Лабораторная работа №7
Библиотека STL. Класс String и файловые потоки

Файловые и строковые потоки. Строки класса string

Файловые потоки

Для поддержки файлового ввода и вывода стандартная библиотека C++ содержит классы, указанные в табл. 1.

Таблица 1. Классы файловых потоков

Класс	Инстанцирован из шаблона	Базовый шаблонный класс	Назначение
ifstream	basic_ifstream	basic_istream	Входной файловый поток
ofstream	basic_ofstream	basic_ostream	Выходной файловый поток
fstream	basic_fstream	basic_iostream	Двунаправленный файловый поток

Так как классы файловых потоков являются производными от классов istream, ostream и iostream соответственно, то они наследуют все методы указанных классов, перегруженные операции вставки и извлечения, манипуляторы, состояние потоков и т.д. Как и стандартные потоки, файловые потоки обеспечивают гораздо более надежный ввод-вывод, чем старые функции библиотеки C. Для использования файловых потоков необходимо подключить заголовок <fstream>. Работа с файлом обычно предполагает следующие операции:

- создание потока (потокового объекта);
- открытие потока и связывание его с файлом;
- обмен с потоком (ввод-вывод);
- закрытие файла.

Классы файловых потоков содержат несколько конструкторов, позволяющих варьировать способы создания потоковых объектов. *Конструкторы с параметрами* создают объект соответствующего класса, открывают файл с указанным именем и связывают файл с объектом:

```
ifstream( const char* name, int mode = ios::in );
ofstream( const char* name, int mode = ios::out | ios::trunc );
fstream ( const char* name, int mode = ios::in | ios::out );
```

Второй параметр конструктора задает режим открытия файла. Если значение по умолчанию вас не устраивает, можно указать другое, выбрав одно или несколько значений (объединенных операцией |) из указанных в табл. .2.

Таблица .2. Значения аргумента mode

Флаг	Назначение
ios::in	Открыть файл для ввода
ios::out	Открыть файл для вывода
ios::ate	Установить указатель на конец файла
ios::app	Открыть в режиме добавления в конец файла
ios::trunk	Если файл существует, обрезать его до нулевой длины
ios::binary	Открыть в двоичном режиме (по умолчанию используется текстовый)

Конструкторы без параметров создают объект соответствующего класса, не связывая его с файлом. В этом случае связь потока с конкретным файлом осуществляется позже — вызовом метода open, который имеет параметры, аналогичные параметрам

рассмотренных выше конструкторов. Приведем примеры создания потоковых объектов и связывания их с конкретными файлами:

```
ofstream flog( "flog.txt" );
ofstream foutl, fout2;
 fout1.open( "test1", ios::app );
 fout2.open( "test2", ios::binary );
ifstream finp1( "data.txt" );
fstream myfile; myfile.open( "mf.dat" );
```

Если в качестве параметра name задано *краткое имя файла* (без указания полного пути), подразумевается, что файл открывается в текущем каталоге, в противном случае требуется задавать *полное имя файла,* например:

```
ifstream finp1( "D:\\VCwork\\Task1\\data.txt" );
```

После того как файловый поток открыт, работа с ним чрезвычайно проста: с входным потоком можно обращаться так же, как со стандартным объектом cin, а с выходным так же, как со стандартным объектом cout.

Если при чтении данных требуется контролировать, был ли достигнут *конец файла* после очередной операции ввода, используется метод eof, возвращающий нулевое значение, если конец файла еще не достигнут, и ненулевое значение — если уже достигнут. Учтите, что в C++ после чтения из файла *последнего элемента* условие конца файла *не возникает*! Оно возникает при следующем чтении, когда программа пытается считать данные за последним элементом в файле.

Если при выполнении операций ввода-вывода фиксируется некоторая ошибочная ситуация, потоковый объект также принимает значение, равное нулю. Рекомендуется особо следить за состоянием потокового объекта во время выполнения операций вывода, так как диски «не резиновые» и имеют тенденцию переполняться. Когда программа покидает область видимости потокового объекта, он уничтожается. При этом перестает существовать связь между потоковым объектом и физическим файлом, а физический файл закрывается. Если алгоритм требует более раннего закрытия файла, можно воспользоваться методом close.

Для примера работы с файловыми потоками приведем программу копирования одного файла в другой (листинг 1) и программу вывода на экран содержимого текстового файла (листинг 2).

Листинг 1. Копирование файлов

```
#include <iostream>
#include <fstream>
using namespace std;
void error(const char* textl, const char* text2 = "")
{
cerr << textl << ' ' << text2 << endl;
 cin.get();
exit(l);
}
int main(int argc, char* argv[])
{ // Имена файлов берутся из командной строки
if (argc != 3) error("Неверное число аргументов");
```

```
ifstream from(argv[1]);                    // открываем входной файл
if (!from) error("Входной файл не найден:", argv[1]);
ofstream to(argv[2]);                      // открываем выходной файл
if (!to) error("Выходной файл не открыт:", argv[2]); char ch;
while ( from.get(ch) )
{
to.put(ch);
if (!to) error("Ошибка записи (диск переполнен).");
}
cout << "Копирование из " << argv[1] << " в " << argv[2] << " завершено.^";
cin.get();
}
```

Листинг 2. Вывод на экран содержимого текстового файла

```
#include <iostream>
#include <fstream>
using namespace std;
// ... <- Здесь определение функции error() - из листинга 1
int main(int argc, char* argv[])
{ // имя файла задается в командной строке
if (argc != 2) error("Неверное число аргументов");
ifstream tfile(argv[1]);                   // открываем входной файл
if (!tfile) error("Входной файл не найден:", argv[1]);
char buf[1024];
while (!tfile.eof())
{
tfile.getline(buf, sizeof(buf));
cout << buf << endl;
}
cin.get();
}
```

В программе предполагается, что длина строки в файле не превышает 1024 символов. При необходимости можно увеличить размер буфера buf до требуемой величины.

Строковые потоки

Работу со строковыми потоками обеспечивают классы istringstream, ostringstream и stringstream, которые являются производными от классов istream, ostream и i ostream соответственно. Для использования строковых потоков необходимо подключить к программе заголовочный файл <sstream>.

Применение строковых потоков аналогично применению файловых потоков, но информация потока физически размещается в оперативной памяти, а не на диске. Кроме того, классы строковых потоков содержат метод str, возвращающий копию строки типа string или присваивающий потоку значение такой строки:

```
string str() const;
void str( const string& s );
```

Строковые потоки являются некоторыми аналогами функций sscanf и sprintf библиотеки C, которые также работают со строками в памяти, имитируя консольный ввод-вывод. Например, с помощью sprintf можно сформировать в памяти некоторую символьную строку, которую затем отобразить на экране. Эта же проблема легко решается с помощью объекта типа ostringstream.

В качестве примера приведем модифицированную версию листинга 2, которая выводит содержимое текстового файла на экран, предваряя каждую строку текстовой меткой «Line N:», где N — номер строки (листинг 3).

Листинг 3. Вывод на экран пронумерованного содержимого текстового файла

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
using namespace std;
// ... <- Здесь определение функции error() - из листинга 1
int main(int argc, char* argv[])
{
if (argc != 2) error("Неверное число аргументов.");
 ifstream tfile( argv[1] );
if (!tfile) error("Входной файл не найден:", argv[1]);
int n = 0;
char buf[1024];
while ( !tfile.eof() )
{
n++;
tfile.getline(buf, sizeof(buf));
ostringstream line;
line << "Line " << setw(3) << n << ": " << buf << endl;
cout << line.str();
}
cin.get();
```

Строки класса string

Мы уже пользовались объектами класса string, начиная со второго семинара, и успели оценить удобства, обеспечиваемые этим классом в сравнении с традиционными C-строками (то есть массивами символов типа char, завершаемыми нулевым байтом). Сейчас мы рассмотрим строки типа string более подробно. Для использования строк типа string необходимо подключить к программе заголовочный файл <string>. Важнейшей особенностью класса string является управление памятью как при размещении строки, так и при ее модификациях, изменяющих длину строки. Поэтому вы можете «забыть» об операциях new и delete, неаккуратное обращение с которыми является источником труднодиагностируемых ошибок. Кроме этого, строки типа string защищены от ошибочных обращений к памяти, связанных с выходом за их границы. Но за все надо платить: строки типа string значительно проигрывают C-строкам в эффективности. Поэтому, если от программы требуется максимальное быстродействие, иногда лучше воспользоваться C-строками. В большинстве же программ на C++ строки типа string

обеспечивают необходимую скорость обработки, поэтому их применение предпочтительней. Для понимания определений методов класса string необходимо знать назначение некоторых имен. Так, в пространстве std определен идентификатор size_type, являющийся синонимом типа unsigned int. В классе string определена константа npos, задающая максимально возможное число, которое в зависимости от контекста означает либо «все элементы» строки, либо отрицательный результат поиска. Так как максимально возможное число имеет вид 0xFFFF...FFFF, то в случае присваивания его переменной типа int получится значение -1.

В классе string имеется несколько конструкторов, вот самые простые:

string(); // создает пустой объект класса string

string(const char*); // создает объект, инициализируя его значением С-строки

Класс содержит три операции присваивания:

string& operator=(const string& str); // присвоить значение другой строки string

string& operator=(const char* s); // присвоить значение С-строки

string& operator=(char c); // присвоить значение символа

В табл. 3 приведены допустимые для объектов класса string операции.

Таблица 3. Операции класса string

Операция	Действие	Операция	Действие
=	Присваивание	>	Больше
+	Конкатенация	>=	Больше или равно
==	Равенство	[ ]	Индексация
1 =	Неравенство	<<	Вывод
<	Меньше	>>	Ввод
<=	Меньше или равно	+=	Добавление

Использование операций очевидно. Размеры строк устанавливаются автоматически так, чтобы объект мог содержать присваиваемое ему значение. Нумерация элементов строки начинается с нуля. Кроме операции индексации, для доступа к элементу строки определен метод at(size_type n), который можно использовать как для чтения, так и для записи n-го элемента строки:

```
cout << s.at(2);
s.at(5) = 'W';          // Будет выведен 2-й символ строки s // 5-й символ заменяется символом W
```

Заметим, что в операции индексации не проверяется выход за диапазон строки. Метод at, напротив, такую проверку содержит, и, если индекс превышает длину строки, порождается исключение out_of_range.

В табл. 4 приведены некоторые употребительные методы класса string

Таблица4. Методы класса string

Метод	Назначение
size_type size() const:	Возвращает размер строки
size_type length() const	То же, что и size
insert(size_type posl, const string& str):	Вставляет строку str в вызывающую строку, начиная с позиции posl
replace(size_type posl, size_type n1, const string& str);	Заменяет nl элементов, начиная с позиции posl вызывающей строки, элементами

	строки str
string substr(size_type pos=0, size_type n=npos) const;	Возвращает подстроку длиной n, начиная с позиции pos
size_type find(const string& str, size_type pos=0) const;	Ищет самое левое вхождение строки str в вызывающую строку, начиная с позиции pos. Возвращает позицию вхождения, или npos, если вхождение не найдено
size_type find(char c, size_type pos=0) const:	Ищет самое левое вхождение символа c, начиная с позиции pos. Возвращает позицию вхождения, или npos, если вхождение не найдено
_type rfind(const string& str, size_type pos=0) const;	
size_type rfind(char c, size_type pos=0) const;	
size_type find_first_of(const string& str, size_type pos=0) const;	
size_type find_last_of(const string& str, size_type pos=0) const;	
swap(string& str);	
erase(size_type pos=0, size_type n=npos);	
clear();	
const char* c str() const	
Size_type copy(char* s, size_type n, size_type pos=0) const;	

Задача 1. Подсчет количествавхождений слова в текст.

Написать программу, которая определяет, сколько раз встретилось заданное слово в текстовом файле. Текст не содержит переносов слов. Максимальная длина строки в файле неизвестна.

Определим *слово* как последовательность алфавитно-цифровых символов, после которых следует либо знак пунктуации (., „ ?, !), либо разделитель. В качестве разделителей могут выступать один или несколько пробелов, один или несколько символов табуляции '\t' и символ конца строки '\n'. Для хранения заданного слова (оно вводится с клавиатуры) определим переменную word типа string. Поскольку максимальная длина строки в файле неизвестна, будем читать файл не построчно, а пословно, размещая очередное прочитанное слово в переменной curword типа string. Это можно реализовать с помощью операции >>, которая в случае операнда типа string игнорирует все разделители, предваряющие текущее слово, и считывает символы текущего слова в переменную curword, пока не встретится очередной разделитель.

Очевидно, что «опознание» текущего слова должно осуществляться с учетом возможного наличия после него одного из знаков пунктуации. Для решения этой задачи определим глобальную функцию

bool equal(const string& cw, const string& w);

которая возвращает значение true, если текущее слово cw совпадает с заданным словом w с точностью до знака пунктуации, или false — в противном случае. Имея такую функцию, очень просто составить алгоритм основного цикла:

- прочитать очередное слово;
- если оно совпадает с заданным словом w (с точностью до знака пунктуации), то увеличить на единицу значение счетчика count.

Текст решения приведен в листинге 4.

Листинг4. Подсчет количества вхождений слова в текст

```
#include <windows.h>          // здесь прототип функции OemToChar
#include <iostream>
#include <fstream>
#include <string>
#include <clocale>            // здесь прототип функции setlocale
using namespace std:
bool equal( const string& cw, const string& w )
{
char punct[] = { '.', ',', '?', '!' };
if ( cw == w ) return true;
 for ( int i = 0; i < sizeof( punct ); ++i )
 if ( cw == w + punct[i] )
return true;
return false;
}
int main() {
string word, curword;
setlocale( LC_ALL, "Russian" );  // Для работы с русскими символами
cout << " Введите слово для поиска: "; cin >> word;
char pattern[30];
OemToChar( word.c_str(), pattern ); // Для работы с русскими символами
ifstream fin("infile.txt");
if ( !fin ) { cout << "Ошибка открытия файла." << endl;
cin.get();
return 1;
}
 int count = 0;
 while ( !fin.eof() )
{
 fin >> curword;
if ( equal( curword, string( pattern ) ) ) count++;
}
cout << "Количество вхождений слова: " << count << endl;
cin.get();
cin.get();
}
```

Обратите внимание на реализацию функции equal и, в частности, на использование операции сложения для добавления в конец строки w одного из знаков пунктуации.

Задача 2. Вывод вопросительных предложений

Написать программу, которая считывает текст из файла и выводит на экран только вопросительные предложения из этого текста.

Итак, имеется текстовый файл неизвестного размера, состоящий из неизвестного количества предложений. Предложение может занимать несколько строк, поэтому читать файл построчно неудобно. При решении аналогичной задачи в первой части практикума было принято решение выделить буфер, в который поместится *весь* файл. Такое решение тоже нельзя признать идеальным — ведь файл может иметь сколь угодно большие размеры, и тогда программа окажется неработоспособной. Поищем более удачное решение, используя новые средства языка C++, с которыми мы познакомились на этом семинаре. Попробуем читать файл пословно, как и в предыдущей программе, в переменную word типа string, и отправлять каждое прочитанное слово в строковый поток sentence типа ostringstream, который, как вы уже догадались, будет хранилищем очередного предложения.

При таком подходе, однако, есть проблема, связанная с потерей разделителей при чтении файла операцией fin >> word. Чтобы ее решить, будем «заглядывать» в следующую позицию файлового потока fin с помощью метода peek. При обнаружении

символа-разделителя его нужно отправить в поток sentence и переместиться на следующую позицию в потоке fin, используя метод seekg. Подробности обнаружения символа-разделителя инкапсулируем в глобальную функцию isLimit. Осталось решить подзадачи:

- обнаружить конец предложения, то есть один из символов '.', '!', '?';
- если это вопросительное предложение, вывести его в поток cout, в

противном случае очистить поток sentence для накопления следующего предложения.

Рассмотренный алгоритм реализуется в листинге 5.

Листинг 5. Вывод вопросительных предложений

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
using namespace std;
bool isLimit( char c )
{
char lim[] = { ' ', '\t', '\n' };
for ( int i = 0; i < sizeof( lim ); ++i )
if ( c == lim[i] )  return true;
return false:
}
int main()
{
ifstream fin( "infile.txt" ):
if ( fin )
{
```

```
 cout << "Ошибка открытия файла." << endl;
cin.get();
return 1;
}
int count = 0;
 string word;
 ostringstream sentence;
 while( !fin.eof() )
{
 char symb:
while( isLimit( symb = fin.peek() ) )
{
 sentence << symb;
 if ( symb == '\n' ) break;
 fin.seekg( 1, ios::cur );
}
fin >> word: sentence << word:
char last = word[word.size() - 1];
if ( ( last == '.' ) || ( last == '!' ) ) {
sentence.str("");                                    // очистка потока
                                                continue:
}
if ( last == '?' )
 {
cout << sentence.str();
sentence.str( "" ); \
count++;
}
}
if ( !count ) cout << "Вопросительных предложений нет." << endl;
cin.get():
}
```

Протестируйте приведенные программы. Не забудьте поместить в один каталог с программой текстовый файл infile.txt. Если программа запускается из среды Visual Studio, файл должен находиться в каталоге с исходными текстами проекта, если же из папки Debug (например, в режиме командной строки), то в этой папке.

Итоги

1. Для поддержки файлового ввода и вывода стандартная библиотека C++ содержит классы ifstream, ofstream, fstream.
2. Работа с файлом предполагает следующие операции: создание потока, открытие потока и связывание его с файлом, обмен с потоком (ввод-вывод), закрытие файла. Рекомендуется всегда проверять, чем завершилось открытие файла.
3. Если в процессе ввода-вывода фиксируется ошибочная ситуация, потоковый объект принимает значение, равное нулю.

4. Следите за состоянием выходного потока после каждой операции вывода, так как на диске может не оказаться свободного места.

5. Работу со строковыми потоками обеспечивают классы i stri ngstream , ostringstream, stringstream. Использование строковых потоков аналогично применению файловых потоков. Различие в том, что физически информация потока размещается в оперативной памяти, а не в файле на диске.

6. Класс string стандартной библиотеки C++ предоставляет программисту очень удобные средства работы со строками. Класс берет на себя управление памятью, как при размещении строки, так и при всех ее модификациях.

Основные концепции стандартной библиотеки

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

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

Обобщенные алгоритмы реализуют большое количество процедур, применимых к контейнерам — например, поиск, сортировку, слияние и т. п., но они не являются методами контейнерных классов. Наоборот, алгоритмы представлены в STL в форме глобальных шаблонных функций. Благодаря этому достигается их универсальность: эти функции можно применять не только к объектам контейнерных классов, но и к массивам. Независимость от типов контейнеров достигается за счет косвенной связи функции с контейнером: в функцию передается не сам контейнер, а пара адресов, first, last, задающая диапазон обрабатываемых элементов. Реализация указанного механизма взаимодействия базируется на использовании так называемых *итераторов. Итераторы* — это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать для последовательного продвижения по контейнеру, как обычные указатели, а также разыменовывать для получения или изменения значения элемента.

Контейнеры

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

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

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

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

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

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

Итераторы

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

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

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

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

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

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

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

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

К *основным операциям,* выполняемым с любыми итераторами, относятся:

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

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

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

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

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

используется следующая:

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

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

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

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

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

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

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

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

Значения прямых, двунаправленных и итераторов произвольного доступа могут быть сохранены, а вот значения входных и выходных итераторов сохраняться не могут (аналогично тому, как не может быть гарантирован ввод тех же самых значений при вторичном обращении к потоку cin). Как следствие, любые алгоритмы, основанные на входных или выходных итераторах, должны быть однопроходными.

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

template <class Inputlterator, class T>

Inputlterator find(Inputlterator first, Inputlterator last, const T& value):

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

Общие свойства контейнеров

В табл. 5 приведены имена типов, определенные с помощью typedef в большинстве контейнерных классов, а в табл. 6 — некоторые общие для всех контейнеров операции.

Таблица 5. Унифицированные типы, определенные в STL

Поле	Пояснение
value_type	Тип элемента контейнера

size_type	Тип индексов, счетчиков элементов и т. д.
iterator	Итератор
const iterator	Константный итератор (значения элементов изменять запрещено)
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)

Таблица 6. 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
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()	Возвращают итератор на начало контейнера (итерации будут производиться в прямом направлении)
iterator end()	Возвращают итератор на конец контейнера (итерации в прямом направлении будут закончены)
reverse_iterator begin()	Возвращают реверсивный итератор на конец контейнера (итерации будут производиться в обратном направлении)
reverse iterator end()	Возвращают реверсивный итератор на начало контейнера (итерации в обратном направлении будут закончены)

Алгоритмы

Алгоритм — это функция, которая производит некоторые действия над элементами контейнера (контейнеров). Чтобы использовать обобщенные алгоритмы, нужно подключить к программе заголовочный файл <algorithm>. В табл. 7 приведены наиболее популярные алгоритмы STL.

Таблица 7. Некоторые типичные алгоритмы STL

Алгоритм	Назначение
accumulate	Вычисление суммы элементов в заданном диапазоне

copy	Копирование последовательности, начиная с первого элемента
count	Подсчет количества вхождений значения в последовательность
count_if	Подсчет количества выполнений условия в последовательности
equal	Подсчет количества выполнений условия в последовательности
fill	Замена всех элементов заданным значением
find	Нахождение первого вхождения значения в последовательность
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 последователь-ности
merge	Слияние отсортированных последовательностей
remove	Перемещение элементов с заданным значением
replace	Замена элементов с заданным значением
search	Нахождение первого вхождения в первую последовательность второй последовательности
sort	Сортировка
swap	Обмен местами двух элементов
transform	Выполнение заданной операции над каждым элементом последовательности

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

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

Примеры использования некоторых алгоритмов будут даны ниже.

Использование последовательных контейнеров

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

Контейнер *вектор* является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации [] или метода at. Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего. Контейнер *список* организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список не поддерживает произвольного доступа к своим элементам: например, для выборки n-го элемента нужно последовательно выбрать предыдущие n-1 элементов. Контейнер *двусторонняя очередь (дек)* во многом аналогичен вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора дек эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего). Существует пять способов определить объект для последовательного контейнера.

1. Создать пустой контейнер:

```
vector<int> vec1; list<string>
list1;
```

2. Создать контейнер заданного размера и инициализировать его элементы значениями по умолчанию:

```
vector<string> vec1( 100 ); list<double> list1( 20 );
```

3. Создать контейнер заданного размера и инициализировать его элементы указанным значением:

```
vector<string> vec1( 100, "Hello!" ); deque<int>
dec1( 300, -1 );
```

4. Создать контейнер и инициализировать его элементы значениями диапазона [first, last) элементов другого контейнера:

```
int arr[7] = { 15, 2, 19, -3, 28, 6, 8 }; vector<int> v1( arr,
arr + 7 ); list<int> lst(v1.beg( ) + 2, v1.end() );
```

5. Создать контейнер и инициализировать его элементы значениями элементов другого *однотипного* контейнера:

```
vector<int> v1;

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

Для вставки и удаления последнего элемента контейнера любого из трех рассматриваемых классов предназначены методы push_back и pop_back. Кроме того, список и очередь (но не вектор) поддерживают операции вставки и удаления первого элемента контейнера push_front и pop_front. Учтите, что методы pop_back и pop_front *не возвращают* удаленное значение. Чтобы считать первый элемент, используется метод front, а для считывания последнего элемента — метод back. Кроме этого, все типы контейнеров имеют более общие операции вставки и удаления, перечисленные в табл. 8.

Таблица 8. Методы вставки в контейнер и удаления из контейнера

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

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

Задания для лабораторной работы № 7 по теме «Библиотека STL. Класс String и файловые потоки»

Вариант 1

Написать программу, которая считывает из текстового файла три предложения и выводит их в обратном порядке.

Вариант 2

Написать программу, которая считывает текст из файла и выводит на экран только предложения, содержащие заданное с клавиатуры слово.

Вариант 3

Написать программу, которая считывает текст из файла и выводит на экран только строки, содержащие двузначные числа.

Вариант 4

Написать программу, которая считывает английский текст из файла и выводит на экран слова, начинающиеся с гласных букв.

Вариант 5

Написать программу, которая считывает текст из файла и выводит его на экран, меняя местами каждые два соседних слова.

Вариант 6

Написать программу, которая считывает текст из файла и выводит на экран только предложения, не содержащие запятых.

Вариант 7

Написать программу, которая считывает текст из файла и определяет, сколько в нем слов, состоящих не более чем из четырех букв.

Вариант 8

Написать программу, которая считывает текст из файла и выводит на экран только цитаты, то есть предложения, заключенные в кавычки.

Вариант 9

Написать программу, которая считывает текст из файла PI выводит на экран только предложения, состоящие из заданного количества слов.

Вариант 10

Написать программу, которая считывает английский текст из файла и выводит на экран слова текста, начинающиеся с гласных букв и оканчивающиеся гласными буквами.

Вариант 11

Написать программу, которая считывает текст из файла и выводит на экран только строки, не содержащие двузначные числа.

Вариант 12

Написать программу, которая считывает текст из файла и выводит на экран только предложения, начинающиеся с тире, перед которым могут следовать только пробельные символы.

Вариант 13

Написать программу, которая считывает английский текст из файла и выводит его на экран, заменив каждую первую букву слов, начинающихся с гласной буквы, на прописную.

Вариант 14

Написать программу, которая считывает текст из файла и выводит его на экран, заменив цифры от 0 до 9 на слова «ноль», «один», ..., «девять», начиная каждое предложение с новой строки.

Вариант 15

Написать программу, которая считывает текст из файла, находит самое длинное слово и определяет, сколько раз оно встретилось в тексте.