порядка следования элементов последовательности.

Таблица 6

Модифицирующие операции с последовательностями

Алгоритм	**Выполняемая функция**
copy, copy_ backward	Копирование последовательности, начиная с первого элемента, Копирование последовательности, начиная с последнего элемента
fill, fill_n	Замена всех элементов заданным значением, Замена первых n элементов заданным значением
generate, generate_n	Замена всех элементов результатом операции,

	Замена первых п элементов результатом операции
iter_swap	Обмен местами двух элементов, заданных итераторами
random_shuffle	Перемещение элементов в соответствии со случайным равномерным распределением
remove, remove_copy, remove_copy_if, remove_if	Перемещение элементов с заданным значением, Копирование последовательности с перемещением элементов с заданным значением, Копирование последовательности с перемещением элементов при выполнении предиката, Перемещение элементов при выполнении предиката
replace, replace_copy, replace_copy_if, replace_if	Замена элементов с заданным значением, Копирование последовательности с заменой элементов с заданным значением, Копирование последовательности с заменой элементов при выполнении предиката, Замена элементов при выполнении предиката
reverse, reverse_copy	Изменение порядка элементов на обратный, Копирование последовательности в обратном порядке
rotate, rotate_copy	Циклическое перемещение элементов последовательности, Циклическое копирование элементов
swap, swap_range	Обмен местами двух элементов. Обмен местами элементов двух последовательностей
transform	Выполнение заданной операции над каждым элементом последовательности
unique, unique_copy	Удаление равных соседних элементов, Копирование последовательности с удалением равных соседних элементов

Рассмотрим эти алгоритмы подробнее на примерах.

Пример применения remove_if совместно с методом erase для удаления элементов вектора, значения которых лежат от 10 до 50:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool In_10_50 (int x) {return x > 10 && x < 50;}
int main(){
vector<int> a;
```

```
int i;
for (i = 1; i<10; i++) a.push_back(i*10);
for (i = 0; i<a.size(); i++) cout << a[i] << " ";
cout << endl;
vector<int>::iterator new_end = remove_if(a.begin(), a.end(), In_10_50);
a.erase(new_end, a.end());
for (i = 0; i<a.size(); i++) cout << a[i] << " ";
cout << endl;
return 0;}
```

Результат работы программы*:*

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

Алгоритм transform

Алгоритм ***transform*** выполняет заданную операцию над каждым элементом последовательности. Первая форма алгоритма выполняет унарную операцию, заданную функцией или функциональным объектом op, и помещает результат в место, заданное итератором ***result***:

```
template <class In, class Out, class Op>
Out transform (In first, In last, Out result, Op op);
```

Вторая форма алгоритма выполняет бинарную операцию над парой соответствующих элементов двух последовательностей и помещает результат в место, заданное итератором ***result:***

```
template <class Inl , class In2, class Out, class BinaryOperation>
Out  transform(In1  first1 , In1 last1 , In2 first2, Out result, BinaryOperation
binary_op);
```

Функциональный объект может быть стандартным или заданным пользователем.

В приведенном ниже примере первый вызов ***transform*** выполняет преобразование массива ***a*** по формуле: $a_i = a_i^2 - b_i^2$, второй вызов меняет знак у элементов массива ***b*** с помощью стандартного функционального объекта ***negate***.

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
struct preobr: binary_function <double, double, double>{
double operator()(double x, double y) const{
return x * x - y * y;}
};
int main(){
const int m = 5;
int  i;
double a[m] = {5, 3, 2, 3,1 },
b[m] = ( 1 , 10, -3, 2, -4};
```

```
transform(a,  a + m, b, a, preobr());
transform(b, b + m, b, negate<double>());
for (i = 0; i<m; i++) cout <<a[i] << " ";
cout << endl;
for (i = 0; i<m; i++) cout <<b[i] << " ";
cout <<endl;
return 0;}
```

Результат работы программы:

```
24 -91 -5 5 -15
-1 -10 3 - 2 4
```

Итераторы вставки и алгоритм copy

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

```
#include  "iterPrint.h"
int main() {
int a[4] = {10, 20, 30, 40};
vector<int> v(a, a + 4);
list<int>L(4); // список из 4 элементов
copy(v.begin(), v.end(), L.begin());
print(L);
return 0;
}
```

Алгоритм ***copy*** при таком использовании, как в этом примере, работает в ***режиме замещения***. Это означает, что 1 -й элемент контейнера-источника замещает 1 -й элемент контейнера-приемника. Однако этот же алгоритм может работать и ***в режиме вставки***, если в качестве третьего аргумента использовать так называемый ***итератор вставки.***

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

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

advance(from, 4);
copy(vc.begin(), vc.end(), inserter(L, from));
print(L);
return 0;}
Результат выполнения программы:
10 20 30 40
10 20 30 40 80 90 100
10 20 30 40 50 60 70 80 90 100

Обратите внимание на следующие моменты:
• Первый вызов функции copy осуществляет копирование (вставку) вектора va в список L, причем итератор вставки ***front_inserter*** обеспечивает размещение очередного элемента вектора va в начале списка — поэтому порядок элементов в списке изменяется на обратный.
• Второй вызов copy пересылает элементы вектора vb в конец списка L благодаря итератору вставки ***back_inserter***, поэтому порядок копируемых элементов не меняется.
• Третий вызов copy копирует вектор vc в заданное итератором from место списка L, а именно после четвертого элемента списка. Чтобы определить нужное значение ***итератора from***, мы предварительно устанавливаем его в начало списка, а затем обеспечиваем приращение на 4 — вызовом функции ***advance().***

Алгоритмы, связанные с сортировкой

Алгоритмы этой категории упорядочивают последовательности, выполняют поиск элементов, слияние последовательностей, поиск минимума и максимума, лексикографическое сравнение, перестановки и т. п

Алгоритм sort

Назначение алгоритма очевидно из его названия. Алгоритм можно применять только для тех контейнеров, которые обеспечивают произвольный доступ к элементам, — этому требованию удовлетворяют ***массив, вектор*** **и** ***двусторонняя очередь,*** но не удовлетворяет ***список.***

В связи с этим класс ***list*** содержит ***метод sort(),*** решающий задачу сортировки.

Алгоритм ***sort*** имеет две сигнатуры:

template<class RandomAccessIt> void sort(RandomAccessIt fist, RandomAccessIt last);

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

Первая форма алгоритма обеспечивает сортировку элементов из диапазона ***[first,last),*** причем для упорядочения по умолчанию используется операция <, которая должна быть определена для типа ***T. T*** — тип данных, содержащихся в контейнере. Таким образом, сортировка по умолчанию— это сортировка по возрастанию значений. Например,

int main() {

```
double arr[6] = {2.2, 0.0, 4.4, 1.1, 3.3, 1.1};
vector<double> v1(arr, arr + 6);
sort (v1.begin(), v1.end());
print(v1);
return 0;}
```

Результат выполнения программы:

0 1.1 1.1 2.2 3.3 4.4

Вторая форма алгоритма ***sort*** позволяет задать произвольный критерий упорядочения. Для этого нужно передать через третий аргумент соответствующий предикат - функцию или функциональный объект, возвращающий значение типа ***bool***. Использование функции в качестве предиката было показано выше. Использованию функциональных объектов посвящен следующий раздел.

Функциональные объекты

Функциональным объектом называется объект некоторого класса, для которого определена единственная операция вызова функции ***operator().***

В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения, встроенных в язык С++. Они возвращают значение типа ***bool***, то есть являются предикатами (табл. 7).

Таблица 7

Предикаты стандартной библиотеки

Операция	предикат (функциональный объект)
==	***equal_to***
!=	***not_equal_to***
>	***greater***
<	***less***
>=	***greater_equal***
<=	***less_equal***

Очевидно, что при подстановке в качестве аргумента алгоритма требуется инстанцирование этих шаблонов, например: ***equal_to<int>().***

Вернемся к последней программе, где с помощью алгоритма ***sort*** был отсортирован вектор ***v1***. Заменим вызов ***sort*** на следующий:

sort(v1.begin(), v1.endO, greater<double>());

В результате вектор будет отсортирован по убыванию значений его элементов.

Несколько сложней обстоит дело, когда сортировка выполняется для контейнера с объектами пользовательского класса. В этом случае программисту нужно самому позаботиться о наличии в классе ***предиката***, задающего сортировку по умолчанию, а также (при необходимости) определить функциональные классы, объекты которых позволяют изменять настройку алгоритма ***sort***.

Сортировка вектора

В приведенной ниже программе показаны варианты вызова алгоритма ***sort*** для вектора ***men***, содержащего объекты класса ***Man***.

В классе ***Man*** определен ***предикат — операция operator<()***, — благодаря которому сортировка по умолчанию будет происходить по возрастанию значений поля ***name***.

Кроме этого, в программе определен функциональный класс ***Less Age***, использование которого позволяет осуществить сортировку по возрастанию значений поля ***age***.

```
char s[80]; int st;
class Man {
public:
Man (string  _name, int  _age);
name(_name), age(_age) { }
// предикат, задающий сортировку по умолчанию
bool operator< (const Man& m) const {
return name < m.name;
}
friend ostream& operator<< (ostream&, const Man&);
friend struct LessAge;
private:
string name;
int age;
};
ostream& operator<<(ostream& os, const Man& m) {
return os << endl << m.name << ",\t age: " << m.age;
}
// Функциональный класс для сравнения по возрасту
struct LessAge {
bool operator() (const Man& a, const Man& b) {
return a.age < b.age;}
};
#include  "iterPrint.h"
int main() {
Man ar []= {
Man("Mary Poppins", 36),
Man("Count Basie",70),
Man("Duke Ellington", 90).
Man("Joy Amore", 18)
};
int size = sizeof(ar) / sizeof(Man);
vector<Man> men(ar, ar + size);
// Сортировка по имени (по умолчанию)
sort(men.begin(), men.end());
print(men);
```

```
// Сортировка по возрасту
sort(men.begin(), men.end(), LessAge());
print(men);
return 0;}
```

Алгоритм merge

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

```
#include  "iterPrint.h"
int main() {
int arr[5] = {2, 3, 8, 20, 25};
vector<int> a(arr. arr+ 5);
int b[6] = {7, 9, 23,28, 30, 33};
list<int> c; // Список с сначала пуст
merge(a.begin(), a.end(), b, b + 6, back_inserter(c));
print(c);
return 0;}
```

Результат выполнения программы:

```
2 3 7 8 9 20 23 25 28 30 33
```

Использование ассоциативных контейнеров

В ассоциативных контейнерах элементы не выстроены в линейную последовательность. Они организованы в более сложные структуры, что дает большой выигрыш в скорости поиска.

Рассмотрим две основные категории ассоциативных контейнеров в STL: множества и словари.

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

Словарь (map) можно представить себе как своего рода таблицу из двух столбцов, в первом из которых хранятся объекты, содержащие ключи, а во втором — объекты, содержащие значения.

И в множествах, и в словарях все ключи являются уникальными (только одно значение соответствует ключу).

Мультимножества (multiset) и мультисловари (multlmap) аналогичны своим родственным контейнерам, но в них одному ключу может соответствовать несколько значений.

Ассоциативные контейнеры имеют много общих методов с последовательными контейнерами. Тем не менее, некоторые методы, а также алгоритмы характерны только для них.

Множества

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

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

Таким образом, если объявить некоторое множество ***set<int> s1*** с опущенным вторым параметром шаблона, то по умолчанию для упорядочения членов множества будет использован ***предикат less<int>.***

Точно так же можно опустить второй параметр при объявлении множества ***set<MyClass>s2***, если в классе ***MyClass*** определена операция ***operator<().*** Для использования контейнеров типа ***set*** необходимо подключить заголовочный файл ***<set>.***

Имеется три простых способа определить объект типа set:

```
set<int> set1; // создается пустое множество
int a[5] = { 1, 2, 3, 4, 5 };
set<int> set2(a, a + 5); // инициализация копированием массива
set<int> set3(set2); // инициализация другим множеством
```

Для вставки элементов в множество можно использовать метод ***insert (),*** для удаления — метод ***erase ().*** Также к множествам применимы общие для всех контейнеров методы, указанные в табл. 2.

Во всех ассоциативных контейнерах есть метод ***count(),*** возвращающий количество объектов с заданным ключом. Так, как и в множествах, и в словарях все ключи уникальны, то метод ***count ()*** возвращает либо ***0***, если элемент не обнаружен, либо ***1***.

Для множеств библиотека содержит некоторые специальные алгоритмы, в частности, реализующие традиционные теоретико-множественные операции. Некоторые алгоритмы перечислены ниже.

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

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

Алгоритм ***set_union*** создает отсортированное ***объединение множеств***, то есть множество, содержащее элементы первого и второго множества без повторяющихся элементов.

Следующая программа демонстрирует работу описанных алгоритмов:

```
#include  "iterPrint.h"
int main() {
const int N = 5;
string sl[N] = {"Bill", "Jessica", "Ben", "Mary", "Monica"};
string s2[N] = {"Sju", "Monica","John","Bill","Sju"};
```

```
typedef set<string> Sets;
Sets A(s1, s1 + N);
Sets B(s2, s2 + N);
print(A); print(B);
Sets prod, sum;
set_intersection(A.begin(), A.end(), B.begin(), B.end(),
inserter(prod, prod.begin());
print(prod);
set_union(A.begin(),A.end(), B.begin(),B.end(),
inserter(sum, sum.begin()));
print(sum);
if (includes(A.begin(), A.end(), prod.begin(), prod.end()))
cout<< "Yes" << endl;
else cout << "No" << endl;
return 0;
}
```

Результат выполнения программы:

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

Словари

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

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

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

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

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

Обратите внимание на то, что тип элементов словаря ***value_type*** определяется как пара элементов типа ***Key*** и ***T***.

Первый конструктор класса ***map*** создает пустой словарь.

Второй конструктор создает словарь и записывает в него элементы, определяемые диапазоном *[first,last).*

Третий конструктор является конструктором копирования.

Для доступа к элементам по ключу определена операция []:

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

с помощью нее можно не только получать значения элементов, но и добавлять в словарь новые.

Для использования контейнеров типа ***map*** необходимо подключить заголовочный файл ***<map>.***

Воспользуемся возможностями контейнера ***map*** для написания программы формирования частотного словаря появления отдельных слов в некотором тексте. Исходный текст читается из файла ***text.txt***, результат — частотный словарь — записывается в файл ***freq_map.txt***

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <map>
#include <set>
#include <string>
using namespace std;
int main() {
char punct[6] = { ' . ' , ' , ' , ' ? ' , ' ! ' , ' : ' , ' ; ' } ;
set<char> punctuation(punct, punct + 6);
ifstream in("text.txt");
if (!in) { cerr <<"File not found\n"; exit(1); }
map<string, int> wordCount;
string s;
while (in >> s) {
int n = s.size();
if (punctuation.count(s[n - 1])
s.erase(n – 1, n);
++wordCount[s];
}
ofstream out("freq_map.txt");
map<string, int>::const_iterator it = wordCount.begin();
for ( it ; it != wordCount.end(); ++it)
out << setw(20) << left << it - > first
<<setw(4)<< right << it->second <<endl;
return 0:
}
```

Определяя в этой программе объект ***wordCount*** как словарь типа ***map<string,int>,*** мы тем самым показываем наше намерение связать каждое прочитанное слово с целочисленным счетчиком.

В цикле ***while*** разворачиваются следующие события:

- В строку ***s*** пословно считываются данные из входного файла.
- Определяется длина ***n*** строки ***s***.
- С помощью метода ***count ()*** проверяется, принадлежит ли последний символ строки ***s*** множеству ***punctuation***, содержащему знаки препинания, которыми может завершаться слово. Если да, то последний символ удаляется из строки (метод ***erase()).***
- Заслуживает особого внимания лаконичная инструкция ***++wordCount[s].*** Здесь мы как бы «заглядываем» в объект ***wordCount***, используя только что считанное слово в качестве ключа. Результат выражения ***wordCount[s]*** представляет собой некоторое целочисленное значение, обозначающее, сколько раз слово ***s*** уже встречалось ранее. Затем операция инкремента увеличивает это целое значение на единицу. А что будет, если мы встречаем некоторое слово в первый раз? Если в словаре нет элемента с таким ключом, то он будет создан с инициализацией поля типа ***int*** значением по умолчанию, то есть нулем. Следовательно, после операции инкремента это значение будет равно единице.
- Завершив считывание входных данных и формирование словаря ***wordCount***, мы должны вывести в выходной файл ***freq_map.txt*** значения обнаруженных слов и соответствующих им счетчиков.
- Вывод результатов реализуется здесь практически так же, как и для последовательных контейнеров — с помощью соответствующего итератора.
- Однако есть одна тонкость, связанная с тем, что при разыменовании итератора ***map-объекта*** мы получаем значение, которое имеет тип ***pair***, соответствующий данному ***map-объекту***. Так как ***pair —*** это структура, то доступ к полям структуры через «указатель» it осуществляется посредством выражений ***it->first, it->second.***

ЛИТЕРАТУРА

1. Страуструп Б. Язык программирования С++. Специальное издание. - Издательство: "Бином", 2011, ISBN: 978-5-9518-0425-9, 1136 с.

2. Подбельский В.В. Стандартный Си++: учебное пособие – М: Финансы и статистика, 2008, 688 с.

3. Страуструп Б. Дизайн и эволюция языка С++. Издательство: "ДМК-Пресс" ,2014, ISBN: 978-5-94074-994, 488 с.

4. Л.А. Надейкина Программирование: учебное пособие - М.: МГТУ ГА, 2017 84 с.

5. Дэвид Вандевурд, Николаи М. Джосаттис. Шаблоны С++. Справочник разработчика / C++ Templates: The Complete Guide. — М.: Вильямс. — 2008, 544 с.

6. Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. — Addison-Wesley Professional. — 2001, 352 p.

СОДЕРЖАНИЕ