ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
(РОСАВИАЦИЯ)

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

Учебно-методическое пособие
по выполнению лабораторных работ № 12, 13, 14

для студентов I курса
направления 09.03.01
очной формы обучения

Москва
ИД Академии Жуковского
2021

УДК 004.42
ББК 6Ф7.3
Н17

Рецензент:
Черкасова Н.И. – канд. физ.-мат. наук, доцент

Надейкина Л.А.

Н17 Программирование [Текст] : учебно-методическое пособие по выполнению лабораторных работ № 12, 13, 14 / Л.А. Надейкина. – М.: ИД Академии Жуковского, 2021. – 48 с.

Данное учебное пособие издается в соответствии с учебным планом для студентов I курса направления подготовки 09.03.01 «Информатика и вычислительная техника» (бакалавриат) очной формы обучения.

Рассмотрено и одобрено на заседаниях кафедры 26.10.2021 г. и методического совета 26.10.2021 г.

УДК 004.42
ББК 6Ф7.3

В авторской редакции

Подписано в печать 23.11.2021 г.
Формат 60x84/16 Печ. л. 3 Усл. печ. л. 2,79
Заказ № 878/1004-УМП36 Тираж 30 экз.

Московский государственный технический университет ГА
125993, Москва, Кронштадтский бульвар, д. 20

Издательский дом Академии имени Н. Е. Жуковского
125167, Москва, 8-го Марта 4-я ул., д. 6А
Тел.: (495) 973-45-68
E-mail: zakaz@itsbook.ru

1. ЛАБОРАТОРНАЯ РАБОТА № 12

Обработка данных текстовых и бинарных файлов.

1.1. Цель лабораторной работы

Целью лабораторной работы является получение навыков программирования с использованием текстовых и бинарных файлов, содержащих структурированные данные.

1.2. Теоретические сведения

Работа с файлами

Информация во внешней памяти сохраняется в виде файлов – именованных участков внешней памяти.

Файлы позволяют сохранять информацию при отключении компьютера, в отличие от оперативной памяти (ОП). Под файлом понимают также логическое устройство – потенциальный источник или приемник информации.

В файлы можно как помещать данные, так и извлекать их из файлов.

Такие действия имеют общее название ***ввод/вывод***.

Функции C++ позволяют читать данные из файлов в ОП, получать их с устройств (напр. с клавиатуры), и записывать данные из ОП в файл или выводить их на различные устройства, например, на экран или на принтер.

Файл имеет следующие характерные особенности:

- имеет имя на диске, что дает возможность программам идентифицировать и работать с файлами;
- длина файла ограничивается только емкостью диска.

Часто бывает необходимо ввести некоторые данные из файла или вывести результаты в файл. Например, необходимо обрабатывать огромные массивы данных, которые слишком велики, чтобы полностью разместиться в оперативной памяти.

Особенностью языка C++ является отсутствие в нем структурированных файлов. Все файлы рассматриваются как **неструктурированная последовательность байтов.** При таком подходе понятие файла распространяется и на различные устройства.

В C++ существуют специальные средства ввод/вывода данных. Все операции ввода/вывода реализуются с помощью функций, которые находятся в библиотеке C++.

Средства ввода/вывода языка C++ можно разделить на три группы:

1) ввод/вывод верхнего уровня – потоковый
   а) библиотека функций Си – интерфейс в файле <cstdio>
   б) библиотека классов C++ – интерфейс в файле <fstream>
2) ввод/вывод нижнего уровня (системный ввод - вывод) – файл <io.h>
3) ввод/вывод для консоли и портов – интерфейс в файле <conio.h>

Поток – это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику.

Функции библиотеки ввода/вывода языка C++, поддерживающие обмен данными с файлами на уровне потока, позволяют обрабатывать данные различных размеров и форматов, обеспечивая при этом буферизованный ввод/вывод. Таким образом, поток представляет сам файл вместе с предоставленными средствами буферизации.

Чтение данных из потока называется извлечением, вывод в поток – помещением (включением) данных в поток.

Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер). Обмен с потоком для увеличения скорости передачи данных производится, как правило, через специальную область оперативной памяти – буфер. Буфер накапливает байты, и фактическая передача данных выполняется после заполнения буфера (рис. 1). При вводе это дает возможность исправить ошибки, если данные из буфера еще не отправлены в программу.

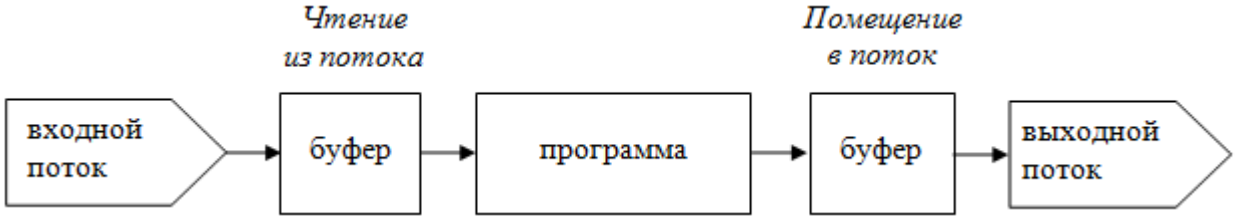

Рисунок 1. Буферизация данных при работе с потоками

При работе с потоком можно:

- открывать и закрывать потоки (связывать указатели на поток с конкретными файлами и разрывать эту связь);
- вводить и выводить строку, символ, форматированные данные, порцию данных произвольной длины;
- анализировать ошибки ввода/вывода и достижения конца файла;
- управлять буферизацией потока и размером буфера;
- получать и устанавливать указатель текущей позиции в файле.

Библиотека ввода/вывода C++ включает средства для работы с последовательными файлами, представляющими собой именованную последовательность байтов, имеющую начало и конец. Чтение из файла или запись в файл ведутся байт за байтом, позиции в файле, откуда производится чтение, или куда ведется запись, определяются указателем позиции файла.

Указатель позиции записи или чтения устанавливается либо автоматически, либо с помощью функций управления положением, указатель можно установить на нужный байт.

Файлы – именованные объекты внешней памяти, доступ к файлам поддерживается операционной системой.

Все что надо – задать способ связи программы с файлом, а также иметь функции, используемые программой при чтении содержимого файла, записи в

файл, создания нового файла, позиционирования записи и чтения данных в файл и из файла. Такие действия являются частью аспекта ввода/вывода данных и для их реализации в С++ имеются различные средства**.**

Все средства имеют альтернативные варианты методов выполнения основных работ с файлами, здесь будут рассмотрены некоторые методы библиотеки потоковых классов.

Текстовые и бинарные (двоичные) файлы

Когда данные сохраняются в файле, их можно сохранять в текстовом или бинарном (двоичном) формате.

Текстовая форма означает, что все данные даже числа сохраняются как текст – последовательность символов.

Двоичный формат означает, что данные в файле сохраняются во внутреннем представлении этих данных в компьютере.

Для символа двоичное представление совпадает с его текстовым представлением - двоичным кодом символа.

Однако для чисел двоичное представление очень сильно отличается от их текстового представления.

Рассмотрим пример - форму сохранения значение переменной **float a=0.375** в текстовом и бинарном файле.

При записи в текстовой файл операция вставки <<производит преобразование внутренних кодов переменной в коды символов числа, то есть в файле сохраняются коды следующих символов:

‘0’	‘.’	‘3’	‘7’	‘5’
00110000	00101110	00110011	00110111	00110101

---------- всего 40 бит ----------

Двоичное представление значения переменной в бинарном файле идентичное внутреннему представлению компьютером данной переменной в формате с плавающей точкой, с нормализованной мантиссой и порядком:

Биты двоичной нормализованной мантиссы (правильной дроби)	биты показателя экспоненты	бит знака числа
00000000000000000000011	0111110	0

----------всего 32 бита для типа **float**----------

Каждый формат имеет свои достоинства.

Текстовой формат прост для чтения и редактирования файла. Текстовой файл легко переносится с одной компьютерной системы на другую.

В двоичном файле числа сохраняются более точно, поскольку он позволяет сохранить точное внутреннее представление числа, не происходит ошибок преобразования или округления чисел. Сохранение данных в бинарном файле происходит быстрее, поскольку при этом опять же не происходит преобразования и данные можно сохранять крупными блоками. Однако при

переносе данных в другую систему возможны проблемы, если в новой системе применяется другое внутреннее представление данных.

Текстовой файл — это последовательность символьных строк переменной длины, разделенных комбинацией символов ***CR*** - “перевод каретки в начало” (символ с кодом 13) и символ ***LF*** - “перевод строки” (символ с кодом 10). Как правило, работа с текстовым файлом организуется построчно.

При записи данных из оперативной памяти в файл, значения числовых типов будут преобразовываться из внутренних кодов хранения данных в символьное представление и в таком виде записываться в строку файла.

При чтении данных из файла в оперативную память часть строки будет пониматься как символьное представление числовой переменной и при вводе данных в оперативную память выполняется преобразование символов в двоичные коды внутреннего представления данных.

Кроме того, в текстовом режиме при чтении из файла два символа ***CR*** и ***LF*** преобразуются в один символ новой строки ***‘\n’***. При записи в текстовой файл один символ – символ новой строки ***‘\n’*** преобразуется в два символа ***CR*** и ***LF***.

Бинарный файл предназначен для двоичного режима обмена данными, когда преобразование символов не происходит, и их значения не анализируются. Бинарный файл – это линейная последовательность байтов, соответствующая внутреннему представлению данных.

Потоковый ввод - вывод на базе библиотеки классов

Рассмотрим механизмы выполнения основных работ с файлами на базе библиотеки классов входных/выходных потоков.

Основные виды работ с файлами

1) создание потока;
2) открытие файла;
3) присоединение файла к потоку;
4) обмены с файлом с помощью потока;
5) отсоединение потока от файла;
6) закрытие файла.

Создание потоков

Потоки для работы с файлами – объекты следующих классов:

ofstream – класс выходных файловых потоков, для записи данных в файл,

ifstream – класс входных файловых потоков для чтения данных из файла,

fstream – класс двунаправленных файловых потоков для чтения и записи данных в файл.

Описание этих классов находится в файле ***<fstream>.***

Определение потоков - объектов классов.

Первый способ – создание объекта с помощью конструктора по - умолчанию.

```
ofstream fout;   //выходной файловый поток
ifstream fin;    //входной файловый поток
```

fstream fio; *//входной выходной (двунаправленный) файловый поток*

При таком определении потоковых объектов, выделяется память на буфер обмена и инициируются переменные, характеризующие состояние потока.

Так как классы файловых потоков являются производными классами от классов стандартных входных и выходных потоков и от базового класса ***ios***, то они наследуют все переменные и флаги состояния потока, а также компонентные функции (методы) для установки флагов, для выполнения форматированного и не форматированного обмена данными.

Создав файловый поток, нужно присоединить его к конкретному файлу с помощью компонентной функции ***open ()*** класса ***fstream***:

void open (const char* filename, ios_base::mode = ios_base::in | ios_base::out);

Первый параметр – полное имя файла на диске.

Второй – дизъюнкция флагов, определяющих режим работы с файлом:

ios::in – открыть файл в режиме чтения данных, этот режим является режимом по - умолчанию для потоков ***ifstream***;

ios::out – открыть файл в режиме записи данных, этот режим является режимом по умолчанию для потоков ***ofstream***;

ios::app – открыть файл в режиме записи данных в конец файла;

ios::ate – передвинуться в конец уже открытого файла;

ios::trunc – очистить файл, это же происходит в режиме ***ios::out***;

ios::nocreate – не выполнять операцию открытия файла, если он не существует;

ios::noreplace – не открывать существующий файл;

ios::binary – открыть двоичный файл.

Функция открывает файл (если он существует) или создает новый файл и связывает его с потоком в нужном режиме.

Вызов компонентной функции осуществляется с помощью уточненного имени:

имя потока. open (имя файла, режим);

Примеры:

fout.open (" D:\\ DATA\\work.dat");

/* по – умолчанию устанавливается режим ***ios::out***, если файл не существует, он будет создан и присоединен к потоку ***fout***.

Теперь, применяя к потоку операцию включения ***fout <<…,*** или вызывая компонентные функции ***fout.put ('…')*** или ***fout.write (…)***, будет производиться запись в файл*/

fin.open ("result.txt");

/* по - умолчанию устанавливается режим ***ios::in***, если файл не существует, то вызов функции приведет к ошибке. Существующий файл присоединяется к потоку ***fin***.

Применяя операцию извлечения ***fin>>…,*** или вызывая компонентные функции ***fin.get (…)***, ***fin.getline (…)***, ***fin.read (…)*** будет производится чтение данных из файла*/

fio.open ("change.dat", ios::out)

/* файл открыт с двунаправленным потоком для записи и будет иметь такую направленность до закрытия, потом его можно вновь открыть для чтения данных или в двунаправленном режиме*/

Если открытие файла завершилось неудачей, объект, соответствующий потоку (пусть его имя – ***ofs***), будет возвращать значение 0:

```
if (!ofs) {
  cout << "Файл не открыт\n";
  }
```

Проверить успешность открытия файла можно также с помощью функции ***is_open ()***, являющейся публичным методом класса ***fstream*** и имеющей следующий прототип:

bool is_open ();

Например,

```
if (!ofs.is_open ())
  cout << "Файл не открыт\n";
```

Закрытие файла

Отрытый метод класса ***void fstream::close ()*** отсоединяет файл от потока и вызывается с помощью уточненного имени:

имя файлового потока (присоединенного к файлу). close ();

Определение потоков - объектов классов с присоединением потока к физическому файлу

Второй способ – создание объекта с помощью вызова конструктора с параметрами. Первый параметр конструктора – имя физического файла, второй - дизъюнкция флагов, определяющих режим работы с файлом.

Примеры*:*

```
ifstream  input (" filename.txt", ios::in);
ofstream output ("filename.out", ios::out);
fstream ioput (" …", ios:: out | ios::in);
```

После можно писать в файл и читать из файла:

```
input.read (buffer, number_of_buffer)
output.write (buffer, number_of_buffer)
```

Эти функции возвращают ссылку на поток, поэтому их можно выполнять цепочкой.

Удобно использовать компонентную функцию потоковых классов

bool fstream:: eof ()

возвращает истину, если файл пуст, а если файл не пуст, то ложь. На самом деле, ***eof ()*** — это функция, входящая в класс ***ios*** (потоковый ввод/вывод), классы файловых потоков наследуют ее. Обращение к ней:

имя файлового потока. eof ();

Возвращает ненулевое значение, если имеет место условие конца файла.

Пример:

```
//Работа с файлом в потоковом режиме.
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  int n = 50;
  char str [20];
  // Открываем файл для вывода (записи)
  ofstream ofs ("Test.txt");
  // Проверка корректности открытия файла
  if (!ofs) {
      cout << "Файл Test.txt не может быть открыт для записи.\n";
       system ("pause"); return;}
  else {
     ofs << "Hello! \n" << n;
     ofs.close (); // Закрываем файл
      }
  // Открываем файл для ввода (чтения)
  ifstream file ("Test.txt");
  // Проверка корректности открытия файла
  if (!file) {
     cout << "Файл Test.txt не может быть открыт для чтения.\n";
     system ("pause"); return;}
  else {
    file >> str >> n;
    cout << str << "\n" << n;
  // Закрываем файл
    file.close ();
       }
  system("pause");
  return 0;
}
```

1.3. Задание на выполнение лабораторной работы

1) Разработать структуру программы, схемы алгоритмов и программу обработки данных бинарного файла. Файл должен содержать структурированные данные конкретного варианта лабораторной работы. Использовать данные варианта лабораторной работы № 7.

2) Программа должна включать следующие функции обработки данных:

- создание бинарного файла из текстового файла с данными;
- дополнение файла новыми записями;
- чтение данных бинарного файла;
- поиск структур бинарного файла, поиск производить:

1) по одному поисковому признаку;
2) по любому сочетанию заданных поисковых признаков;

- модификация ряда структур бинарного файла;

3) Главная функция должна производить вызов разработанных функций.

1.4. Порядок выполнения работы

1) Сформировать файлы с данными для тестирования программы:

- sozd.txt - файл с исходными данными
- poisk1.txt – файл для тестирования функции поиска по одному поисковому признаку;
- poisk2.txt - файл для тестирования функции поиска по сочетанию двух поисковому признаков;
- dop.txt – файл, аналогичный по форме файлу sozd.txt с данными для дополнения бинарного файла;
- kor.txt – файл с данными для модификации записей бинарного файла (в каждой строке файла два данных – первое для поиска структуры в бинарном файле и второе – новое значение другого элемента найденной структуры).

2) Написать программу, в которой определить в соответствии с заданием функции обработки данных.

3) В главной функции произвести вызовы функций. Вывод результирующей текстовой информации следует производить в текстовой файл результатов.

1.5. Пример выполнения лабораторной работы

Даны сведения о студентах: номер зачетной книжки, наименование группы, фамилия и инициалы, размер стипендии. Исходные данные для создания бинарного файла поместить в файл данных в виде, представленном на рис. 4.1 (файл sozd.txt).

Данные для дополнения бинарного файла разместить в текстовом файле в виде, показанном на рис. 4.2 (файл dop.txt).

Данные для тестирования функций поиска по фамилии студента, поиска по сочетанию двух признаков и функции коррекции записей в бинарном файле показаны на рис. 4.3, 4.4, 4.5 (файлы poisk1.txt, poisk2.txt, kor.txt).

Файл результатов представлены на рис. 4.6.

```
11007   ЭВМ 2-1   Иванов А.А.       280.5
11000   ЭВМ 2-1   Петров П.А.       167.9
11001   ЭВМ 2-2   Митин П.Ю.        0
11002   ЭВМ 2-2   Качнов А.И.       250.5
11004   ЭВМ 2-3   Васильева И.Б.    180.7
11005   ЭВМ 2-3   Горнец Н.Н.       167.9
```

Рисунок 4.1 Файл данных для создания sozd.txt

```
11012   ЭВМ 2-1   Козинов А.И.      240.0
11014   ЭВМ 2-2   Арапова Н.А.      0
11015   ЭВМ 2-3   Васина Р.П.       167.9
```

Рисунок 4.2. Файл данных для дополнения dop.txt

```
Митин П.Ю.
Качнов А.И.
Сидоров А.О.
```

Рисунок 4.3. Файл данных poisk1.txt

```
250.5   ЭВМ 2-2
0       ЭВМ 2-1
200
120     ЭВМ 1-1
150     ЭВМ 2-3
```

Рисунок 4.4. Файл данных poisk2.txt

```
Васильева И.Б.  250.5
                0
Горнец Н.Н.     0
```

Рисунок 4.5. Файл данных kor.txt

Листинг программы

```
//Лабораторная работа №12 студента группы ЭВМ 1-1 Иванова И.Н.
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <cstdlib>
using namespace std;
//объявляется структурный тип stud и структура st
struct stud { long nz;
              char gr [8], fio [16];
              float rs;} st;
int z = sizeof (stud);
char sh [][150] = {//строка шапки таблицы
  "                СВЕДЕНИЯ О СТУДЕНТАХ                ",
  "┌──────────┬──────────┬──────────┬──────────┐",
  "│НОМЕР      │ГРУППА    │ФАМИЛИЯ   │РАЗМЕР    │",
  "│ЗАЧЕТКИ    │          │ИНИЦИАЛЫ  │СТИПЕНДИИ │",
  "├──────────┼──────────┼──────────┼──────────┤",
  "│          │          │          │          │",
  "└──────────┴──────────┴──────────┴──────────┘"};
ifstream fin;           // определение входного файлового потока
ofstream fout;          //выходного файлового потока
fstream io;             //двунаправленного
```

//прототипы вызываемых функций:

```
void p (), cht (), sozd (), dop (), zf (), poisk1(), poisk2 (), kor (), filtr (char*), psh ()
```

//главная функция

```
int main () {
fout.open ("lr12.txt");              // открытие файла результатов
if (!fout) {cout<<"Ошибка при открытии файла результатов";
system ("pause");
return;
}
//вызовы функций
sozd (); cht (); dop (); cht (); poisk1(); poisk2 (); kor (); cht ();
fout.close (); //закрытие файла результатов
}
```

// вывод строк шапки таблицы в файл результатов

```
void psh ()   {
for (int i = 0; i < 5; i++)
fout << sh [i] << endl;
}
```

// вывод одной строки таблицы в файл результатов

```
void p () {
fout << "||" << setw (13) << st.nz << "||"<< setw (10) << st.gr << "||"
<< setw (15) << st.fio << "||"<< setw (14) << st.rs << "||\n";
}
```

//чтение бинарного файла, вывод структур в файл результатов в таблицу:

```
void cht () {
int n = 0;              //счетчик структур в бинарном файле
float s= 0;             // суммарная стипендия
fout << "\n           ЧТЕНИЕ ФАЙЛА\n";
fin.open ("binary.cpp", ios::in|ios::binary); //открытие бинарного файла для чтения
if (!fin) {cout<<"Ошибка открытия бинарного файла для чтения";
system ("pause");
return;
}
psh (); // вывод строк шапки таблицы
while (fin.peek ()!= EOF) {      //пока не достигли символа конца файла
fin.read ((char*) &st, z);          //считываем структуру в переменную st
p ();                    //выводим поля структуры в таблицу
s += st.rs; n++;
}
fout << sh [5] << endl << "Суммарная стипендия = " << s << endl
 << "Средняя стипендия = " << s/n << endl
 << "Количество структур в файле = " << n << endl;
```

```
fin.close ();
}
```
//функция создания бинарного файла
```
void sozd () {
fout << "\n            СОЗДАНИЕ ФАЙЛА ";
fin.open ("sozd.txt");                  //открываем файл с исходными данными
if (!fin) {cout<<"Ошибка при открытии файла данных";
system ("pause");
return;
}
//создаем новый бинарный файл для записи:
io.open ("binary.cpp", ios :: out | ios :: binary);
if (!io) {cout<<"Ошибка открытия нового бинарного файла";
system ("pause");
return;
}
psh (); //вывод строк шапки таблицы в файл результатов
zf (); //вызов функции чтения данных и записи их в бинарный файл:
fin.close (); io.close ();
}
```
//чтение данных из файла данных и запись данных в бинарный файл
```
void zf () {
char T [80];
while (!fin.eof ()) {
fin>>st.nz; fin.get (st.gr,11); filtr(st.gr); fin.get (st.fio,17); filtr(st.fio);
fin >> st.rs; fin.getline(T,80);
io.write ((char*) &st, z); // запись структуры целиком из ОП  в бин. файл
p ();   //запись полей структуры в строку таблицы файла результатов
}
fout << sh [5] <<endl;
}
```
//функция удаления пробелов в начале и конце строки
```
void filtr (char* Stroka) {
  char Source [255] = ""; //Вспомогательная строка
  int x, x1 = 0, x2 = strlen (Stroka) - 1; // Установка на начало и конец строки;
  while (Stroka [x1] == '  ') x1++;
  while (Stroka [x2] == '  ' || Stroka[x2] == '\n') x2--;
  for (x = x1; x <= x2; x++)
     Source [x - x1] = Stroka [x];
  Source [x - x1] = '\0'; // В конце строки устанавливаем байтовский нуль
  strcpy (Stroka, Source);
}
```

```
//функция поиска данных по фамилии
void poisk1() {
char name [20];     //символьный массив для хранения поискового признака
fin.open ("poisk1.txt");              //открытие файла данных для поиска
if (!fin) {
cout<<"Ошибка открытия файла данных для поиска 1";
system ("pause");
return;
}
fout<<"\n\n           ПОИСК ПО ФАМИЛИИ СТУДЕНТА\n";
io.open ("binary.cpp", ios :: in | ios :: binary); //открытие бин. ф. для чтения
if (!io) {
cout<<"Ошибка открытия бинарного файла для чтения ";
system ("pause");
return;
}
while (fin.peek ()!=EOF){
fin.getline (name,20); filtr(name);
fout << "\nИщем студента с фамилией -" << "\" "<< name <<" \"" << endl;
if (!strcmp (name,"")) {
fout << "Нет фамилии для поиска\n"; continue;
}
io.seekg (0, ios :: beg);    //указатель бин. файла устанавливается на начало
while (io.peek ()!= EOF)  { //пока не достигнут символ конца файла
io.read ((char*) &st, z);   //читаем из файла одну структуру в st
if (strcmp (st.fio, name) = = 0) //сравниваем поле fio структуры st с name
  {p (); goto m;}   //если совпадение выводим данные структуры
 }                          //и управление передается на метку m
fout<< "Студент с фамилией\" "<< name<<" \" не найден"<<endl;
m:;
}
fin.close (); io.close ();
}
//функция поиска по сочетанию признаков
void poisk2() {
char grup [15]; float stip; //переменные для хранения поисковых признаков-
                                  //это наименование группы и размер стипендии
fin.open ("poisk2.txt", ios :: in);         // открытие файла данных для поиска
if(!fin) {cout<<"Ошибка открытия файла данных для поиска 2";
system ("pause");
return;
}
```

```
fout << "\n\n ПОИСК ПО НАИМЕНОВАНИЮ ГРУППЫ СТУДЕНТА И"
"ПО РАЗМЕРУ СТИПЕНДИИ\n";
while (fin.peek () != EOF) {      //пока не достигнут символ конца файла
fin >> stip; fin.getline (grup, 16); filtr(grup); //считываются поиск. признаки
int k = 0;
fout << "\nИщем в группе -" << "\"" << grup << "\""
 <<" студентов со стипендией > = " << stip << endl;
if ((!strcmp (grup, "")) && (stip = = 0.0)) {
fout << "Нет для поиска ни группы, ни стипендии \n"; continue;}
io.open ("binary.cpp", ios :: in | ios :: binary); //открытие бин. ф. для чтения
if (!io) {cout<<"Ошибка открытия бинарного файла для чтения";
system ("pause");
return;
}
while (io.read ((char*) &st, z)) //пока читаются данные из бинарного файла
if ((!strcmp (st.gr, grup) || !strcmp(grup, "")) && (stip <= st.rs || stip == 0))
{p (); k++;}            //если условие истинно, поля выводятся
                        //и счетчик выводимых структур увеличивается на 1
if (!k) fout << "Таких студентов нет" << endl;
io.close ();
}//конец тела цикла по файлу данных с поисковыми признаками
fin.close (); io.close ();
}
```

//функция дополнения бинарного файла

```
void dop ()  {
fout<< "\n\n ДОПОЛНЕНИЕ ФАЙЛА НОВЫМИ ЗАПИСЯМИ" << endl;
fin.open ("dop.txt", ios :: in); // открытие файла данных для дополнения
if(!fin) {cout<<"Ошибка открытия файла данных для дополнения";
system ("pause");
return;
}
//открытие бинарного файла в режиме дополнения
io.open ("binary.cpp", ios :: app | ios :: binary);
if (!io) {cout<<"Ошибка открытия бин. файла для дополнения";
system ("pause");
return;
}
for (int i = 1; i < 5; i++)
fout<<sh[i]<<endl;
zf ();
fin.close (); io.close ();
}
```

//функция коррекции структур бинарного файла

```
void kor () {
char t [80];            //массив - буфер
char name [25]; //для хранения фамилии студента – поисковый признак
float stip;                    //переменная для новой стипендии
fout << "\n\n                  КОРРЕКЦИЯ ФАЙЛА\n\n";
fin.open ("kor.txt", ios ::in);
if (!fin) {cout<<"Ошибка открытия файла данных для коррекции";
system ("pause");
return;
}
while (fin.peek () != EOF) {      //1 цикл ввода данных для коррекции
fin.get (name, 17); filtr(name); fin >> stip; fin.getline (t, 80);
int k=0;
if (!strcmp (name, ""))
{fout << "\nНет фамилии для поиска структуры\n"; continue;}
//открытие бинарного файла в режиме чтения и записи
io.open ("binary.cpp", ios :: in | ios :: out | ios :: binary);
if (!io){cout<<"Ошибка открытия бин. файла для его коррекции");
system ("pause");
return;
}
while (io.read ((char*) &st, z)) // 2
if (!strcmp (st.fio, name)) {              //если найдена структура
p (); st.rs = stip; p ();     //изменяем ее
io.seekp(-z, ios :: cur); //смещаем указатель файла на одну структуру назад
io.write ((char*) &st, z); //запись в файл откорректированной структуры
k++; break;
} // while 2
if (!k) fout<<"Студент с ФИО - " << name << " не найден" << endl;
io.close ();
}// while 1
fin.close (); io.close ();
}
```

СОЗДАНИЕ ФАЙЛА
СВЕДЕНИЯ О СТУДЕНТАХ

НОМЕР ЗАЧЕТКИ	ГРУППА	ФАМИЛИЯ ИНИЦИАЛЫ	РАЗМЕР СТИПЕНДИИ
11007	ЭВМ 2-1	Иванов И. И.	280.5
11000	ЭВМ 2-1	Петров Т.О.	167.9
11001	ЭВМ 2-2	Митин П.Ю.	0
11002	ЭВМ 2-2	Качнов А.И.	250.5
11004	ЭВМ 2-3	Горнец Н.Н.	180.7
11005	ЭВМ 2-3	Васильева И.Б.	167.9

ДОПОЛНЕНИЕ ФАЙЛА ЗАПИСЯМИ

НОМЕР ЗАЧЕТКИ	ГРУППА	ФАМИЛИЯ ИНИЦИАЛЫ	РАЗМЕР СТИПЕНДИИ
11012	ЭВМ 2-1	Козинов А.И.	240
11014	ЭВМ 2-2	Арапова Н.А.	0
11015	ЭВМ 2-3	Васина Р.П.	167.9

ЧТЕНИЕ ФАЙЛА
СВЕДЕНИЯ О СТУДЕНТАХ

НОМЕР ЗАЧЕТКИ	ГРУППА	ФАМИЛИЯ ИНИЦИАЛЫ	РАЗМЕР СТИПЕНДИИ
11007	ЭВМ 2-1	Иванов И. И.	280.5
11000	ЭВМ 2-1	Петров Т.О.	167.9
11001	ЭВМ 2-2	Митин П.Ю.	0
11002	ЭВМ 2-2	Качнов А.И.	250.5
11004	ЭВМ 2-3	Горнец Н.Н.	180.7
11005	ЭВМ 2-3	Васильева И.Б.	167.9
11012	ЭВМ 2-1	Козинов А.И.	240
11014	ЭВМ 2-2	Арапова Н.А.	0
11015	ЭВМ 2-3	Васина Р.П.	167.9

Суммарная стипендия = 1.46e+03
Средняя стипендмя = 161.71
Количество структур в файле = 9

ПОИСК ПО ФАМИЛИИ СТУДЕНТА

Ищем данные студента с фамилией -" Митин П.Ю. "
||11001 ||ЭВМ 2-2 ||Митин П.Ю. ||0 ||

Ищем данные студента с фамилией -" "
Нет фамилии для поиска

Ищем данные студента с фамилией -" Качнов А.И. "
||11002 ||ЭВМ 2-2 ||Качнов А.И. ||250.5 ||

Ищем данные студента с фамилией -" Сидоров И.В. "
Студент с фамилией " Сидоров И.В. " не найден

Рисунок 4.6. Файл результатов lr12.txt

```
 ПОИСК ПО НАИМЕНОВАНИЮ ГРУППЫ СТУДЕНТА И ПО РАЗМЕРУ СТИПЕНДИИ

Ищем в группе -"ЭВМ 2-2"студентов со стипендией > = 250.5
11002          ЭВМ 2-2    Качнов А.И.      250.5

Ищем в группе -"ЭВМ 2-1"студентов со стипендией > = 0
11007          ЭВМ 2-1    Иванов И. И.     280.5
11000          ЭВМ 2-1    Петров Т.О.      167.9
11012          ЭВМ 2-1    Козинов А.И.     240

Ищем в группе -""студентов со стипендией > = 200
11007          ЭВМ 2-1    Иванов И. И.     280.5
11002          ЭВМ 2-2    Качнов А.И.      250.5
11012          ЭВМ 2-1    Козинов А.И.     240

Ищем в группе -"ЭВМ 1-1"студентов со стипендией > = 120
Таких студентов нет

Ищем в группе -"ЭВМ 2-3"студентов со стипендией > = 150
11004          ЭВМ 2-3    Горнец Н.Н.      180.7
11005          ЭВМ 2-3    Васильева И.Б.   167.9
11015          ЭВМ 2-3    Васина Р.П.      167.9

                    КОРРЕКЦИЯ ФАЙЛА

11005          ЭВМ 2-3    Васильева И.Б.   167.9
11005          ЭВМ 2-3    Васильева И.Б.   250.5

Нет фамилии для корректировки

11004          ЭВМ 2-3    Горнец Н.Н.      180.7
11004          ЭВМ 2-3    Горнец Н.Н.      0

                    ЧТЕНИЕ ФАЙЛА
                    СВЕДЕНИЯ О СТУДЕНТАХ
```

НОМЕР ЗАЧЕТКИ	ГРУППА	ФАМИЛИЯ ИНИЦИАЛЫ	РАЗМЕР СТИПЕНДИИ
11007	ЭВМ 2-1	Иванов И. И.	280.5
11000	ЭВМ 2-1	Петров Т.О.	167.9
11001	ЭВМ 2-2	Митин П.Ю.	0
11002	ЭВМ 2-2	Качнов А.И.	250.5
11004	ЭВМ 2-3	Горнец Н.Н.	0
11005	ЭВМ 2-3	Васильева И.Б.	250.5
11012	ЭВМ 2-1	Козинов А.И.	240
11014	ЭВМ 2-2	Арапова Н.А.	0
11015	ЭВМ 2-3	Васина Р.П.	167.9

```
Суммарная стипендия = 1.36e+03
Средняя стипендмя = 150.81
Количество структур в файле = 9
```

Рисунок 4.6. Продолжение

1.6. Контрольные вопросы

1) Что такое текстовые и бинарные файлы?
2) Как объявить текстовой и бинарный файл?
3) Какие типы данных можно хранить в бинарном файле?
4) Какая функция используется для связи логического файла программы (файлового потока) с физическим файлом?
5) Средства обмена данными с потоками: операции ввода (>>) и вывода (<<) данных.
6) Средства обмена данными с потоками: функции двоичного ввода/вывода данных.
7) Какие функции используются для обмена данными между ОП и бинарным файлом?
8) Поясните назначение, параметры и возвращаемое значение для следующих функций: peek (), putback (), gcount (), ignore (), seekg (), seekp (), tellg (), tellp (), close (), remove (), rename ().
9) Поясните процесс обработки данных, схемы алгоритмов и тексты функций программы вашей лабораторной работы.

2 ЛАБОРАТОРНАЯ РАБОТА № 13

Разработка шаблонов абстрактных типов данных с перегрузкой в них ряда операций.

2.1. Цель лабораторной работы

Целью лабораторной работы является получение практических навыков обобщенного программирования с использованием механизма шаблонов классов и функций для создания абстрактных типов данных и перегрузки стандартных операций для этих типов.

2.2. Теоретические сведения

Абстрактные типы данных (АТД)

Используемые в приложениях структуры данных часто содержат огромное количество разнотипной информации. Например, файл персональных данных может содержать записи с именами, адресами и другой информацией о служащих; и вполне возможно, что каждая запись должна принадлежать к структуре данных, используемой для поиска информации об отдельных служащих, и к структуре данных, используемой для ответа на запросы статистического характера, и т.д.

Несмотря на это разнообразие и сложность, в большом классе программ производятся обобщенные действия с объектами данных, а доступ к информации, связанной с этими объектами, требуется только в ограниченном числе особых случаев. Многие из этих действий являются естественным продолжением базовых вычислительных процедур, поэтому они востребованы в самых разнообразных приложениях.

Многочисленные АТД, связанны с подобными манипуляциями. В этих АТД определены различные операции с коллекциями абстрактных объектов, не зависящие от типа самих объектов.

Концепция АТД, наряду с объектно-ориентированным подходом, является в настоящее время наиболее популярной методологией написания программ.

В объектно-ориентированном и обобщенном программировании C++ используются классы и шаблоны классов с целью построения АТД для обобщенных объектов данных. Создание АТД для объектов обобщенного типа ***Item***, позволяющего писать клиентские программы, в которых объекты ***Item*** используются точно так же, как и встроенные типы данных. При этом необходимо явно определять в классе ***Item*** операции, которые нужны для работы с обобщенными объектами в обобщенных алгоритмах.

Таким образом, после реализации класса ***Item*** для обобщенных объектов используется механизм шаблонов языка C++ для написания кода, который является обобщенным относительно типов объектов.

Разобравшись с классами обобщенных объектов, можно перейти к рассмотрению коллекций (collection) объектов. Многие структуры данных и алгоритмы, которые будут рассмотрены, применяются для реализации фундаментальных АТД, представляющих собой коллекции абстрактных объектов и создаваемых с помощью двух следующих операций:

- вставить новый объект в коллекцию.
- удалить объект из коллекции.

В языке C++ классы, реализующие коллекции абстрактных объектов, называются контейнерными классами (***container class***). Некоторые из структур данных, которые будут рассмотрены ниже, реализованы в библиотеке языка C++ или ее расширениях (стандартной библиотеке шаблонов - Standard Template Library).

Шаблоны функций и классов

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.

Шаблоны функций

Цель определения шаблона семейства функций – автоматизация определения одноименных функции, обрабатывающих разные типы данных.

Шаблон определяется один, но в нем определение функции параметризуется. Параметризовать в шаблоне функций 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;}
```

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, как 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.

Механизм шаблонов позволяет автоматизировать подготовку определений перегруженных функций. Компилятор, автоматически анализируя вызовы шаблонных функций, формирует необходимые определения именно для таких типов аргументов, которые использовались в вызовах.

Процесс автоматического определения типов аргументов шаблона функций по типам аргументов в конкретном обращении к шаблонной функции называется ***выведением (deduction) аргументов шаблона функций***.

Перечислим основные свойства параметров шаблона.

- Список параметров шаблона не может быть пустым.
- Кроме типизирующих параметров, у шаблона функций могут быть нетипизирующие параметры.
- 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.
- Имена параметров должны быть уникальными в определении шаблона. Имя параметра шаблона скрывает другие использования того же идентификатора в области, глобальной по отношению к шаблону.

Стандарт определяет общий формат обращения к шаблонным функциям с явным указанием фактических параметров шаблона:

имя_шаблонной_функции<список_фактических_параметров_шаблона>
(список_фактических_параметров_шаблонной_функции);

Шаблоны классов

Шаблоны классов, которые называют родовыми или параметризированными типами, позволяют создавать семейство родственных классов. Определение шаблонного (обобщенного, родового) класса имеет вид:

template <список_параметров_шаблона>
спецификация_шаблонного_класса

Как и параметры шаблона функции, параметры шаблона класса и соответствующие им аргументы, могут быть трех видов:

- *типизирующие (задающие тип в спецификации класса),*
- *нетипизирующие,*
- *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.

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 программе.

Инстанцирование (актуализация) шаблона классов – это генерация определения конкретного класса.

Общая форма объявления параметризованного класса:

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 с параметрами 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>&)

Отметим еще раз, имя шаблонного класса (в данном примере) – это ***point***, а классу, вводимому приведенным шаблоном, соответствует тип *point<T>*.

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

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 шаблона ***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>
возвращаемый_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 point <T> :: show ()  //тип, пространство имен, 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.

Рассмотрим пример шаблона 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"; return;}
return data[i];}
}; //конец шаблона классов массивов
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

Абстрактные типы данных (обобщенные контейнерные классы) для списков и векторов и примеры их реализаций были подробно рассмотрены на лекции «Шаблоны списков и векторов». Выше был рассмотрен обобщенный контейнерный класс и его реализация для массивов ограниченной размерности. На лекции «Связанные структуры данных» рассмотрены структура и функционал стека и очереди. Для этих структур надо создать обобщенный класс, используя механизмы шаблонов и перегрузки операций.

2.3 Задание на выполнение лабораторной работы

Создать шаблон заданного контейнерного класса и реализовать его для данных различных типов.

2.4. Порядок выполнения работы

1) Определить шаблон заданного класса, в соответствии с вариантом. Определить конструкторы, деструктор, перегруженную операцию присваивания ("=") и операции, заданные в варианте задания.

2) Написать программу тестирования, в которой проверяется использование шаблона для стандартных типов данных. Выполнить тестирование.

3) Определить пользовательский класс, который будет использоваться в качестве параметра шаблона. Определить в классе необходимые функции и перегруженные операции.

4) Написать программу тестирования, в которой проверяется использование шаблона для пользовательского типа. Выполнить тестирование.

2.5. Методические указания

1) Класс одномерный массив реализовать

1. Как динамический массив (вектор), для этого определение класса должно иметь следующие поля:

- указатель на начало массива;
- текущий размер массива.

2. Как массивов ограниченной размерности, должен иметь паля:

- одномерный массив и размер массива,
- размер массива определяется нетипизирующем параметром шаблона.

2) Для ввода и вывода определить в классе функции ***input*** и ***print.***

3) Аккуратно работать с константными объектами. Например:

- конструктор копирования следует определить так:

MyTmp (const MyTmp& ob);

- операцию присваивания перегрузить так:

MyTmp& operator = (const MyTmp& ob);

4) Для шаблонов списков, стеков и очередей в качестве стандартных типов использовать символьные, целые и вещественные типы. Для пользовательского типа взять класс из лабораторной работы № 8.

5) Для шаблонов массивов в качестве стандартных типов использовать целые и вещественные типы. Для пользовательского типа взять пользовательский класс "комплексное число" - ***complex***.

```
class complex {
int re, im;          // действительная и мнимая части
public:  // необходимые функции и перегруженные операции
};
```

6) Определение шаблона следует разместить вместе с определением пользовательского класса в заголовочном файле.

7) Тестирование должно быть выполнено для всех типов данных и для всех операций.

2.6. Содержание отчета

1) Титульный лист: название дисциплины, номер варианта и наименование работы, фамилия, имя, отчество студента, дата выполнения.

2) Краткие теоретические сведения.

3) Техническое задание в соответствии с вариантом работы.

4) Следует дать конкретное задание, то есть указать шаблон, какого класса должен быть создан, какие должны быть в нем конструкторы, компонентные функции, перегруженные операции и т.д.

5) То же самое следует указать для пользовательского класса.

6) Определение шаблона класса с комментариями.

7) Определение пользовательского класса с комментариями.

8) Реализация конструкторов, деструктора, операции присваивания и перегружаемых операций, которые заданы в варианте задания.

9) То же самое для пользовательского класса.

10) Реализация перегруженных операций.

11) Результаты тестирования (экранные формы). Следует указать для каких типов, и какие операции проверены.

2.7. Контрольные вопросы

1) В чем смысл использования шаблонов?
2) Синтаксис/семантика шаблонов функций
3) Синтаксис/семантика шаблонов классов
4) Определить параметризованную функцию сортировки массива методом обмена.
5) Определить различия и схожесть обобщенных контейнерных классов одномерных массивов, векторов, стеков, очередей, списков.
6) Что такое параметры шаблона функции?
7) Перечислите основные свойства параметров шаблона функции.
8) Можно ли перегружать параметризованные функции?
9) Перечислите основные свойства параметризованных классов.
10) Может ли быть пустым список параметров шаблона? Объясните.
11) Все ли компонентные функции параметризованного класса являются параметризованными?
12) Являются ли дружественные функции, описанные в параметризованном классе, параметризованными?
13) Могут ли шаблоны классов содержать виртуальные компонентные функции?
14) Как определяются компонентные функции параметризованных классов вне определения шаблона класса?
15) Каковы синтаксис/семантика “операции-функции”?
16) Когда нужно перегружать операцию присваивания для определенного пользователем типа данных, например класса?
17) Можно ли изменить приоритет перегруженной операции?
18) Можно ли изменить количество операндов перегруженной операции?
19) Можно ли, используя дружественную функцию, перегрузить оператор присваивания?
20) Все ли операции языка C++ могут быть перегружены?
21) Все ли операции можно перегрузить с помощью глобальной дружественной функции?

2.8. Варианты заданий лабораторной работы

1) Класс - одномерный массив. Дополнительно перегрузить следующие операции:
   * - умножение массивов;
   [] - доступ по индексу.
2) Класс - одномерный массив. Дополнительно перегрузить следующие операции:
   int () - размер массива;
   [] - доступ по индексу.
3) Класс - одномерный массив. Дополнительно перегрузить следующие операции:

[] - доступ по индексу;
= = - проверка на равенство;
!= - проверка на неравенство.

4) Класс - однонаправленный список list**.** Дополнительно перегрузить следующие операции:
   + - добавить элемент в начало (list+item);
   -- - удалить элемент из начала (--list);
   = = - проверка на равенство.
5) Класс - однонаправленный список list. Дополнительно перегрузить следующие операции:
   + - добавить элемент в начало (item+list);
   -- - удалить элемент из начала (--list);
   != - проверка на неравенство.
6) Класс - однонаправленный список list. Дополнительно перегрузить следующие операции:
   + - добавить элемент в конец (list+item);
   -- - удалить элемент из конца (типа list--);
   != - проверка на неравенство.
7) Класс - однонаправленный список list. Дополнительно перегрузить следующие операции:
   [] - доступ к элементу в заданной позиции, например:
   Type c;
   int i;
   list L;
   c=L[i];
   + - объединить два списка;
   = = - проверка на равенство.
8) Класс - однонаправленный список list. Дополнительно перегрузить следующие операции:
   [] - доступ к элементу в заданной позиции, например:
   int i; Type c;
   list L;
   c=L[i];
   + - объединить два списка;
   != - проверка на неравенство.
9) Класс - однонаправленный список list. Дополнительно перегрузить следующие операции:
   () - удалить элемент в заданной позиции, например:
   int i;
   list L;
   L(i);
   () - добавить элемент в заданную позицию, например:
   int i;

Type c;
list L;
L(c,i);
!= - проверка на неравенство.

10)Класс - стек stack. Дополнительно перегрузить следующие операции:
+ - добавить элемент в стек;
-- - извлечь элемент из стека;
bool () - проверка, пустой ли стек.

11) Класс - очередь queue. Дополнительно перегрузить следующие операции:
+ - добавить элемент;
-- - извлечь элемент;
bool () - проверка, пустая ли очередь.

12) Класс - одномерный массив. Дополнительно перегрузить следующие операции:
+ - сложение массивов;
[] - доступ по индексу;
+ - сложить элемент с массивом.

13) Класс - одномерный массив. Дополнительно перегрузить следующие операции:
- - вычитание массивов;
[] - доступ по индексу;
- - вычесть из массива элемент.

3. ЛАБОРАТОРНАЯ РАБОТА № 14

Демонстрация использования алгоритмов STL для обработки элементов контейнерных классов, встроенных и пользовательских типов.

3.1. Цель лабораторной работы

Освоить технологию обобщенного программирования с использованием библиотеки стандартных шаблонов (***STL***) языка C++.

3.2. Теоретические сведения

Стандартная библиотека шаблонов (STL)

Назначение ***STL*** - обеспечивать программиста типовыми структурами данных и наиболее эффективными алгоритмами, настроенными на обработку информации, представленной этими данными.

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 абстрактных (параметризированных) типов данных.

STL обеспечивает общецелевые, стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных. ***STL*** строится на основе шаблонов классов, и поэтому входящие в неё алгоритмы и структуры применимы почти ко всем типам данных.

Состав STL

В целом, ***STL*** состоит из двух основных частей:

- ***классов контейнеров***, пригодных для хранения элементов разных типов
- ***набора обобщенных алгоритмов*** для выполнения различных операций над контейнерами, над их элементами.

Алгоритмы ***STL*** называют обобщенными потому, что они не зависят 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.

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

Контейнеры

Контейнеры (containers) — это коллекции (динамические структуры данных), содержащие другие однотипные объекты.

Контейнерные классы являются шаблонными, 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 многие другие. Использование контейнеров позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки.

Алгоритмы

Обобщенные алгоритмы реализуют большое количество процедур, применимых к контейнерам — например, поиск, сортировку, слияние и другие. 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*** и ***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,
- 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 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 данного контейнера.

Контейнер содержит элементы последовательности, начало, и конец последовательности представляются значениями итераторов. Эти значения доступны с помощью методов любого контейнерного класса:

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

end () - возвращает значение итератора, установленного за последним элементом последовательности.

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 итератора 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 имени шаблонного класса**,** 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 всех 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 большое: ***std::list <double>::iterator*;**

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

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.

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.

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 ни изучить ***STL***.

Каждый раз, когда нужно ввести очередной элемент информации, используйте оператор ++ точно так же, как с основными итераторами. По потоку перемещаемся как по контейнеру! 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;}
```

Продолжим рассмотрение контейнеров

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, которые адаптируют методы других контейнеров — ***стеки (stack), очереди (queue)*** и ***очереди с приоритетами (priority_queue).***

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.

Общие свойства контейнеров

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

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

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 табл. 2.

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)

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

Таблица 2 (продолжение)

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)

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:

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

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

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

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 коротко на каждом из них.

Каждый из контейнеров имеет большое число методов (табл. 3), решающих схожие задачи, но по-своему. Каждый имеет набор конструкторов для создания объектов. Подробно это описано в лекции «Контейнеры STL»

Пример 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:
 vector <int> v; // Создание пустого вектора
 vector <int> v1(3);  //Создание вектора из трех элементов, со
                      //значениями по умолчанию (0).
vector <int> v2(5, 2); // Создание вектора из 5 элементов, со значениями 2
vector <int> v4(v2); // Создание вектора v4 - копии вектора     v2
vector <int> v5(5);   // Создание временного пустого вектора для
                        //демонстрации копирования диапазона
for (auto i: v5) {
   v5[i] = i;}
vector <int> v6(v5.begin () + 1, v5.begin () + 3); // Создание v6
 //копированием диапазона элементов вектора v5[first, last)
//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;}
```

Таблица 3.

Методы, которые поддерживают последовательные контейнеры

vector		**deque**		**list**	
push_back()	добавление в конец	push_back(T&key)	добавление в конец	push_back(T&key)	добавление в конец
pop_back()	удаление из конца	pop_back()	удаление из конца	pop_back()	удаление из конца
insert	Вставка в произвольное место	push_front(T&key)	добавление в начало	push_front(T&key)	добавление в начало
erase	удаление из произвольного места	pop_front()	удаление из начала	pop_front()	удаление из начала
[] at	доступ к произвольному элементу	insert	Вставка в произвольное место	insert	Вставка в произвольное место
		erase	удаление из произвольного места	erase	удаление из произвольного места
		[] at	доступ к произвольному элементу		
swap	обмен векторов			swap	обмен списков
clear()	очистить вектор			clear()	очистить список
				splice	сцепка списков

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.

Стек

Интерфейс класса ***stack*** формируется из методов класса-прототипа. 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 этому классу.

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 тем, что ***концепция итераторов*** в стеке вообще не поддерживается.

Очередь

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

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

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 <.

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 (map)

Словарь построен на основе пар значений, первое из которых представляет собой ключ для идентификации элемента, а второе — собственно элемент. Ключ ассоциирован с элементом, откуда и произошло название этих контейнеров. 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>.***

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 в 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.

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 в множестве в порядке их занесения. Функция ***find*** возвращает ***итератор на первый найденный элемент*** или ***end (),*** если ни одного элемента с заданным ключом не найдено.

Обобщенные алгоритмы STL

Каждый алгоритм реализован в виде шаблона или набора шаблонов функций, поэтому может работать с различными видами последовательностей и данными разнообразных типов.

Для ***настройки алгоритма на конкретные требования пользователя*** применяются ***функциональные объекты***.

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***.

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. 4

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 ().***

Таблица 4

Не модифицирующие операции с последовательностями

Алгоритм	Выполняемая функция
adjacent_find	Нахождение пары соседних значений
count	Подсчет количества вхождений значения в последовательность
count_if	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

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 программе добавить определение глобальной функции:

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

возвращающая истину, если целое значение попадает между 2 и 5.

Работая с целочисленным контейнером ***v***, вызов функции ***count_if*** :

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

определит, сколько таких элементов содержит контейнер.

Модифицирующие операции с последовательностями представлены в табл. 5. 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 5

Модифицирующие операции

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 n 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

Алгоритмы, связанные с сортировкой

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

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 свой метод ***sort (),*** решающий задачу сортировки.

Алгоритм sort имеет две сигнатуры:

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

tempiate<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. Сортировка по умолчанию— это сортировка по возрастанию значений.

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 функция ***operator ().***

В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения, встроенных в язык C++. Они возвращают значение типа ***bool***, то есть являются предикатами. Стандартные предекаты представлены в табл. 6.

Таблица 6

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> ().***

Если произвести следующий вызов sort:

sort (v.begin (), v.end(), greater<double> ());
вектор ***v*** 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.***

Алгоритмы всех категорий и настроек подробно с примерами рассмотрены в лекции «Алгоритмы STL».

3.3.Задание на выполнение лабораторной работы

Написать программу с использованием STL, состоящую из двух частей. Первая часть должна демонстрировать работу с контейнерными классами, используя только их методы, вторая - использование алгоритмов STL.

3.4. Порядок выполнения работы

Часть 1

1) Создать объект-контейнер в соответствии с вариантом задания и заполнить его данными, тип которых определяется вариантом задания. Вывести элементы контейнера.
2) Изменить контейнер, удалив из него одни элементы и заменив другие.
3) Создать второй контейнер этого же класса и заполнить его данными того же типа, что и первый контейнер.
4) Изменить первый контейнер, удалив из него n элементов после заданного и добавив затем в него все элементы из второго контейнера. Вывести элементы первого и второго контейнеров
5) Выполнить то же самое для данных пользовательского типа.

Часть2

1) Отсортировать контейнер, содержащий объекты пользовательского типа по убыванию элементов.
2) Используя подходящий алгоритм, найти в контейнере элемент, удовлетворяющий заданному условию.
3) Переместить элементы, удовлетворяющие заданному условию в другой (предварительно пустой) контейнер. Тип второго контейнера определяется вариантом задания. Вывести элементы второго контейнера.
4) Отсортировать оба контейнеры по возрастанию элементов. Вывести.
5) Создать третий контейнер путем слияния первых двух.
6) Вывести, количество элементов третьего контейнера, удовлетворяющих заданному условию. Вывести и сами элементы.
7) Определить, есть ли в третьем контейнере элемент, удовлетворяющий заданному условию.

3.5. Методические указания

1) В качестве пользовательского типа данных определить пользовательский класс.

2) Элементы контейнера загружать из потока.

3) Для вставки и удаления элементов контейнера в программе части 1 использовать соответствующие операции, определенные в классе контейнера.

4) Для создания второго контейнера в части 2 можно использовать либо алгоритм ***remove_copy_if***, либо определить свой алгоритм ***copy_if***, которого нет в ***STL***.

5) Для поиска элемента в коллекции можно использовать алгоритм ***find_if***, либо ***for_each***, либо ***binary_search***, если контейнер отсортирован.

6) Для сравнения элементов при сортировке по возрастанию используется операция <, которая должна быть перегружена в пользовательском классе. Для сортировки по убыванию следует определить функцию ***comp*** и использовать вторую версию алгоритма ***sort***.

7) Условия поиска и замены элементов выбираются самостоятельно и для них пишется функция-предикат.

8) Для ввода/вывода объектов пользовательского класса следует перегрузить операции “>>” и “ <<”.

9) Некоторые алгоритмы могут не поддерживать используемые в вашей программе контейнеры. Например, алгоритм ***sort*** не поддерживает контейнеры, которые не имеют итераторов произвольного доступа. В этом случае следует написать свой алгоритм. Например, для стека алгоритм сортировки может выполняться следующим образом: переписать стек в вектор, отсортировать вектор, переписать вектор в стек.

10) При перемещении элементов ассоциативного контейнера в не ассоциативный перемещаются только данные (ключи не перемещаются). И наоборот, при перемещении элементов не ассоциативного контейнера в ассоциативный должен быть сформирован ключ.

3.6. Контрольные вопросы

1) Назначение, состав STL.

2) Определение контейнера. Контейнеры STL. Общие свойства и различия контейнеров.

3) Понятия итератора. Назначение, определение, операции с итераторами. Виды итераторов. Контейнеры и итераторы.

4) Алгоритмы STL. Как достигается их универсальность, возможность применять функции не только к объектам различных контейнерных классов, но и к массивам, любым последовательностям.

5) Алгоритмы STL. 4 группы алгоритмов. Дать примеры из каждой группы. Алгоритмы и итераторы.

6) Алгоритмы и функциональные объекты.

7) Последовательные и ассоциативные контейнеры.

3.7. Варианты заданий лабораторной работы

№ п/п	Первый контейнер	Второй контейнер	Встроенный тип данных
1	vector	list	int
2	list	deque	long
3	deque	stack	float
4	stack	queue	double
5	queue	vector	char
6	vector	stack	string
7	map	list	long
8	multimap	deque	float
9	set	stack	int
10	multiset	queue	char
11	vector	map	double
12	list	set	int
13	deque	multiset	long
14	stack	vector	float
15	queue	map	int
16	priority-queue	stack	char
17	map	queue	char
18	multimap	list	int
19	set	map	char
20	multiset	vector	int

4. СПИСОК ЛИТЕРАТУРЫ

1) Надейкина Л.А. Программирование на языке высокого уровня. Часть 1. Учебное пособие. - М: МГТУ ГА, 2012, 84 с.

2) Надейкина Л.А. Программирование. Часть 2. Учебное пособие. - М: МГТУ ГА, 2017, 84 с.

3) Надейкина Л.А. Программирование. Обобщенное программирование. Учебное пособие. – Воронеж ООО «МИР», 2019, 80 с.

4) Подбельский В.В. Стандартный Си++. М.: Финансы и статистика, 2008, 688с.

5) Павловская Т.А. С/ С++. Программирование на языке высокого уровня - СПб: Питер, 2011. – 461 с.

СОДЕРЖАНИЕ

1. Лабораторная работа № 12
Обработка данных текстовых и бинарных файлов. 3
1.1. Цель лабораторной работы 3
1.2. Теоретические сведения 3
1.3. Задание на выполнение лабораторной работы 9
1.4. Порядок выполнения работы 10
1.5. Пример выполнения лабораторной работы 10
1.6. Контрольные вопросы 19

2. Лабораторная работа № 13
Разработка шаблонов абстрактных типов данных с перегрузкой в них ряда операций. 19
2.1. Цель лабораторной работы 19
2.2. Теоретические сведения 19
2.3. Задание на выполнение лабораторной работы 26
2.4. Порядок выполнения работы 26
2.5. Методические указания 26
2.6. Содержание отчета 27
2.7. Контрольные вопросы 28
2.8. Варианты заданий лабораторной работы 28

3. Лабораторная работа № 14
Демонстрация использования алгоритмов STL для обработки элементов контейнерных классов, встроенных и пользовательских типов. 30
3.1. Цель лабораторной работы 30
3.2. Теоретические сведения 30
3.3. Задание на выполнение лабораторной работы 45
3.4. Порядок выполнения работы 45
3.5. Методические указания 46
3.6. Контрольные вопросы 46
3.7. Варианты заданий лабораторной работы 47

4. СПИСОК ЛИТЕРАТУРЫ 47